mrjob Documentation

Release 0.7.4

Steve Johnson

September 17, 2020

Contents

mrjob lets you write MapReduce jobs in Python 2.7/3.4+ 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)
- Run in the cloud using Google Cloud Dataproc (Dataproc)
- Easily run Spark jobs on EMR or your own Hadoop cluster

To get started, install with `pip`:

```
pip install mrjob
```

and begin reading the tutorial below.

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 `mr_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 `mr_word_count.py`), and try this:

```
$ python mr_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 Dataproc configured (see Dataproc Quickstart), you can run it there with `-r dataproc`.

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

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

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 *Uploading your source tree*.

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()* to return a list of *MRStep*s.

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-2
  inline:
    local_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" (Hadoop 1) or "resource manager" (Hadoop 2) and a number of worker nodes. The master is responsible for scheduling the jobs' component tasks on the worker nodes and re-executing the failed tasks. The worker nodes 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\tvalue2\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. Your mappers and reducers are not called in this process at all [1].

This process creates a runner (see *`MRJobRunner`*), which then sends the job to Hadoop [2].

It tells 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`.

If you have a multi-step job, `--step-num` helps your script know which step is being run.

When Hadoop distributes tasks among the task nodes, Hadoop Streaming will use the appropriate command to process the data it is given.

Note: Prior to v0.6.7, your job would *also* run itself locally with the `--steps` switch, to get a JSON representation of the job's step. Jobs now pass that representation directly to the runner when they instantiate it. See mrjob.step - represent Job Steps for more information.

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 when using the `local` runner, a simulation of Hadoop.

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 *steps()* to return a list of *MRStep*s:

```
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 *MRStep* 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 *`MRStep`*.)

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 a shell 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, so if you want to use pipes, etc, you'll need to run them in a subshell. For example:

```
class DemoJob(MRJob):

    def mapper_cmd(self):
        return 'sh -c "grep '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 command 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.

1.4.2 Protocols

Hadoop streaming assumes that all data is newline-delimited bytes. By default, mrjob assumes all output is in JSON format, but it can actually read and write lines in any format by using protocols.

(If you need to read non-line-based data, see *Passing entire files to the mapper*, below.)

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 just reads in a line as a `str`. (The line won't have a trailing newline character because *MRJob* strips it.) So by default, the first step in your job sees `(None, line)` for each line of input [3].

The default output and internal protocols are both *JSONProtocol* [4], which reads and writes JSON strings separated by a tab character. (By default, Hadoop Streaming uses the tab character to separate keys and values within one line when it sorts your data.)

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 whatever comes before the first tab character.

See *mrjob.protocol* for the full list of protocols built-in to mrjob.

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.

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
```

[3] Experienced Pythonistas might notice that a `str` is a bytestring on Python 2, but Unicode on Python 3. That's right! *RawValueProtocol* is an alias for one of two different protocols depending on your Python version.

[4] *JSONProtocol* is an alias for one of four different implementations; we try to use the (much faster) `ujson` library if it is available, and if not, `rapidjson` or `simplejson` before falling back to the built-in `json` implementation.

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_args(self):
        super(CommandLineProtocolJob, self).configure_args()
        self.add_passthru_arg(
            '--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 bytestring and returns a 2-tuple of decoded objects, and `write()` takes the key and value and returns bytes to be passed back to Hadoop Streaming or as output.

Protocols don't have to worry about adding or stripping newlines; this is handled automatically by *MRJob*.

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.

1.4.3 Passing entire files to the mapper

New in version 0.6.3.

Sometimes you need to read binary data (e.g. image files), or text-based data that has records longer than one line.

By using *mapper_raw()*, you can pass entire files to your mapper, and read them however you want. Each mapper gets one file, and is passed both the path of a local copy of the file, and the URI where the original file is located on Hadoop's filesystem.

For example, if you want to read `.wet` files from Common Crawl data, you could handle them like this:

```
class MRCrawler(MRJob):

    def mapper_raw(self, wet_path, wet_uri):
        from warcio.archiveiterator import ArchiveIterator

        with open(wet_path, 'rb') as f:
            for record in ArchiveIterator(f):
                ...
```

To use a library like `warcio`, you'll need to ensure that it gets installed on your cluster. See *Using a virtualenv* for one way to do this.

Under the hood, mrjob is passes an input manifest (a list of URIs of input files) to Hadoop, and instructs Hadoop to send one line to each mapper. In most cases, this should be seamless, even to the point of telling you which file was being read when a task fails.

Warning: For all runners except EMR, mrjob uses **`hadoop fs`** to download files to the local filesystem, which means Hadoop has to invoke itself. If your cluster has tightly tuned memory requirements, this can sometimes cause an out-of-memory error.

1.4.4 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 *mrjob.step.INPUT* and *mrjob.step.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', INPUT, OUTPUT]
            )
        ]
```

Changed in version 0.6.6: mrjob no longer passes hadoop generic args (*-D* and *-libjars*) to `JarSteps`. If you want them, add *mrjob.step.GENERIC_ARGS* to your `JarStep`'s *args*, and mrjob will automatically interpolate them.

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 *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.

1.4.5 Using other python modules and packages

New in version 0.6.4.

If you want to run Python code outside of the file containing your *MRJob*, you'll to make sure that code gets uploaded to Hadoop.

The easiest way to do this is with by setting the *DIRS* attribute in your job. Put the code you want to import in one or more packages (directories with an `__init__.py` file), and point *DIRS* at them:

```
class MRPackageUsingJob(MRJob):

  DIRS = ['mycode', '../someothercode']

  ...
```

And then import code from inside a mapper or reducer:

```
def mapper(self, key, value):
  from mycode.custom import important_business_logic
  from someotherlibrary import util_function
  ...
```

(If you want to import code from the top level of your script rather than inside a method, make sure it's in your `PYTHONPATH`, just like with any other code.)

DIRS is relative to the directory your script is in (not the current working directory). This works inside Hadoop because the current working directory is the same as the directory your script is in.

If you want to access individual Python modules or other support code, you can use *FILES* to upload them to your job's working directory inside Hadoop:

```
class MRFileUsingJob(MRJob):

  FILES = ['mymodule.py', '../data/zipcodes.db']

  def mapper(self, key, value):
    from mymodule import open_zipcode_db
```

```
    with open_zipcode_db('zipcodes.db') as db:
      ...
```

For jobs with more complex dependencies (e.g. code that needs to be compiled), you may need to use the setup option. See Job Environment Setup Cookbook for more information.

1.4.6 Defining command line options

Recall from *How your program is run* that your script is executed in several contexts: once for the initial invocation, 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_passthru_arg()* and *add_file_arg()*.

Passthrough options

A *passthrough option* is an `argparse` option that mrjob is aware of. mrjob inspects the value of the option when you invoke your script 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_args(self):
        super(CommandLineProtocolJob, self).configure_args()
        self.add_passthru_arg(
            '--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_passthru_arg() takes the same arguments as `argparse.ArgumentParser.add_argument()`. For more information, see the argparse docs.

Passing through existing options

Occasionally, it'll be useful for mappers, reducers, etc. to be able to see the value of other command-line options. For this, use *pass_arg_through()* with the corresponding command-line switch.

For example, you might wish to fetch supporting data for your job from different locations, depending on whether your job is running on EMR or locally:

```
class MRRunnerAwareJob(MRJob):

    def configure_args(self):
        super(MRRunnerAwareJob, self).configure_args()

        self.pass_arg_through('--runner')
```

```
    def mapper_init(self):
        if self.options.runner == 'emr':
            self.data = ...  # load from S3
        else:
            self.data = ... # load from local FS
```

Note: Keep in mind that `self.options.runner` (and the values of most options) will be `None` unless the user explicitly set them with a command-line switch.

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_args(self):
        super(SqliteJob, self).configure_args()
        self.add_file_arg('--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
```

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.

1.4.7 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
```

1.4.8 Input and output formats

Input and output formats are Java classes that determine how your job interfaces with data on Hadoop's filesystem(s).

Suppose we wanted to write a word frequency count job that wrote output into a separate directory based on the first letter of the word counted (`a/part-*`, `b/part-*`, etc.). We could accomplish this by using the `MultipleValueOutputFormat` class from the Open Source project nicknack.

First, we need to tell our job to use the custom output format by setting *`HADOOP_OUTPUT_FORMAT`* in our job class:

```
HADOOP_OUTPUT_FORMAT = 'nicknack.MultipleValueOutputFormat'
```

The output format class is part of a custom JAR, so we need to make sure that this JAR gets included in Hadoop's classpath. First download the jar to the same directory as your script, and add its name to *`LIBJARS`*:

```
LIBJARS = ['nicknack-1.0.0.jar']
```

(You can skip this step if you're using a format class that's built into Hadoop.)

Finally, output your data the way that your output format expects. `MultipleValueOutputFormat` expects the subdirectory name, followed by a tab, followed the actual line to write into the file.

First, we need to take direct control of how the job writes output by setting *`OUTPUT_PROTOCOL`* to *`RawValueProtocol`*:

```
OUTPUT_PROTOCOL = RawValueProtocol
```

Then we need to format the line accordingly. In this case, let's continue output our final data in the standard format (two JSONs separated by a tab):

```
def reducer(self, word, counts):
    total = sum(counts)
    yield None, '\t'.join([word[0], json.dumps(word), json.dumps(total)])
```

Done! Here's the full, working job (this is `mrjob.examples.mr_nick_nack`):

```
import json
import re

from mrjob.job import MRJob
from mrjob.protocol import RawValueProtocol

WORD_RE = re.compile(r"[A-Za-z]+")


class MRNickNack(MRJob):

    HADOOP_OUTPUT_FORMAT = 'nicknack.MultipleValueOutputFormat'
```

```
    LIBJARS = ['nicknack-1.0.0.jar']

    OUTPUT_PROTOCOL = RawValueProtocol

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

    def reducer(self, word, counts):
        total = sum(counts)
        yield None, '\t'.join([word[0], json.dumps(word), json.dumps(total)])


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

Input formats work the same way; just set *HADOOP_INPUT_FORMAT*. (You usually won't need to set *INPUT_PROTOCOL* because it already defaults to *RawValueProtocol*.)

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 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:
  - 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: `DataprocJobRunner`, *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/docs/current/)
- Run your job with `-r hadoop`:

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

Note: You don't need to install `mrjob` or any other libraries on the nodes of your Hadoop cluster, but they *do* at least need a version of Python that's compatible with your job.

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 Running on Dataproc

- Set up your Google account and credentials (see *Getting started with Google Cloud*)
- Run your job with `-r dataproc`:

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

Note: Dataproc does not yet support Spark or libjars.

1.5.5 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.

1.5.6 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).

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

You instantiate the *MRJob*, use a context manager to create the runner, run the job, and cat its output, parsing that output with the job's output protocol.

Further reference:

- *make_runner()*
- *run()*
- *parse_output()*
- *cat_output()*

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)

# no, stop, what are you doing?!?!
mr_job = MyJob(args=[args])
with mr_job.make_runner() as runner:
    runner.run()
    # ... etc
```

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 io import BytesIO
from unittest import TestCase

from tests.mr_counting_job import MRCountingJob


class CounterTestCase(TestCase):

    def test_counters(self):
        stdin = BytesIO(b'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 Spark

1.6.1 Why use mrjob with Spark?

mrjob augments Spark's native Python support with the following features familiar to users of mrjob:

- automatically upload input and other support files to HDFS, GCS, or S3 (see upload_files, upload_archives, and py_files)
- run **`make`** and other command before running Spark tasks (see setup).
- passthrough and file arguments (see *Defining command line options*)
- automatically parse logs to explain errors and other Spark job failures
- easily pass through environment variables (see cmdenv)
- support for libjars
- automatic matching of Python version (see python_bin)
- automatically set up Spark on EMR (see bootstrap_spark)
- automatically making the mrjob library available to your job (see bootstrap_mrjob)

1.6.2 mrjob spark-submit

If you already have a Spark script written, the easiest way to access mrjob's features is to run your job with **`mrjob spark-submit`**, just like you would normally run it with **`spark-submit`**. This can, for instance, make running a Spark job on EMR as easy as running it locally, or allow you to access features (e.g. setup) not natively supported by Spark.

For more details, see *mrjob spark-submit*.

1.6.3 Writing your first Spark MRJob

Another way to integrate mrjob with Spark is to add a *spark()* method to your *MRJob* class, and put your Spark code inside it. This will allow you to access features only availble to MRJobs (e.g. *FILES*).

Here's how you'd implement a word frequency count job in Spark:

```
import re
from operator import add

from mrjob.job import MRJob

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


class MRSparkWordcount(MRJob):

    def spark(self, input_path, output_path):
        # Spark may not be available where script is launched
        from pyspark import SparkContext

        sc = SparkContext(appName='mrjob Spark wordcount script')

        lines = sc.textFile(input_path)

        counts = (
            lines.flatMap(self.get_words)
            .map(lambda word: (word, 1))
            .reduceByKey(add))

        counts.saveAsTextFile(output_path)

        sc.stop()

    def get_words(self, line):
        return WORD_RE.findall(line)


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

Since Spark already supports Python, mrjob takes care of setting up your cluster, passes in input and output paths, and otherwise gets out of the way. If you pass in multiple input paths, *input_path* will be these paths joined by a comma (`SparkContext.textFile()` will accept this).

Note that `pyspark` is imported *inside* the *spark()* method. This allows your job to run whether `pyspark` is installed locally or not.

The *spark()* method can be used to execute arbitrary code, so there's nothing stopping you from using *SparkSession* instead of *SparkContext* in Spark 2, or writing a streaming-mode job rather than a batch one.

> **Warning:** Prior to v0.6.8, to pass job methods into Spark (e.g. `rdd.flatMap(self.get_words)`), you first had to call *self.sandbox()*; otherwise Spark would error because *self* was not serializable.

1.6.4 Running on your Spark cluster

By default, mrjob runs your job on the inline runner (see below). If you want to run your job on your own Spark cluster, run it with `-r spark`:

Use `--spark-master` (see spark_master) to control where your job runs.

You can pass in spark options with `-D` (see jobconf) and set deploy mode (client or cluster) with `--spark-deploy-mode`. If you need to pass other arguments to **`spark-submit`**, use spark_args.

The Spark runner can also run "classic" MRJobs (i.e. those made by defining *`mapper()`* etc. or with *`MRStep`*s) directly on Spark, allowing you to move off Hadoop without rewriting your jobs. See *below* for details.

Warning: If you don't set spark_master, your job will run on Spark's default `local[*]` master, which can't handle setup scripts or `--files` because it doesn't give tasks their own working directory.

Note: mrjob needs to know what master and deploy mode you're using, so it will override attempts to set spark master or deploy mode through jobconf (e.g. `-D spark.master=...`).

1.6.5 Using remote filesystems other than HDFS

By default, if you use a remote Spark master (i.e. not `local` or `local-cluster`), Spark will assume you want to use HDFS for your job's temp space, and that you will want to access it through **`hadoop fs`**.

Some Spark installations don't use HDFS at all. Fortunately, the Spark runner also supports S3 and GCS. Use spark_tmp_dir to specify a remote temp directory not on HDFS (e.g. `--spark-tmp-dir s3a://bucket/path`).

For more information on accessing S3 or GCS, see *Configuring AWS credentials* (S3) or *Configuring Google Cloud credentials* (GCS).

1.6.6 Other ways to run on Spark

Inline runner

Running your Spark job with `-r inline` (the default) will launch it directly through the `pyspark` library, effectively running it on the `local[*]` master. This is convenient for debugging because exceptions will bubble up directly to your Python process.

The inline runner also builds a simulated working directory for your job, making it possible to test scripts that rely on certain files being in the working directory (it doesn't run setup scripts).

Note: If you don't have a local Spark installation, the pyspark library on PyPI is a pretty quick way to get one (`pip install pyspark`).

Local runner

Running your Spark job with `-r local` will launch it through **`spark-submit`** on a `local-cluster` master. `local-cluster` is designed to simulate a real Spark cluster, so setup will work as expected.

By default, the local runner launches Spark jobs with as many executors as your system has CPUs. Use `--num-cores` (see num_cores to change this).

By default, the local runner gives each executor 1 GB of memory. If you need more, you can specify it through jobconf, e.g. `-D spark.core.memory=4g`.

EMR runner

Running your Spark job with `-r emr` will launch it in Amazon Elastic MapReduce (EMR), with the same seamless integration and features mrjob provides for Hadoop jobs on EMR.

The EMR runner will always run your job on the `yarn` Spark master in `cluster` deploy mode.

Hadoop runner

Running your Spark job with `-r hadoop` will launch it on your own Hadoop cluster. This is not significantly different than the Spark runner. The main advantage of the Hadoop runner is that is has more knowledge about how to find logs and can be better at finding the relevant error if your job fails.

Unlike the Spark runner, the Hadoop runner's default spark master is `yarn`.

Note: mrjob does not yet support Spark on Google Cloud Dataproc.

1.6.7 Passing in libraries

Use `--py-files` to pass in `.zip` or `.egg` files full of Python code:

```
python your_mr_spark_job -r hadoop --py-files lib1.zip,lib2.egg
```

Or set py_files in `mrjob.conf`.

1.6.8 Command-line options

Command-line options (passthrough options, etc) work exactly like they do with regular streaming jobs (even *`add_file_arg()`* on the `local[*]` Spark master. See *Defining command line options*.

1.6.9 Uploading files to the working directory

upload_files, *`FILES`*, and files uploaded via setup scripts all should work as expected (except on `local` masters because there is no working directory).

Note that you can give files a different name in the working directory (e.g. `--files foo#bar`) on all Spark masters, even though Spark treats that as a YARN-specific feature.

1.6.10 Archives and directories

Spark treats `--archives` as a YARN-specific feature. This means that upload_archives, *`ARCHIVES`*, *`DIRS`*, etc. will be ignored on non-`yarn` Spark masters.

Future versions of mrjob may simulate archives on non-`yarn` masters using a setup script.

1.6.11 Multi-step jobs

There generally isn't a need to define multiple Spark steps (Spark lets you map/reduce as many times as you want). However, it may sometimes be useful to pre- or post-process Spark data using a *streaming* or *jar* step.

This is accomplished by overriding your job's *steps()* method and using the *SparkStep* class:

```
def steps():
    return [
        MRStep(mapper=self.preprocessing_mapper),
        SparkStep(spark=self.spark),
    ]
```

1.6.12 External Spark scripts

mrjob can also be used to launch external (non-mrjob) Spark scripts using the *SparkScriptStep* class, which specifies the path (or URI) of the script and its arguments.

As with *JarStep*s, you can interpolate input and output paths using *INPUT* and *OUTPUT* constants. For example, you could set your job's *steps()* method up like this:

```
def steps():
    return [
        SparkScriptStep(
           script=os.path.join(
               os.path.dirname(__file__), 'my_spark_script.py'),
           args=[INPUT, '-o', OUTPUT, '--other-switch'],
        ),
    ]
```

1.6.13 Custom input and output formats

mrjob allows you to use input and output formats from custom JARs with Spark, just like you can *with streaming jobs*.

First download your JAR to the same directory as your job, and add it to your job class with the *LIBJARS* attribute:

```
LIBJARS = ['nicknack-1.0.0.jar']
```

Then use Spark's own capabilities to reference your input or output format, keeping in mind the data types they expect.

For example, nicknack's `MultipleValueOutputFormat` expects `<Text,Text>`, so if we wanted to integrate it with our wordcount example, we'd have to convert the count to a string:

```
def spark(self, input_path, output_path):
    from pyspark import SparkContext

    sc = SparkContext(appName='mrjob Spark wordcount script')

    lines = sc.textFile(input_path)

    counts = (
        lines.flatMap(self.get_words
        .map(lambda word: (word, 1))
        .reduceByKey(add))

    # MultipleValueOutputFormat expects Text, Text
    # w_c is (word, count)
```

```
    counts = counts.map(lambda w_c: (w_c[0], str(w_c[1])))

    counts.saveAsHadoopFile(output_path,
                            'nicknack.MultipleValueOutputFormat')

    sc.stop()
```

1.6.14 Running "classic" MRJobs on Spark

The Spark runner provides near-total support for running "classic" *MRJob*s (the sort described in *Writing your first job* and *Writing your second job*) directly on any Spark installation, even though these jobs were originally designed to run on Hadoop Streaming. Support includes:

- **_init()* and **_final()* methods
- *HADOOP_INPUT_FORMAT* and *HADOOP_OUTPUT_FORMAT*
- *SORT_VALUES*
- *passthrough arguments*
- *increment_counter()*

Jobs will often run more quickly on Spark than Hadoop Streaming, so it's worth trying even if you don't plan to move off Hadoop in the forseeable future.

Multiple steps are run as a single job

If you have a job with multiple consecutive *MRStep*s, the Spark runner will run them all as a single Spark job. This is usually what you want (more efficient), but it can make debugging slightly more challenging (step failure exceptions give a range of steps, no way to access intermediate data).

To force the Spark runner to run steps separately, you can initialize each *MRStep* with a different `jobconf` dictionary.

No support for subprocesses

Pre-filters (e.g. *mapper_pre_filter()*) and command steps (e.g. *reducer_cmd()*) are not supported because they require launching subprocesses.

It wouldn't be *impossible* to emulate this inside Spark, but then we'd essentially be turning Spark *into* Hadoop Streaming. (If you have a use case for this seemingly implausible feature, let us know through GitHub.)

Spark loves combiners

Hadoop's "reduce" paradigm is a lot more heavyweight than Spark's; whereas a Spark reducer just wants to know how to combine two values into one, a Hadoop reducer expects to be able to see all the values for a given key, and to emit zero or more key-value pairs.

In fact, Spark reducers are a lot more like Hadoop combiners. The Spark runner knows how to translate something like:

```
def combiner(self, key, values):
    yield key, sum(values)
```

into Spark's reduce paradigm–basically it'll pass your combiner two values at a time, and hope it emits one. If your combiner does *not* behave like a Spark reducer function (emitting multiple or zero values), the Spark runner handles that gracefully as well.

Counter emulation is *almost* perfect

Counters (see *increment_counter()*) are a feature specific to Hadoop. mrjob emulates them on Spark anyway. If you have a multi-step job, mrjob will dutifully print out counters for each step and make them available through *counters()*.

The only drawback is that while Hadoop has the ability to "take back" counters produced by a failed task, there isn't a clean way to do this with Spark accumulators. Therefore, the counters produced by the Spark runner's Hadoop emulation may be overestimates.

Spark does not stream data

While Hadoop streaming (as its name implies) passes a stream of data to your job, Spark instead operates on *partitions*, which are loaded into memory.

A reducer like this can't run out of memory on Hadoop streaming, no matter how many values there are for *key*:

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

However, on Spark, simply storing the partition that contains these values can cause Spark to run out of memory.

If this happens, you can let Spark use more memory (`-D spark.executor.memory=10g`) or add a combiner to your job.

Compression emulation

It's fairly common for people to request compressed output from Hadoop via configuration properties, for example:

```
python mr_your_job.py -D mapreduce.output.fileoutputformat.compress=true -D\
 mapreduce.output.fileoutputformat.compress.codec=org.apache.hadoop.io.compress.BZip2Codec ...
```

This works with `-r spark` too; the Spark runner knows how to recognize these properties and pass the codec specified to Spark when it writes output.

Spark won't split .gz files either

A common trick on Hadoop to ensure that segments of your data don't get split between mappers is to gzip each segment (since .gz is not a seekable compression format).

This works on Spark as well.

Controlling number of output files

By default, Spark will write one output file per partition. This may give more output files than you expect, since Hadoop and Spark are tuned differently.

The Spark runner knows how to emulate the Hadoop configuration property that sets number of reducers on Hadoop (e.g. `-D mapreduce.job.reduces=100`), which will control the number of output files (assuming your last step has a reducer).

However, this is a somewhat heavyweight solution; once Spark runs a step's reducer, mrjob has to forbid Spark from re-partitioning until the end of the step.

A lighter weight solution is `--max-output-files`, allows you to limit the number of output files by running `coalesce()` just before writing output. Running your job with `--max-output-files=100` would ensure it produces no more than 100 output files (but it could output less).

Running classic MRJobs on Spark on EMR

It's often faster to run classic MRJobs on Spark than Hadoop Streaming. It's also convenient to be able to run on EMR rather than setting up your own Spark cluster (or SSH'ing in).

Can you do both? Yes! Run the job with the Spark runner, but tell it to use **`mrjob spark-submit`** to launch Spark jobs on EMR.

It looks something like this:

```
python mr_your_job.py -r spark \
  --spark-submit-bin 'mrjob spark-submit -r emr' \
  --spark-master yarn --spark-tmp-dir s3://your-bucket/tmp/ input1 input2
```

Note that because the Spark runner itself doesn't know the job is going to run on EMR, you have to give it a couple of hints so that it knows it's running on YARN (**`--spark-master`**) and that it needs to use S3 as its temp space (**`--spark-tmp-dir`**).

1.7 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"
      }
    }
  }
}
```

1.7.1 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.

1.7.2 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()`*.

Strings and non-sequence data types (e.g. numbers) are treated as single-item lists.

For example,

```
runners:
  emr:
    setup: /run/some/command with args
```

is equivalent to:

```
runners:
  emr:
    setup:
    - /run/some/command with args
```

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. Combined with *`combine_dicts()`*.

JobConf Dicts

New in version 0.6.6: Like plain dicts except that non-string values are converted into a format that Java understands. For example, the boolean value `true` here:

```
jobconf:
  mapreduce.output.fileoutputformat.compress: true
```

gets passed through to Hadoop in Java format (`true`), not Python format (`True`).

Keys whose values are `None` are not passed to Hadoop at all.

Warning: Prior to version 0.6.6, you should use `"true"` and `"false"`, for boolean jobconf values in config files, not `true` and `false`.

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 function that handles this is *`combine_envs()`*.

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 `:`. In local mode, the function that combines config values is *`combine_local_envs()`*.

1.7.3 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_core_instances: 20
    core_instance_type: m1.xlarge
```

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

```
include: $HOME/.mrjob.base.conf
runners:
  emr:
    num_core_instances: 2
    core_instance_type: m1.small
```

`~/.mrjob.base.conf`:

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

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_secret_access_key: MEMIMOMADOOPBANANAFANAFOFADOOPHADOOP
    core_instance_type: m1.xlarge
    num_core_instances: 20
    region: us-west-1
```

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

Relative includes

Relative `include:` paths are relative to the real (after resolving symlinks) path of the including conf file.

For example, you could do this:

`~/.mrjob/base.conf`:

```
runners:
  ...
```

`~/.mrjob/default.conf`:

```
include: base.conf
```

You could then load your configs via a symlink `~/.mrjob.conf` to `~/.mrjob/default.conf` and `~/.mrjob/base.conf` would still be included (even though it's not in the same directory as the symlink).

1.7.4 Clearing configs

Sometimes, you just want to override a list-type config (e.g. `setup`) or a `*PATH` environment variable, rather than having mrjob cleverly concatenate it with previous configs.

You can do this in YAML config files by tagging the values you want to take precedence with the `!clear` tag.

For example:

`~/.mrjob.base.conf`

```
runners:
  emr:
    aws_access_key_id: HADOOPHADOOPBOBADOOP
    aws_secret_access_key: MEMIMOMADOOPBANANAFANAFOFADOOPHADOOP
    cmdenv:
      PATH: /this/nice/path
      PYTHONPATH: /here/be/serpents
```

```
      USER: dave
    setup:
    - /run/this/command
```

~/.mrjob.conf

```
include: ~/mrjob.base.conf
runners:
  emr:
    cmdenv:
      PATH: !clear /this/even/better/path/yay
      PYTHONPATH: !clear
    setup: !clear
    - /run/this/other/command
```

is equivalent to:

```
runners:
  emr:
    aws_access_key_id: HADOOPHADOOPBOBADOOP
    aws_secret_access_key: MEMIMOMADOOPBANANAFANAFOFADOOPHADOOP
    cmdenv:
      PATH: /this/even/better/path/yay
      USER: dave
    setup:
    - /run/this/other/command
```

If you specify multiple config files (e.g. `-c ~/mrjob.base.conf -c ~/mrjob.conf`), a `!clear` in a later file will override earlier files. `include:` is really just another way to prepend to the list of config files to load.

If you find it more readable, you may put the `!clear` tag *before* the key you want to clear. For example,

```
runners:
  emr:
    !clear setup:
    - /run/this/other/command
```

is equivalent to:

```
runners:
  emr:
    setup: !clear
    - /run/this/other/command
```

`!clear` tags in lists are ignored. You cannot currently clear an entire set of configs (e.g. `runners: emr: !clear ...` does not work).

1.8 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.

1.8.1 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 zip 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.

py_files (`--py-files`) [*path list*] **Default:** `[]`

List of `.egg` or `.zip` files to add to your job's `PYTHONPATH`.

This is based on a Spark feature, but it works just as well with streaming jobs.

Changed in version 0.6.7: Deprecated `--py-file` in favor of `--py-files`

upload_archives (`--archives`) [*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 name in the job's working directory we unpack into by appending `#nameinworkingdir` 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/`).

Changed in version 0.6.7: Deprecated `--archive` in favor of `--archives`

upload_dirs (`--dirs`) [*path list*] **Default:** `[]`

A list of directories to copy to the local directory of the mr_job script when it runs (mrjob does this by tarballing the directory and submitting the tarball to Hadoop as an archive).

You can set the name in the job's working directory of the directory we copy by appending `#nameinworkingdir` to the path; otherwise we just use its name.

This works with Spark on YARN only.

Changed in version 0.6.7: Deprecated `--dir` in favor of `--dirs`

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

Files to copy to the local directory of the mr_job script when it runs. You can set the name of the file in the job's working directory by appending `#nameinworkingdir` 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:

```
--files file_1.txt,file_2.sqlite
```

Changed in version 0.6.8: In Spark, can use `#nameinworkingdir` even when not on YARN.

Changed in version 0.6.7: Deprecated `--file` in favor of `--files`

1.8.2 Temp files and cleanup

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

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

- **`'ALL'`: delete logs and local and remote temp files; stop cluster** if on EMR and the job is not done when cleanup is run.
- `'CLUSTER'`: terminate EMR cluster if job not done when cleanup is run
- `'JOB'`: stop job if not done when cleanup runs (temporarily disabled)
- `'LOCAL_TMP'`: delete local temp files only
- `'LOGS'`: delete logs only
- `'NONE'`: delete nothing
- `'REMOTE_TMP'`: delete remote temp files only
- `'TMP'`: delete local and remote temp files, but not logs

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**.

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

Alternate local temp directory.

`--local-tmp-dir ''` tells mrjob to ignore the config file and use the default temp directory (`tempfile.gettempdir()`)

Changed in version 0.6.6: Added *–local-tmp-dir* switch.

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.

cat_output (`--cat-output`, `--no-cat-output`) [boolean] **Default:** output if output_dir is not set

Should we stream job output to STDOUT after completion?

Changed in version 0.6.3: used to be `--no-output`.

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

For a multi-step job, where to put output from job steps other than the last one. Each step's output will go into a numbered subdirectory of this one (`0000/`, `0001/`, etc.)

This option can be useful for debugging. By default, intermediate output goes into HDFS, which is fastest but not easily accessible on EMR or Dataproc.

This option currently does nothing on local and inline runners.

1.8.3 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
```

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

Name/path of alternate Python binary for wrapper scripts and mappers/reducers (e.g. `'python -v'`).

If you're on Python 3, this always defaults to `'python3'`.

If you're on Python 2, this defaults to `'python2.7'`.

If you're using PyPy, this defaults to `'pypy'` (not `'pypy2.7'`) or `'pypy3'` depending on your version.

This option also affects which Python binary is used for file locking in setup scripts. It's also used by *EMRJobRunner* to compile mrjob after bootstrapping it (see bootstrap_mrjob).

Changed in version 0.7.2: Defaults to `'python2.7'` (not `'python'`) on Python 2.

Changed in version 0.6.10: added `'pypy'` and `'pypy3'` as possible defaults

---

Note: mrjob does not auto-install PyPy for you on EMR; see *Installing PyPy* for how to do this

---

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* (e.g. `foo.tar.gz#foo/`).

You can indicate that a directory should be copied into the job's working directory by putting a `/` after *path* (e.g. `src-tree/#`). You may optionally put a `/` after *name* as well (e.g. `cd src-tree/#/subdir`).

This works for Spark as well (except on the `local[*]` master, where it doesn't make sense). The setup script is run before every executor, but only run before the driver in cluster mode.

Note: Uploading archives and directories (e.g. `src-tree/#`) to Spark's working directory still only works on YARN.

Changed in version 0.6.8: added full support for Spark

Changed in version 0.6.7: added support for Spark on YARN only

For more details of parsing, see *parse_setup_cmd()*.

sh_bin (`--sh-bin`) [*command*] **Default: `/bin/sh -ex`**

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

If you want to add an argument, use an absolute path (**`/bin/bash -x`**, not **`bash -x`**). Please do not pass multiple args to your shell binary (this plays poorly with Linux shebang syntax).

This is also used to wrap mappers, reducers, etc. that require piping one command into another (see e.g. *mapper_pre_filter()*).

Changed in version 0.6.8: Setting this to an empty value (`--sh-bin ''`) means to use the default (used to cause an error).

Changed in version 0.6.7: Used to be **`sh -ex`** on local and Hadoop runners

task_python_bin (`--task-python-bin`) [*command*] **Default:** same as python_bin

Name/path of alternate python binary to run the job (invoking it with `--mapper`, `--reducer`, `--spark`, etc.).

In most cases, you're better off setting python_bin, which this defaults to. This option exists mostly to support running tasks inside Docker while using a normal Python binary in setup wrapper scripts.

1.8.4 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
```

read_logs (`--read-logs`, `--no-read-logs`) [boolean] **Default:** `True`

New in version 0.6.5.

If set to false, don't list or read the contents of log files generated in the course of running your job.

The main impact of turning off *read_logs* is that if your job fails, mrjob won't spend any time or effort determining why it failed. On EMR, this effectively disables counter fetching as well.

This option does not stop the Hadoop and Dataproc runners from reading the output of the job driver (i.e. `hadoop jar ...`), so you will continue to get counters and high-level Java errors on these platforms.

1.8.5 Options ignored by the local and inline runners

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

- *hadoop_input_format*
- *hadoop_output_format*
- libjars
- *partitioner*

1.8.6 Options ignored by the inline runner

These options are ignored because the `inline` runner does not invoke the job as a subprocess:

- bootstrap_mrjob
- py_files
- python_bin
- read_logs
- setup

1.9 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.

1.9.1 Options specific to the local and inline runners

hadoop_version (`--hadoop-version`) [*string*] **Default:** `None`

Set the version of Hadoop to simulate (this currently only matters for jobconf).

If you don't set this, the `local` and `inline` runners will run in a version-agnostic mode, where anytime the runner sets a simulated jobconf variable, it'll use *every* possible name for it (e.g. `user.name` *and* `mapreduce.job.user.name`).

num_cores (`--num-cores`) [integer] **Default:** `None`

Maximum number of tasks to handle at one time. If not set, defaults to the number of CPUs on your system.

This also affects the number of input file splits the runner makes (the only impact in `inline` mode).

New in version 0.6.2.

1.9.2 Options available to local, hadoop, and emr runners

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

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

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

Changed in version 0.6.6: added the `-D` switch on the command line, to match Hadoop.

Changed in version 0.6.6: boolean `true` and `false` values in config files are passed correctly to Hadoop (see *JobConf dicts*)

1.9.3 Options available to hadoop and emr runners

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

Extra arguments to pass to hadoop streaming.

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

Path to a custom hadoop streaming jar.

On EMR, this can be either a local path or a URI (`s3://...`). If you want to use a jar at a path on the master node, use a `file://` URI.

On Hadoop, mrjob tries its best to find your hadoop streaming jar, searching these directories (recursively) for a `.jar` file with `hadoop` followed by `streaming` in its name:

- `$HADOOP_PREFIX`
- `$HADOOP_HOME`
- `$HADOOP_INSTALL`
- `$HADOOP_MAPRED_HOME`
- the parent of the directory containing the Hadoop binary (see hadoop_bin), unless it's one of `/`, `/usr` or `/usr/local`
- `$HADOOP_*_HOME` (in alphabetical order by environment variable name)
- `/home/hadoop/contrib`
- `/usr/lib/hadoop-mapreduce`

(The last two paths allow the Hadoop runner to work out-of-the box inside EMR.)

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

List of paths of JARs to be passed to Hadoop with the `-libjars` switch.

~ and environment variables within paths will be resolved based on the local environment.

Changed in version 0.6.7: Deprecated `--libjar` in favor of `--libjars`

Note: mrjob does not yet support *libjars* on Google Cloud Dataproc.

---

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

Alternate label for the job

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

Who is running this job (if different from the current user)

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.

spark_args (`--spark-args`) [*string list*] **Default:** `[]`

Extra arguments to pass to **`spark-submit`**.

> **Warning:** Don't use this to set `--master` or `--deploy-mode`. On the Hadoop runner, you can change these with spark_master and spark_deploy_mode. Other runners don't allow you to set these because they can only handle the defaults.

1.9.4 Options available to hadoop runner only

hadoop_bin (`--hadoop-bin`) [*command*] **Default:** (automatic)

Name/path of your **`hadoop`** binary (may include arguments).

mrjob tries its best to find **`hadoop`**, checking all of the following places for an executable file named `hadoop`:

- `$HADOOP_PREFIX/bin`
- `$HADOOP_HOME/bin`
- `$HADOOP_INSTALL/bin`
- `$HADOOP_INSTALL/hadoop/bin`
- `$PATH`
- `$HADOOP_*_HOME/bin` (in alphabetical order by environment variable name)

If all else fails, we just use `hadoop` and hope for the best.

Changed in version 0.6.8: Setting this to an empty value (`--hadoop-bin ''`) means to search for the Hadoop binary (used to effectively disable use of the **`hadoop`** command).

hadoop_log_dirs (`--hadoop-log-dir`) [*path list*] **Default:** (automatic)

Where to look for Hadoop logs (to find counters and probable cause of job failure). These can be (local) paths or URIs (`hdfs:///...`).

If this is *not* set, mrjob will try its best to find the logs, searching in:

- `$HADOOP_LOG_DIR`
- `$YARN_LOG_DIR` (on YARN only)
- `hdfs:///tmp/hadoop-yarn/staging` (on YARN only)
- `<job output dir>/_logs` (usually this is on HDFS)
- `$HADOOP_PREFIX/logs`
- `$HADOOP_HOME/logs`
- `$HADOOP_INSTALL/logs`
- `$HADOOP_MAPRED_HOME/logs`

- `<dir containing hadoop bin>/logs` (see hadoop_bin), unless the hadoop binary is in `/bin`, `/usr/bin`, or `/usr/local/bin`
- `$HADOOP_*_HOME/logs` (in alphabetical order by environment variable name)
- `/var/log/hadoop-yarn` (on YARN only)
- `/mnt/var/log/hadoop-yarn` (on YARN only)
- `/var/log/hadoop`
- `/mnt/var/log/hadoop`

hadoop_tmp_dir (`--hadoop-tmp-dir`) [*path*] **Default:** `tmp/mrjob`

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.

spark_deploy_mode (`--spark-deploy-mode`) [*string*] **Default:** `'client'`

Deploy mode (`client` or `cluster`) to pass to the `--deploy-mode` argument of **`spark-submit`**.

New in version 0.6.6.

spark_master (`--spark-master`) [*string*] **Default:** `'yarn'`

Name or URL to pass to the `--master` argument of **`spark-submit`** (e.g. `spark://host:port`, `yarn`).

Note that archives (see upload_archives) only work when this is set to `yarn`.

spark_submit_bin (`--spark-submit-bin`) [*command*] **Default:** (automatic)

Name/path of your **`spark-submit`** binary (may include arguments).

mrjob tries its best to find **`spark-submit`**, checking all of the following places for an executable file named `spark-submit`:

- `$SPARK_HOME/bin`
- `$PATH`
- your `pyspark` installation's `bin/` directory
- `/usr/lib/spark/bin`
- `/usr/local/spark/bin`
- `/usr/local/lib/spark/bin`

If all else fails, we just use `spark-submit` and hope for the best.

Changed in version 0.6.8: Searches for **`spark-submit`** in `pyspark` installation.

1.10 Spark runner options

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

In addition, the Spark runner has the following options in common with other runners:

- aws_access_key_id
- aws_secret_access_key
- aws_session_token
- cloud_fs_sync_secs

- cloud_part_size_mb
- gcs_region
- project_id
- s3_endpoint
- s3_region

Options unique to the Spark runner:

emulate_map_input_file (`--emulate-map-input-file`, `--no-emulate-map-input-file`) [boolean] **Default:** `False`

Imitate Hadoop by setting `$mapreduce_map_input_file` to the path of the input file for the current partition. This helps support jobs that rely on *`jobconf_from_env('mapreduce.map.input.file')`*.

This feature only applies to the mapper of the job's first step, and is ignored by jobs that set *`HADOOP_INPUT_FORMAT`*.

New in version 0.6.9.

gcs_region (`--gcs-region`) [*string*] **Default:** `None`

The region to use when creating a temporary bucket on Google Cloud Storage.

Similar in meaning to region, but only used to configure GCS (not S3)

s3_region (`--s3-region`) [*string*] **Default:** `None`

The region to use when creating a temporary bucket on S3.

Similar in meaning to region, but only used to configure S3 (not GCS)

skip_internal_protocol (`--skip-internal-protocol`, `--no-skip-internal-protocol`) [boolean] **Default:** `False`

Don't emulate the job's internal protocol (used for communicating between job steps and tasks in the same step), instead relying on Spark to encode and decode data structures.

This should work for most but not all jobs, and make them run at least somewhat faster. Some things to keep in mind:

- data will no longer be "normalized" by being converted to and from string representation. For example, running a tuple through *`JSONProtocol`* (the default) implicitly converts it to a list because there are no tuples in JSON. With internal protocols skipped, it would remain a tuple.
- if your job uses *`SORT_VALUES`*, keep in mind that your values will need to be comparable as Spark will be comparing them directly, rather than comparing their internal-protocol-encoded representation. This may also affect sorting order.

New in version 0.6.10.

spark_tmp_dir (`--spark-tmp-dir`) [*string*] **Default:** (automatic)

A place to put files where they are visible to Spark executors, similar to cloud_tmp_dir.

If running locally, defaults to a directory inside local_tmp_dir, and if running on a cluster, to `tmp/mrjob` on HDFS.

1.11 Configuration quick reference

1.11.1 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: python3.6  # only used in local runner
    emr:
        python_bin: python3  # only used in Elastic MapReduce runner
```

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

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

There are some options that it makes no sense to set in the config file.

These options can be set via command-line switches:

Config	Command line	Default	Type
cat_output	–cat-output, –no-cat-output	output if output_dir is not set	boolean
conf_paths	-c, –conf-path, –no-conf	see *find_mrjob_conf()*	*path list*
output_dir	–output-dir	(automatic)	*string*
step_output_dir	–step-output-dir	(automatic)	*string*

These options can be set by overriding attributes or methods in your job class:

Option	Attribute	Method	Default
hadoop_input_format	*HADOOP_INPUT_FORMAT*	*hadoop_input_format()*	None
hadoop_output_format	*HADOOP_OUTPUT_FORMAT*	*hadoop_output_format()*	None
partitioner	*PARTITIONER*	*partitioner()*	None

These options can be set by overriding your job's *configure_args()* to call the appropriate method:

Option	Method	Default
extra_args	*add_passthru_arg()*	[]

All of the above can be passed as keyword arguments to *MRJobRunner.__init__()* (this is what makes them runner options), but you usually don't want to instantiate runners directly.

1.11.3 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
bootstrap_mrjob	–bootstrap-mrjob, –no-bootstrap-mrjob	`True`	boolean
check_input_paths	–check-input-paths, –no-check-input-paths	`True`	boolean
cleanup	–cleanup	`'ALL'`	*string*
cleanup_on_failure	–cleanup-on-failure	`'NONE'`	*string*
cmdenv	–cmdenv	`{}`	*environment variable dict*
hadoop_extra_args	–hadoop-args	`[]`	*string list*
hadoop_streaming_jar	–hadoop-streaming-jar	(automatic)	*string*
jobconf	-D, –jobconf	`{}`	*jobconf dict*
label	–label	script's module name, or `no_script`	*string*
libjars	–libjars	`[]`	*string list*
local_tmp_dir	–local-tmp-dir	value of `tempfile.gettempdir()`	*path*
owner	–owner	`getpass.getuser()`, or `no_user` if that fails	*string*
py_files	–py-files	`[]`	*path list*
python_bin	–python-bin	(automatic)	*command*
read_logs	–read-logs, –no-read-logs	`True`	boolean
setup	–setup	`[]`	*string list*
sh_bin	–sh-bin	**`/bin/sh -ex`**	*command*
spark_args	–spark-args	`[]`	*string list*
task_python_bin	–task-python-bin	same as python_bin	*command*
upload_archives	–archives	`[]`	*path list*
upload_dirs	–dirs	`[]`	*path list*
upload_files	–files	`[]`	*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*, *upload_archives*, and *upload_files*.

1.11.4 Additional options for `DataprocJobRunner`

Config	Command line	Default	Type
cluster_properties	–cluster-property	`None`	
core_instance_config	–core-instance-config	`None`	
gcloud_bin	–gcloud-bin	`'gcloud'`	*command*
master_instance_config	–master-instance-config	`None`	
network	–network	`None`	*string*
project_id	–project-id	read from credentials config file	*string*
service_account	–service-account	`None`	
service_account_scopes	–service-account-scopes	(automatic)	
task_instance_config	–task-instance-config	`None`	

1.11.5 Additional options for `EMRJobRunner`

Config	Command line	Default
add_steps_in_batch	–add-steps-in-batch, –no-add-steps-in-batch	`True` for AMIs before 5.28.0, `False` otherwise
additional_emr_info	–additional-emr-info	`None`
applications	–application, –applications	`[]`
aws_access_key_id		`None`
aws_secret_access_key	–aws-secret-access-key	`None`
aws_session_token		`None`
bootstrap_actions	–bootstrap-actions	`[]`
bootstrap_spark	–bootstrap-spark, –no-bootstrap-spark	(automatic)
cloud_log_dir	–cloud-log-dir	append `logs` to cloud_tmp_dir
core_instance_bid_price	–core-instance-bid-price	`None`
docker_client_config	–docker-client-config	`None`
docker_image	–docker-image, –no-docker	`None`
docker_mounts	–docker-mount	`[]`
ebs_root_volume_gb	–ebs-root-volume-gb	`None`
ec2_endpoint	–ec2-endpoint	(automatic)
ec2_key_pair	–ec2-key-pair	`None`
ec2_key_pair_file	–ec2-key-pair-file	`None`
emr_action_on_failure	–emr-action-on-failure	(automatic)
emr_configurations	–emr-configuration	`[]`
emr_endpoint	–emr-endpoint	infer from region
enable_emr_debugging	–enable-emr-debugging	`False`
hadoop_streaming_jar_on_emr	–hadoop-streaming-jar-on-emr	AWS default
iam_endpoint	–iam-endpoint	(automatic)
iam_instance_profile	–iam-instance-profile	(automatic)
iam_service_role	–iam-service-role	(automatic)
instance_fleets	–instance-fleet	`None`
instance_groups	–instance-groups	`None`
master_instance_bid_price	–master-instance-bid-price	`None`
max_clusters_in_pool	–max-clusters-in-pool	0 (disabled)
max_concurrent_steps	–max-concurrent-steps	1
min_available_mb	–min-available-mb	0 (disabled)
min_available_virtual_cores	–min-available-virtual-cores	0 (disabled)
pool_clusters	–pool-clusters	`True`
pool_jitter_seconds	–pool-jitter-seconds	60
pool_name	–pool-name	`'default'`
pool_timeout_minutes	–pool-timeout-minutes	0 (disabled)
pool_wait_minutes	–pool-wait-minutes	0
release_label	–release-label	`None`
s3_endpoint	–s3-endpoint	(automatic)
ssh_add_bin	–ssh-add-bin	`'ssh-add'`
ssh_bin	–ssh-bin	`'ssh'`
tags	–tag	`{}`
task_instance_bid_price	–task-instance-bid-price	`None`

1.11.6 Additional options for `HadoopJobRunner`

Config	Command line	Default	Type
hadoop_bin	–hadoop-bin	(automatic)	*command*
hadoop_log_dirs	–hadoop-log-dir	(automatic)	*path list*
hadoop_tmp_dir	–hadoop-tmp-dir	`tmp/mrjob`	*path*
spark_deploy_mode	–spark-deploy-mode	`'client'`	*string*
spark_master	–spark-master	`'yarn'`	*string*
spark_submit_bin	–spark-submit-bin	(automatic)	*command*

1.12 Cloud runner options

These options are generally available whenever you run your job on a Hadoop cloud service (AWS Elastic MapReduce or Google Cloud Dataproc).

All options from Options available to all runners and Hadoop-related options are also available on cloud services.

1.12.1 Google credentials

See *Getting started with Google Cloud* for specific instructions about setting these options.

1.12.2 Choosing/creating a cluster to join

cluster_id (`--cluster-id`) [*string*] **Default:** automatically create a cluster and use it

The ID of a persistent cluster to run jobs in (on Dataproc, this is the same thing as "cluster name").

It's fine for other jobs to be using the cluster; we give our job's steps a unique ID.

1.12.3 Job placement

region (`--region`) [*string*] **Default:** `'us-west-2'` on EMR, `'us-west1'` on Dataproc

Geographic region to run jobs in (e.g. `us-central-1`).

If mrjob create a temporary bucket, it will be created in this region as well.

If you set region, you do not need to set zone; a zone will be chosen for you automatically.

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

Optional subnet(s) to launch your job in.

On Amazon EMR, this is the ID of a VPC subnet to launch cluster in (e.g. `'subnet-12345678'`). This can also be a list of possible subnets if you are using instance_fleets.

On Google Cloud Dataproc, this is the name of a subnetwork (e.g. `'default'`). Specifying *subnet* rather than network will ensure that your cluster only has access to one specific geographic region, rather than the entire VPC.

Changed in version 0.6.8: `--subnet ''` un-sets the subnet on EMR (used to be ignored).

Changed in version 0.6.3: Works on Google Cloud Dataproc as well as AWS Elastic MapReduce.

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

Zone within a specific geographic region to run your job in.

If you set this, you do not neet to set region.

1.12.4 Number and type of instances

instance_type (`--instance-type`) [*string*] **Default:** `m4.large` or `m5.xlarge` on EMR, `n1-standard-1` on Dataproc

Type of instance that runs your Hadoop tasks.

Once you've tested a job and want to run it at scale, it's usually a good idea to use instances larger than the default. For EMR, see Amazon EC2 Instance Types and for Dataproc, see Machine Types.

Note: Many EC2 instance types can only run in a VPC (see subnet).

If you're running multiple nodes (see num_core_instances), this option *doesn't* apply to the master node because it's just coordinating tasks, not running them. Use master_instance_type instead.

Changed in version 0.6.11: Default on EMR is `m5.xlarge` on AMI version 5.13.0 and later, `m4.large` on earlier versions

Changed in version 0.6.10: Default on EMR changed to `m5.xlarge`

Changed in version 0.6.6: Default on EMR changed to `m4.large`. Was previously *m1.large'* if running Spark, `m1.small` if running on the (deprecated) 2.x AMIs, and `m1.medium` otherwise

core_instance_type (`--core-instance-type`) [*string*] **Default:** value of instance_type

like instance_type, but only for the core (worker) Hadoop nodes; these nodes run tasks and host HDFS. Usually you just want to use instance_type.

num_core_instances (`--num-core-instances`) [integer] **Default:** 0 on EMR, 2 on Dataproc

Number of core (worker) instances to start up. These run your job and host HDFS. This is in addition to the single master instance.

On Google Cloud Dataproc, this must be at least 2.

master_instance_type (`--master-instance-type`) [*string*] **Default:** (automatic)

like instance_type, but only for the master Hadoop node. This node hosts the task tracker/resource manager and HDFS, and runs tasks if there are no other nodes.

If you're running a single node (no num_core_instances or num_task_instances), this will default to the value of instance_type.

Otherwise, on Dataproc, defaults to `n1-standard-1`, and on EMR defaults to `m1.medium` (exception: `m1.small` on the deprecated 2.x AMIs), which is usually adequate for all but the largest jobs.

task_instance_type (`--task-instance-type`) [*string*] **Default:** value of core_instance_type

like instance_type, but only for the task (secondary worker) Hadoop nodes; these nodes run tasks but do not host HDFS. Usually you just want to use instance_type.

num_task_instances (`--num-task-instances`) [integer] **Default:** 0

Number of task (secondary worker) instances to start up. These run your job but do not host HDFS.

You must have at least one core instance (see num_core_instances) to run task instances; otherwise there's nowhere to host HDFS.

1.12.5 Cluster software configuration

image_id (`--image-id`) [*string*] **Default:** None

ID of a custom machine image.

On EMR, this is complimentary with image_version; you can install packages and libraries on your custom AMI, but it's up to EMR to install Hadoop, create the `hadoop` user, etc. image_version may not be less than 5.7.0.

You can use *`describe_base_emr_images()`* to identify Amazon Linux images that are compatible with EMR.

For more details about how to create a custom AMI that works with EMR, see Best Practices and Considerations.

Note: This is not yet implemented in the Dataproc runner.

New in version 0.6.5.

image_version (`--image-version`) [*string*] **Default:** `'6.0.0'` on EMR, `'1.3'` on Dataproc

Machine image version to use. This controls which Hadoop version(s) are available and which version of Python is installed, among other things.

See the AMI version docs (EMR) or the Dataproc version docs for more details.

You can use this instead of release_label on EMR, even for 4.x+ AMIs; mrjob will just prepend `emr-` to form the release label.

Changed in version 0.6.12: Default on Dataproc changed from `1.0` to `1.3`

Changed in version 0.6.11: Default on EMR is now `5.27.0`

Changed in version 0.6.5: Default on EMR is now `5.16.0` (was `5.8.0`)

Warning: The 2.x series of AMIs is deprecated by Amazon and not recommended.

Warning: The 1.x series of AMIs is no longer supported because they use Python 2.5.

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

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

This option is complex and powerful. On EMR, 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_python (`--bootstrap-python`, `--no-bootstrap-python`) [boolean] **Default:** `True` on Dataproc, as needed on EMR.

Attempt to install a compatible (major) version of Python at bootstrap time, including header files and **`pip`** (see *Installing Python packages with pip*).

The only reason to set this to `False` is if you want to customize Python/pip installation using bootstrap.

extra_cluster_params (`--extra-cluster-param`) [*dict*] **Default:** `{}`

An escape hatch that allows you to pass extra parameters to the EMR/Dataproc API at cluster create time, to access API features that mrjob does not yet support.

For EMR, see the API documentation for RunJobFlow for the full list of options.

Option names are strings, and values are data structures. On the command line, `--extra-cluster-param name=value`:

```
--extra-cluster-param SupportedProducts='["mapr-m3"]'
--extra-cluster-param AutoScalingRole=HankPym
```

value can be either a JSON or a string (unless it starts with `{`, `[`, or `"`, so that we don't convert malformed JSON to strings). Parameters can be suppressed by setting them to `null`:

```
--extra-cluster-param LogUri=null
```

This also works with Google dataproc:

```
--extra-cluster-param labels='{"name": "wrench"}'
```

In the config file, *extra_cluster_param* is a dict:

```
runners:
  emr:
    extra_cluster_params:
      AutoScalingRole: HankPym
      LogUri: null  # !clear works too
      SupportedProducts:
      - mapr-m3
```

Changed in version 0.7.2: Dictionaries will be recursively merged into existing parameters. For example:

```
runners:
  emr:
    extra_cluster_params:
      Instances:
        EmrManagedMasterSecurityGroup: sg-foo
```

Changed in version 0.6.8: You may use a *name* with dots in it to set (or unset) nested properties. For example: `--extra-cluster-param Instances.EmrManagedMasterSecurityGroup=sg-foo`.

1.12.6 Monitoring your job

check_cluster_every (`--check-cluster-every`) [float] **Default:** 10 seconds on Dataproc, 30 seconds on EMR

How often to check on the status of your job, in seconds.

Changed in version 0.6.5: When the EMR client encounters a transient error, it will wait at least this many seconds before trying again.

ssh_tunnel (`--ssh-tunnel`, `--no-ssh-tunnel`) [boolean] **Default:** `False`

If True, create an ssh tunnel to the job tracker/resource manager and listen on a randomly chosen port.

On EMR, this requires you to set ec2_key_pair and ec2_key_pair_file. See *Configuring SSH credentials* for detailed instructions.

On Dataproc, you don't need to set a key, but you do need to have the **gcloud** utility installed and set up (make sure you ran **gcloud auth login** and **gcloud config set project <project_id>**). See *Installing gcloud, gsutil, and other utilities*.

Changed in version 0.6.3: Enabled on Google Cloud Dataproc

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.

Does nothing unless ssh_tunnel is set.

ssh_bind_ports (--ssh-bind-ports) [list of integers] **Default:** `range(40001, 40841)`

A list of ports that are safe to listen on.

The main reason to set this is if your firewall blocks the default range of ports, or if you want to pick a single port for consistency.

On the command line, this looks like `--ssh-bind-ports 2000[:2001][,2003,2005:2008,etc]`, where commas separate ranges and colons separate range endpoints.

1.12.7 Cloud Filesystem

cloud_fs_sync_secs (--cloud-fs-sync-secs) [float] **Default:** 5.0

How long to wait for cloud filesystem (e.g. S3, GCS) to reach eventual consistency? This is typically less than a second, but the default is 5 seconds to be safe.

cloud_part_size_mb (--cloud-part-size-mb) [integer] **Default:** 100

Upload files to cloud filesystem in parts no bigger than this many megabytes (technically, mebibytes). Default is 100 MiB.

Set to 0 to disable multipart uploading entirely.

Currently, Amazon requires parts to be between 5 MiB and 5 GiB. mrjob does not enforce these limits.

Changed in version 0.6.3: Enabled on Google Cloud Storage. This used to be called *cloud_upload_part_size*.

cloud_tmp_dir (--cloud-tmp-dir) [*string*] **Default:** (automatic)

Directory on your cloud filesystem to use as temp space (e.g. `s3://yourbucket/tmp/`, `gs://yourbucket/tmp/`).

By default, mrjob looks for a bucket belong to you whose name starts with `mrjob-` and which matches region. If it can't find one, it creates one with a random name. This option is then set to *tmp/* in this bucket (e.g. `s3://mrjob-01234567890abcdef/tmp/`).

1.12.8 Auto-termination

max_mins_idle (--max-mins-idle) [float] **Default:** 10.0

Automatically terminate your cluster after it has been idle at least this many minutes. You cannot turn this off (clusters left idle rack up billing charges).

If your cluster is only running a single job, mrjob will attempt to terminate it as soon as your job finishes. This acts as an additional safeguard, as well as affecting *Cluster Pooling* on EMR.

Changed in version 0.6.5: EMR's idle termination script is more robust against **sudo shutdown -h now** being ignored, and logs the script's stdout and stderr to `/var/log/bootstrap-actions/mrjob-idle-termination.log`.

Changed in version 0.6.3: Uses Dataproc's built-in cluster termination feature rather than a custom script. The API will not allow you to set an idle time of less than 10 minutes.

Changed in version 0.6.2: No matter how small a value you set this to, there is a grace period of 10 minutes between when the idle termination daemon launches and when it may first terminate the cluster, to allow Hadoop to accept your first job.

1.13 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.

Note: Setup scripts don't work with Spark; try py_files instead.

1.13.1 Uploading your source tree

Note: If you're using mrjob 0.6.4 or later, check out *Using other python modules and packages* first.

mrjob can automatically tarball your source directory and include it in your job's working directory. We can use setup scripts to upload the directory and then add it to `PYTHONPATH`.

Run your job with:

```
--setup 'export PYTHONPATH=$PYTHONPATH:your-src-code/#'
```

The `/` before the `#` tells mrjob that `your-src-code` is a directory. You may optionally include a `/` after the `#` as well (e.g. `export PYTHONPATH=$PYTHONPATH:your-source-code/#/your-lib`).

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

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

1.13.2 Uploading your source tree as an archive

Prior to mrjob 0.5.8, you had to archive directories yourself before uploading them.

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

Then, run your job with:

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

The / after the # (without one before it) is what tells mrjob that `your-src-code.tar.gz` is an archive that Hadoop should unpack.

To do the same thing in `mrjob.conf`:

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

1.13.3 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.

1.13.4 Making data files available to your job

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

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.

1.13.5 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
```

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
```

With Hadoop 1, you'd want to use `$mapred_job_id` instead of `$mapreduce_job_id`.

1.13.6 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#'
```

1.14 Hadoop Cookbook

1.14.1 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. For example, to set the timeout to one hour:

```
runners:
  hadoop: # also works for emr runner
    jobconf:
      mapreduce.task.timeout: 3600000
```

Note: If you're using Hadoop 1, which uses `mapred.task.timeout`, don't worry: this example still works because mrjob auto-converts your jobconf options between Hadoop versions.

1.14.2 Writing compressed output

To save space, you can have Hadoop automatically save your job's output as compressed files. Here's how you tell it to bzip them:

```
runners:
  hadoop: # also works for emr runner
    jobconf:
      # "true" must be a string argument, not a boolean! (Issue #323)
      mapreduce.output.fileoutputformat.compress: "true"
      mapreduce.output.fileoutputformat.compress.codec: org.apache.hadoop.io.compress.BZip2Codec
```

Note: You could also gzip your files with `org.apache.hadoop.io.compress.GzipCodec`. Usually bzip is a better option, as `.bz2` files are splittable, and `.gz` files are not. For example, if you use `.gz` files as input, Hadoop has no choice but to create one mapper per `.gz` file.

1.15 Testing jobs

mrjob can run jobs without the help of Hadoop. This isn't very efficient, but it's a great way to test a job before submitting it to a cluster.

1.15.1 Inline runner

The `inline` runner (*InlineMRJobRunner*) is the default runner for mrjob (it's what's used when you run **`python mr_your_job.py <input>`** without any `-r` option). It runs your job in a single process so that you get faster feedback and simpler tracebacks.

Multiple splits

The `inline` runner doesn't run mappers or reducers concurrently, but it does run at least two mappers and two reducers for each step. This can help catch bad assumptions about the MapReduce programming model.

For example, say we wanted to write a simple script that counted the number of lines of input:

```
from mrjob.job import MRJob

 class MRCountLinesWrong(MRJob):

     def mapper_init(self):
         self.num_lines = 0

     def mapper(self, _, line):
         self.num_lines += 1

     def mapper_final(self):
         yield None, self.num_lines


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

Looks good, but if we run it, we get more than one line count:

```
$ python -m mrjob.examples.mr_count_lines_wrong README.rst 2> /dev/null
null        77
null        60
```

Aha! Because there can be more than one mapper! It's fine to use *mapper_final()* like this, but we need to reduce on a single key:

```
from mrjob.job import MRJob

class MRCountLinesRight(MRJob):

    def mapper_init(self):
```

```
        self.num_lines = 0

    def mapper(self, _, line):
        self.num_lines += 1

    def mapper_final(self):
        yield None, self.num_lines

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


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

```
$ python -m mrjob.examples.mr_count_lines_right README.rst 2> /dev/null
null        137
```

Thanks, inline runner!

Isolated working directories

Just like Hadoop, the inline runner runs each mapper and reducer in its own (temporary) working directory. It *does* add the original working directory to `$PYTHONPATH` so it can still access your local source tree.

Simulating jobconf

The inline runner simulates jobconf variables/properties set by Hadoop (and their Hadoop 1 equivalents):

- `mapreduce.job.cache.archives` (`mapred.cache.archives`)
- `mapreduce.job.cache.files` (`mapred.cache.files`)
- `mapreduce.job.cache.local.archives` (`mapred.cache.localArchives`)
- `mapreduce.job.cache.local.files` (`mapred.cache.localFiles`)
- `mapreduce.job.id` (`mapred.job.id`)
- `mapreduce.job.local.dir` (`job.local.dir`)
- `mapreduce.map.input.file` (`map.input.file`)
- `mapreduce.map.input.length` (`map.input.length`)
- `mapreduce.map.input.start` (`map.input.start`)
- `mapreduce.task.attempt.id` (`mapred.task.id`)
- `mapreduce.task.id` (`mapred.tip.id`)
- `mapreduce.task.ismap` (`mapred.task.is.map`)
- `mapreduce.task.output.dir` (`mapred.work.output.dir`)
- `mapreduce.task.partition` (`mapred.task.partition`)

You can use *`jobconf_from_env()`* to read these from your job's environment. For example:

```
from mrjob.compat import jobconf_from_env
from mrjob.job import MRJob

class MRCountLinesByFile(MRJob):

    def mapper(self, _, line):
        yield jobconf_from_env('mapreduce.map.input.file'), 1

    def reducer(self, path, ones):
        yield path, sum(ones)


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

```
$ python -m mrjob.examples.mr_count_lines_by_file README.rst CHANGES.txt 2> /dev/null
"CHANGES.txt"       564
"README.rst"        137
```

If you only want to simulate jobconf variables from a single version of Hadoop (for more stringent testing), you can set hadoop_version.

Setting number of mappers and reducers

Want more or less splits? You can tell the inline runner the same way you'd tell hadoop, with the `mapreduce.job.maps` and `mapreduces.job.reduces` jobconf options:

```
$ python -m mrjob.examples.mr_count_lines_wrong --jobconf mapreduce.job.maps=5 README.rst 2> /dev/nul
null        24
null        33
null        38
null        30
null        12
```

1.15.2 Local runnner

The `local` runner (*LocalMRJobRunner*; run using `-r local`) supports the above features, but, unlike the `inline` runner, it uses subprocesses.

This means it can be used to test options that don't make sense in a single-process context, including:

- python_bin
- setup

The local runner *does* run multiple subprocesses concurrently, but it's not really meant as a replacement for Hadoop; it's just for testing!

1.15.3 Anatomy of a test case

So, you've gotten a job working. Great! Here's how you write a regression test so that future developers won't break it.

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:

```
from unittest import TestCase

class MRInitTestCase(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 `BytesIO` object, populate it with data, initialize your job to read from stdin, and enable the sandbox with your `BytesIO` as stdin.

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):

```
from io import BytesIO

    def test_init_funcs(self):
        num_inputs = 2
        stdin = BytesIO(b'x\n' * num_inputs)
        mr_job = MRInitJob(['--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 per-

formed regardless of the success or failure of the job.

Run the job with *run()*. The job's output is available as a generator through *cat_output()* and can be parsed with the job's output protocol using *parse_output()*:

```
results = []
with mr_job.make_runner() as runner:
    runner.run()
    for key, value in mrjob.parse_output(runner.cat_output()):
        results.append(value)

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

Warning: Do not let your tests depend on the input lines being processed in a certain order. Both mrjob and Hadoop divide input non-deterministically.

1.16 Cloud Dataproc

1.16.1 Dataproc Quickstart

Getting started with Google Cloud

Using mrjob with Google Cloud Dataproc is as simple creating an account, enabling Google Cloud Dataproc, and creating credentials.

Creating an account

- Go to cloud.google.com.
- Click the circle in the upper right, and select your Google account (if you don't have one sign up here. *If you have multiple Google accounts, sign out first, and then sign into the account you want to use.*
- Click **Try it Free** in the upper right
- Enter your name and payment information
- Wait a few minutes while your first project is created

Enabling Google Cloud Dataproc

- Go here (or search for "dataproc" under **APIs & Services > Library** in the upper left-hand menu)
- Click **Enable**

Configuring Google Cloud credentials

- Go here (or pick **APIs & Services > Credentials** in the upper left-hand menu)
- Pick **Create credentials > Service account key**
- Select **Compute engine default service account**

- Click **Create** to download a JSON file.

Then you should either install and set up the optional **gcloud** utility (see below) or point **$GOOGLE_APPLICATION_CREDENTIALS** at the file you downloaded (`export GOOGLE_APPLICATION_CREDENTIALS="/path/to/Your Credentials.json"`).

Installing gcloud, gsutil, and other utilities

mrjob does not require you to install the **gcloud** command in order to run jobs on Google Cloud Dataproc, unless you want to set up an SSH tunnel to the Hadoop resource manager (see ssh_tunnel).

The **gcloud** command can be very useful for monitoring your job. The **gsutil** utility, packaged with it, is very helpful for dealing with Google Storage, the cloud filesystem that Google Cloud Dataproc uses.

To install **gcloud** and **gsutil**:

- Follow these three steps to install the utilities
- Log in with your Google credentials (these will launch a browser): * **gcloud auth login** * **gcloud auth application-default init**

On some versions of gcloud, you may have to manually configure project ID for ssh_tunnel to work.

- run **gcloud projects list** to get your project ID
- **gcloud config set project <project_id>**

It's also helpful to set **gcloud**'s region and zone to match mrjob's defaults:

- **gcloud config set compute/region us-west1**
- **gcloud config set compute/zone us-west1-a**
- **gcloud config set dataproc/region us-west1**

Running a Dataproc Job

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

- The job and related files are uploaded to GCS before being run
- The job is run on Dataproc (of course)
- Output is written to GCS before mrjob streams it to stdout locally
- The Hadoop version is specified by the Dataproc 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 dataproc 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 GCS, use `--output-dir`:

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

Choosing Type and Number of GCE Instances

When you create a cluster on Dataproc, you'll have the option of specifying a number and type of GCE 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 Machine 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.
- **Worker**: You may have one or more worker instances. These run tasks and host HDFS.
- **Preemptible Worker**: You may have zero or more of these. These run tasks, but do *not* host HDFS. This is mostly useful because your cluster can lose task instances without killing your job (see Preemptible VMs).

By default, `mrjob` runs a single `n1-standard-1`, 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:

- Google Cloud bills you a 10-minute minimum even if your cluster only lasts for a few minutes (this is an artifact of the Google Cloud 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 10 minutes.
- 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`**.) 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 `n1-highcpu-8` costs about 6 times as much as an `n1-standard-1`, but it has about 8 times as much processing power (and more memory).

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

```
--instance-type n1-highcpu-8 --num-core-instances 4
```

or in *`mrjob.conf`*, like this:

```
runners:
  dataproc:
    instance_type: n1-highcpu-8
    num_core_instances: 4
```

In most cases, your master instance type doesn't need to be larger than `n1-standard-1` to schedule tasks. *instance_type* only applies to instances that actually run tasks. (In this example, there are 1 `n1-standard-1` master instance, and 4 `n1-highcpu-8` worker instances.) You *will* need a larger master instance if you have a very large number of input files; in this case, use the master_instance_type option.

If you want to run preemptible instances, use the task_instance_type and num_task_instances options.

1.16.2 Dataproc runner options

All options from Options available to all runners, Hadoop-related options, and Cloud runner options are available when running jobs on Google Cloud Dataproc.

Google credentials

Basic credentials are not set in the config file; see *Getting started with Google Cloud* for details.

project_id (`--project-id`) [*string*] **Default:** read from credentials config file

The ID of the Google Cloud Project to run under.

Changed in version 0.6.2: This used to be called *gcp_project*

service_account (`--service-account`) [] **Default:** `None`

Optional service account to use when creating a cluster. For more information see Service Accounts.

New in version 0.6.3.

service_account_scopes (`--service-account-scopes`) [] **Default:** (automatic)

Optional service account scopes to pass to the API when creating a cluster.

Generally it's suggested that you instead create a service_account with the scopes you want.

New in version 0.6.3.

Job placement

See also subnet, region, zone

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

Name or URI of network to launch cluster in. Incompatible with with subnet.

New in version 0.6.3.

Cluster configuration

cluster_properties (`--cluster-property`) [] **Default:** `None`

A dictionary of properties to set in the cluster's config files (e.g. `mapred-site.xml`). For details, see Cluster properties.

core_instance_config (`--core-instance-config`) [] **Default:** `None`

A dictionary of additional parameters to pass as `config.worker_config` when creating the cluster. Follows the format of InstanceGroupConfig except that it uses *snake_case* instead of *camel_case*.

For example, to specify 100GB of disk space on core instances, add this to your config file:

```
runners:
  dataproc:
    core_instance_config:
      disk_config:
        boot_disk_size_gb: 100
```

To set this option on the command line, pass in JSON:

```
--core-instance-config '{"disk_config": {"boot_disk_size_gb": 100}}'
```

This option *can* be used to set number of core instances (`num_instances`) or instance type (`machine_type_uri`), but usually you'll want to use num_core_instances and core_instance_type along with this option.

New in version 0.6.3.

master_instance_config (--master-instance-config) [] **Default:** `None`

A dictionary of additional parameters to pass as `config.master_config` when creating the cluster. See core_instance_config for more details.

New in version 0.6.3.

task_instance_config (--task-instance-config) [] **Default:** `None`

A dictionary of additional parameters to pass as `config.secondary_worker_config` when creating the cluster. See task_instance_config for more details.

To make task instances preemptible, add this to your config file:

```
runners:
  dataproc:
    task_instance_config:
      is_preemptible: true
```

Note that this config won't be applied unless you specify at least one task instance (either through num_task_instances or by passing `num_instances` to this option).

New in version 0.6.3.

1.16.3 Other rarely used options

gcloud_bin (--gcloud-bin) [*command*] **Default:** `'gcloud'`

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

Used only as a way to create an SSH tunnel to the Resource Manager.

Changed in version 0.6.8: Setting this to an empty value (`--gcloud-bin ''`) instructs mrjob to use the default (used to disable SSH).

1.17 Elastic MapReduce

1.17.1 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
- Go to Security Credentials in the login menu (upper right, third from the right), say yes, you want to proceed, click on **Access Keys**, and then **Create New Access Key**. Make sure to copy the secret access key, as there is no way to recover it after creation.

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 right, second from the right) matches the region you want to run jobs in (usually "Oregon").
- Click on **Key Pairs** (left sidebar, under **Network & Security**)
- Click on **Create Key Pair** (top left).
- 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: 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, use `--output-dir`:

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

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 cluster 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 cluster can lose task instances without killing your job (see *Spot Instances*).

There's a special case where your cluster *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.medium`, 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:

- 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 running **`mrjob boss j-CLUSTERID vmstat`** and then looking in `j-CLUSTERID/*/stdout`.) 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 6 times as much as an `m1.medium`, but it has about 8 times as much processing power (and more memory).

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

```
--instance-type c1.medium --num-core-instances 4
```

or in *`mrjob.conf`*, like this:

```
runners:
  emr:
    instance_type: c1.medium
    num_core_instances: 4
```

In most cases, your master instance type doesn't need to be larger than `m1.medium` to schedule tasks, so *instance_type* only applies to the 4 instances that actually run tasks. You *will* need a larger master instance if you have a very large number of input files; in this case, use the *master_instance_type* option.

The *num_task_instances* option can be used to run 1 or more task instances (these run tasks but don't host HDFS). There are also *core_instance_type* and *task_instance_type* options if you want to set these directly.

1.17.2 Cluster Pooling

Clusters on EMR take several minutes to spin up, which can make development painfully slow.

To get around this, `mrjob` provides **cluster pooling.**. If you set pool_clusters to true, once your job completes, the cluster will stay open to accept additional jobs, and eventually shut itself down after it has been idle for a certain amount of time (by default, ten minutes; see max_mins_idle).

Note: Pooling is a way to reduce latency, not to save money. Though pooling was originally created to optimize AWS's practice of billing by the full hour, this ended in October 2017.

Pooling is designed so that jobs run with the same version of mrjob and the same (or similar) *mrjob.conf* can share the same clusters. Options that affect which cluster a job can join:

- additional_emr_info: (or lack thereof) must match
- applications: must match
- bootstrap: must match, and files referenced must have identical contents
- bootstrap_actions: must match
- image_version/release_label: must match
- image_id (or lack thereof) must match
- ec2_key_pair: if specified, only join clusters with the same key pair
- emr_configurations: (or lack thereof) must match
- subnet: only join clusters with the same EC2 subnet ID (or lack thereof)

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

Pooling is flexible about instance type and number of instances. It will attempt to select the cluster with the greatest CPU capacity (based on `NormalizedInstanceHours` in the cluster summary returned by the `ListClusters` API call), as long as the cluster's instances provide at least as much memory and at least as much CPU as your job requests.

Pooling is also somewhat flexible about EBS volumes (see instance_groups). Each volume must have the same volume type, but larger volumes or volumes with more I/O ops per second are acceptable, as are additional volumes of any type.

Pooling cannot match configurations with explicitly set ebs_root_volume_gb against clusters that use the default (or vice versa) because the EMR API does not report what the default value is.

If you are using instance_fleets, your jobs will only join other clusters which use instance fleets. The rules are similar, but jobs will only join clusters whose fleets use the same set of instances or a subset; there is no concept of "better" instances.

Pooling uses EMR tags to implement a simple "locking" mechanism that keeps two jobs from joining the same cluster simultaneously. Locks automatically expire after a minute (which is more than long enough for a new step to be submitted to the EMR API and enter the `RUNNING` state).

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

1.17.3 EMR runner options

All options from Options available to all runners, Hadoop-related options, and Cloud runner options are available when running jobs on Amazon Elastic MapReduce.

Amazon credentials

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

aws_access_key_id [*string*] **Default:** `None`

"Username" for Amazon web services.

There isn't a command-line switch for this option because credentials are supposed to be secret! Use the environment variable `AWS_ACCESS_KEY_ID` instead.

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

Your "password" on AWS.

There isn't a command-line switch for this option because credentials are supposed to be secret! Use the environment variable `AWS_SECRET_ACCESS_KEY` instead.

aws_session_token [*string*] **Default:** `None`

Temporary AWS session token, used along with aws_access_key_id and aws_secret_access_key when using temporary credentials.

There isn't a command-line switch for this option because credentials are supposed to be secret! Use the environment variable `AWS_SESSION_TOKEN` instead.

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

iam_instance_profile (`--iam-instance-profile`) [*string*] **Default:** (automatic)

Name of an IAM instance profile to use for EC2 clusters created by EMR. See http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-iam-roles.html for more details on using IAM with EMR.

iam_service_role (`--iam-service-role`) [*string*] **Default:** (automatic)

Name of an IAM role for the EMR service to use. See http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-iam-roles.html for more details on using IAM with EMR.

Instance configuration

On EMR, there are three ways to configure instances:

- instance_fleets
- instance_groups
- individual instance options:
  - core_instance_bid_price
  - core_instance_type
  - instance_type
  - master_instance_bid_price
  - master_instance_type
  - num_core_instances
  - num_task_instances
  - task_instance_bid_price,
  - task_instance_type

If there is a conflict, whichever comes later in the config files takes precedence, and the command line beats config files. In the case of a tie, *instance_fleets* beats *instance_groups* beats other instance options.

You may set ebs_root_volume_gb regardless of which style of instance configuration you use.

instance_fleets (`--instance-fleet`) [] **Default:** `None`

A list of instance fleet definitions to pass to the EMR API. Pass a JSON string on the command line or use data structures in the config file (which is itself basically JSON). For example:

```
runners:
  emr:
    instance_fleets:
    - InstanceFleetType: MASTER
      InstanceTypeConfigs:
      - InstanceType: m1.medium
      TargetOnDemandCapacity: 1
    - InstanceFleetType: CORE
      TargetSpotCapacity: 2
      TargetOnDemandCapacity: 2
      LaunchSpecifications:
        SpotSpecification:
          TimeoutDurationMinutes: 20
          TimeoutAction: SWITCH_TO_ON_DEMAND
      InstanceTypeConfigs:
      - InstanceType: m1.medium
        BidPriceAsPercentageOfOnDemandPrice: 50
        WeightedCapacity: 1
      - InstanceType: m1.large
        BidPriceAsPercentageOfOnDemandPrice: 50
        WeightedCapacity: 2
```

instance_groups (`--instance-groups`) [] **Default:** `None`

A list of instance group definitions to pass to the EMR API. Pass a JSON string on the command line or use data structures in the config file (which is itself basically JSON).

This allows for more fine-tuned EBS volume configuration than ebs_root_volume_gb. For example:

```
runners:
  emr:
    instance_groups:
    - InstanceRole: MASTER
      InstanceCount: 1
      InstanceType: m1.medium
    - InstanceRole: CORE
      InstanceCount: 10
      InstanceType: c1.xlarge
      EbsConfiguration:
        EbsOptimized: true
        EbsBlockDeviceConfigs:
        - VolumeSpecification:
            SizeInGB: 100
            VolumeType: gp2
```

instance_groups is incompatible with instance_fleets and other instance options. See instance_fleets for details.

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

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

master_instance_bid_price (--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.

task_instance_bid_price (--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.)

ebs_root_volume_gb (--ebs-root-volume-gb) [integer] **Default:** `None`

When specified (and not zero), sets the size of the root EBS volume, in GiB.

New in version 0.6.5.

Cluster software configuration

See also bootstrap, image_id, and image_version.

applications (--application, --applications) [*string list*] **Default:** `[]`

Additional applications to run on 4.x AMIs (e.g. `'Ganglia'`, `'Mahout'`, `'Spark'`).

You do not need to specify `'Hadoop'`; mrjob will always include it automatically. In most cases it'll auto-detect when to include `'Spark'` as well.

See Applications in the EMR docs for more details.

Changed in version 0.6.7: Added `--applications` switch

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_spark (--bootstrap-spark, --no-bootstrap-spark) [boolean] **Default:** (automatic)

Install Spark on the cluster. This works on AMI version 3.x and later.

By default, we automatically install Spark only if our job has Spark steps.

In case you're curious, here's how mrjob determines you're using Spark:

- any *SparkStep* or *SparkScriptStep* in your job's steps (including implicitly through the *spark* method)
- "Spark" included in applications option
- any bootstrap action (see bootstrap_actions) ending in `/spark-install` (this is how you install Spark on 3.x AMIs)

emr_configurations (--emr-configuration) [list of dicts] **Default:** `[]`

Cluster configs for AMI version 4.x and later. For example:

```
runners:
  emr:
    emr_configurations:
    - Classification: core-site
      Properties:
        hadoop.security.groups.cache.secs: 250
```

On the command line, configurations should be JSON-encoded:

```
--emr-configuration '{"Classification": "core-site", ...}
```

See Configuring Applications in the EMR docs for more details.

Changed in version 0.6.11: `!clear` tag works. Later config dicts will overwrite earlier ones with the same `Classification`. If the later dict has empty `Properties` and `Configurations`, the earlier dict will be simply deleted.

max_concurrent_steps (`--max-concurrent-steps`) [*string*] **Default:** 1

How many steps may an EMR cluster run at the same time? This affects both clusters launched by our job, and, if using cluster pooling, which clusters our job will join.

Prior to AMI 5.28.0, EMR clusters could only ever run one step at a time.

New in version 0.7.4.

release_label (`--release-label`) [*string*] **Default:** `None`

EMR Release to use (e.g. `emr-4.0.0`). This overrides image_version.

For more information about Release Labels, see Differences Introduced in 4.x.

Monitoring your job

See also check_cluster_every, ssh_tunnel.

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

store Hadoop logs in SimpleDB

Cluster pooling

max_clusters_in_pool (`--max-clusters-in-pool`) [integer] **Default:** 0 (disabled)

Don't create a new pooled cluster if there are already this many active (not terminated) clusters in our pool; instead wait until one of the clusters is available to join or terminates.

To deal with the situation where several jobs start at once, before creating a cluster, we wait a random number of seconds (see pool_jitter_seconds and double-check before creating a new cluster).

New in version 0.7.4.

min_available_mb (`--min-available-mb`) [integer] **Default:** 0 (disabled)

When joining a pooled cluster, connect to its YARN resource manager's metrics API and make sure that `availableMB` is at least this high.

This requires SSH to work, so ec2_key_pair and ec2_key_pair_file must be set.

If you enable this option, pooling will no longer query clusters about their instance groups/fleets, since this information is mostly redundant.

New in version 0.7.4.

min_available_virtual_cores (`--min-available-virtual-cores`) [integer] **Default:** 0 (disabled)

When joining a pooled cluster, connect to its YARN resource manager's metrics API and make sure that `availableVirtualCores` is at least this high.

Like with min_available_mb, this requires SSH to work and disables querying clusters about their instances.

New in version 0.7.4.

pool_clusters (`--pool-clusters`) [*string*] **Default:** `True`

Try to run the job on a `WAITING` pooled cluster with the same bootstrap configuration. Prefer the one with the most compute units. If we can't join an existing cluster, create our own (unless max_clusters_in_pool or pool_wait_minutes disallow it).

pool_jitter_seconds (`--pool-jitter-seconds`) [*string*] **Default:** 60

Wait a random number of seconds between 0 and this many before double-checking active clusters in the pool for max_clusters_in_pool or to bypass pool_wait_minutes.

The main point of this option is so that if several jobs start simultaneously, they can double-check if the other jobs have launched a cluster before launching one themselves. You may need wish to adjust this based on your maximum pool size and the number of jobs you expect to launch simultaneously.

New in version 0.7.4.

pool_name (`--pool-name`) [*string*] **Default:** `'default'`

Specify a pool name to join. Does not imply pool_clusters.

pool_timeout_minutes (`--pool-timeout-minutes`) [*string*] **Default:** 0 (disabled)

If we can't create or join a cluster after this many minutes, raise an exception and bail out.

New in version 0.7.4.

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

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

Changed in version 0.7.4: If there aren't any active clusters with a matching pool name and hash, we may create our own cluster before *pool_wait_minutes* is up. We first wait a random number of seconds and double-check that other clusters have not been created (see pool_jitter_seconds).

S3 Filesystem

See also cloud_tmp_dir, cloud_part_size_mb

cloud_log_dir (`--cloud-log-dir`) [*string*] **Default:** append `logs` to cloud_tmp_dir

Where on S3 to put logs, for example `s3://yourbucket/logs/`. Logs for your cluster will go into a subdirectory, e.g. `s3://yourbucket/logs/j-CLUSTERID/`.

Docker

docker_client_config (`--docker-client-config`) [*string*] **Default:** `None`

An `hdfs://` URI pointing to the client config, which is used to authenticate with a private Docker registry (e.g. ECR). This is mostly useful on AMIs prior to 6.1.0; otherwise you can use auto-authentication (see this page).

See "Using ECR" on this page for information about how to fetch working credentials. Because ECR credentials only last 12 hours, if you want to use ECR and Docker for multiple jobs on a long-running cluster, you may wish to set up a cron job at bootstrap time.

New in version 0.7.4.

docker_image (`--docker-image`, `--no-docker`) [*string*] **Default:** `None`

The repository, name, and optionally, tag of a docker image, in the format `registry/repository:tag`. If `registry/` is omitted, we assume the default registry on Docker Hub (`library`). If registry is a hostname, we connect to that host instead (e.g. for use of ECR).

Other `docker_*` options will do nothing if this is not set.

Note that you must be running at least AMI 6.0.0 to use Docker on EMR.

New in version 0.7.4.

docker_mounts (`--docker-mount`) [*string list*] **Default:** `[]`

Optional mounting instructions to pass to Docker, in the format `/local/path:/path/inside/docker:ro_or_rw`.

New in version 0.7.4.

API Endpoints

Note: You usually don't want to set `*_endpoint` options unless you have a challenging network situation (e.g. you have to use a proxy to get around a firewall).

ec2_endpoint (`--ec2-endpoint`) [*string*] **Default:** (automatic)

New in version 0.6.5.

Force mrjob to connect to EC2 on this endpoint (e.g. `ec2.us-gov-west-1.amazonaws.com`).

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

Force mrjob to connect to EMR on this endpoint (e.g. `us-west-1.elasticmapreduce.amazonaws.com`).

iam_endpoint (`--iam-endpoint`) [*string*] **Default:** (automatic)

Force mrjob to connect to IAM on this endpoint (e.g. `iam.us-gov.amazonaws.com`).

s3_endpoint (`--s3-endpoint`) [*string*] **Default:** (automatic)

Force mrjob to connect to S3 on this endpoint, rather than letting it choose the appropriate endpoint for each S3 bucket.

Warning: If you set this to a region-specific endpoint (e.g. `'s3-us-west-1.amazonaws.com'`) mrjob may not be able to access buckets located in other regions.

Other rarely used options

add_steps_in_batch (`--add-steps-in-batch`, `--no-add-steps-in-batch`) [boolean] **Default:** `True` for AMIs before 5.28.0, `False` otherwise

For a multi-step job, should we submit all steps at once, or one at a time? By default, we only submit steps all at once if the AMI doesn't support running concurrent steps (that is, before AMI 5.28.0).

New in version 0.7.4.

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).

emr_action_on_failure (`--emr-action-on-failure`) [*string*] **Default:** (automatic)

What happens if step of your job fails

- `'CANCEL_AND_WAIT'` cancels all steps on the cluster
- **`'CONTINUE'` continues to the next step (useful when submitting several** jobs to the same cluster)
- `'TERMINATE_CLUSTER'` shuts down the cluster entirely

The default is `'CANCEL_AND_WAIT'` when using pooling (see pool_clusters) or an existing cluster (see cluster_id), and `'TERMINATE_CLUSTER'` otherwise.

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

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

Path to the `ssh-add` binary. Used on EMR to access logs on the non-master node, without copying your SSH key to the master node.

New in version 0.7.2.

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`**.

On EMR, mrjob uses SSH to tunnel to the job tracker (see ssh_tunnel), as a fallback way of fetching job progress, and as a quicker way of accessing your job's logs.

Changed in version 0.6.8: Setting this to an empty value (`--ssh-bin ''`) instructs mrjob to use the default value (used to effectively disable SSH).

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

Metadata tags to apply to the EMR cluster after its creation. See Tagging Amazon EMR Clusters for more information on applying metadata tags to EMR clusters.

Tag names and values are strings. On the command line, to set a tag use `--tag KEY=VALUE`:

```
--tag team=development
```

In the config file, `tags` is a dict:

```
runners:
  emr:
    tags:
      team: development
      project: mrjob
```

1.17.4 EMR Bootstrapping Cookbook

Bootstrapping allows you to run commands to customize EMR machines, at the time the cluster is created.

When to use bootstrap, 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 using *Cluster Pooling*.

EMR will generally not allow you to use **`sudo`** in setup commands. See Job Environment Setup Cookbook for how to install libraries, etc. without using **`sudo`**.

Installing Python packages with pip

The only tricky thing is making sure you install packages for the correct version of Python. Figure out which version of Python you'll be running on EMR (see python_bin for defaults).

- If it's Python 2, use **`pip-2.7`** (just plain **`pip`** also works on AMI 4.3.0 and later)
- If it's Python 3, use **`pip-3.6`** on AMI 5.20.0+, and **`pip-3.4`** for earlier AMIs

For example, to install `ujson` on Python 2:

```
runners:
  emr:
    bootstrap:
    - sudo pip-2.7 install ujson
```

See PyPI for a the full list of available Python packages.

You can also install packages from a requirements file:

```
runners:
  emr:
    bootstrap:
    - sudo pip-2.7 install -r /local/path/of/requirements.txt#
```

Or a tarball:

```
runners:
  emr:
    bootstrap:
    - sudo pip-2.7 install /local/path/of/tarball.tar.gz#
```

Warning: If you're trying to run jobs on AMI version 3.0.0 (protip: don't do that) **`pip`** appears not to work due to out-of-date SSL certificate information.

Installing PyPy

First, download the version of PyPy you want to use from Portable PyPy Distributions for Linux.

Then instruct EMR to un-tar it and link to the binary in `/usr/bin`. For example:

```
runners:
  emr:
    bootstrap:
    - sudo tar xvfj /local/path/to/pypy-7.1.1-linux_x86_64-portable.tar.bz2# -C /opt
    - sudo ln -s /opt/pypy-7.1.1-linux_x86_64-portable/bin/pypy /usr/bin/pypy
```

Installing System Packages

EMR gives you access to a variety of different Amazon Machine Images, or AMIs for short (see image_version).

3.x and later AMIs

Starting with 3.0.0, EMR AMIs use Amazon Linux, which uses **`yum`** to install packages. For example, to install NumPy:

```
runners:
  emr:
    bootstrap:
    - sudo yum install -y python-numpy
```

(Don't forget the `-y`!)

Amazon Linux's Python packages generally only work for Python 2. If you're on Python 3, just *use pip*.

The most recent list of Amazon linux packages can be found here (click on "Packages List" in the left sidebar).

2.x AMIs

Probably not worth the trouble. The 2.x AMIs are based on a version of Debian that is so old it has been "archived," which makes their package installer, **`apt-get`**, no longer work out-of-the-box. Moreover, Python system packages work for Python 2.6, not 2.7.

Instead, just use **`pip-2.7`** to install Python libraries.

1.17.5 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.

In addition to looking at S3, `mrjob` can be configured to also use SSH to fetch error logs directly from the master and worker nodes. This can speed up debugging significantly (EMR only transfers logs to S3 every five minutes).

Using persistent clusters

When troubleshooting a job, it can be convenient to use a persistent cluster to avoid having to wait for bootstrapping every run.

First, use the **`mrjob create-cluster`** to create a persistent cluster:

```
$ mrjob create-cluster
Using configs in /Users/davidmarin/.mrjob.conf
Using s3://mrjob-35cdec11663cb1cb/tmp/ as our temp dir on S3
Creating persistent cluster to run several jobs in...
Creating temp directory /var/folders/zv/jmtt5bxs6xl3kzt38470hcxm0000gn/T/no_script.davidmarin.201603
Copying local files to s3://mrjob-35cdec11663cb1cb/tmp/no_script.davidmarin.20160324.231018.720057/fi
j-3BYHP30KB81XE
```

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

```
$ python mrjob/examples/mr_boom.py README.rst -r emr --cluster-id j-3BYHP30KB81XE
Using configs in /Users/davidmarin/.mrjob.conf
Using s3://mrjob-35cdec11663cb1cb/tmp/ as our temp dir on S3
Creating temp directory /var/folders/zv/jmtt5bxs6xl3kzt38470hcxm0000gn/T/mr_boom.davidmarin.20160324
Copying local files to s3://mrjob-35cdec11663cb1cb/tmp/mr_boom.davidmarin.20160324.231045.501027/file
Adding our job to existing cluster j-3BYHP30KB81XE
Waiting for step 1 of 1 (s-SGVW9B5LEXF5) to complete...
  PENDING (cluster is STARTING: Provisioning Amazon EC2 capacity)
  PENDING (cluster is STARTING: Provisioning Amazon EC2 capacity)
  PENDING (cluster is STARTING: Provisioning Amazon EC2 capacity)
  PENDING (cluster is STARTING: Provisioning Amazon EC2 capacity)
```

```
  PENDING (cluster is STARTING: Provisioning Amazon EC2 capacity)
  PENDING (cluster is BOOTSTRAPPING: Running bootstrap actions)
  PENDING (cluster is BOOTSTRAPPING: Running bootstrap actions)
  PENDING (cluster is BOOTSTRAPPING: Running bootstrap actions)
  PENDING (cluster is BOOTSTRAPPING: Running bootstrap actions)
  PENDING (cluster is BOOTSTRAPPING: Running bootstrap actions)
  Opening ssh tunnel to resource manager...
  Connect to resource manager at: http://localhost:40069/cluster
  RUNNING for 9.2s
  RUNNING for 42.3s
     0.0% complete
  RUNNING for 72.6s
     5.0% complete
  RUNNING for 102.9s
     5.0% complete
  RUNNING for 133.4s
    100.0% complete
  FAILED
Cluster j-3BYHP30KB81XE is WAITING: Cluster ready after last step failed.
Attempting to fetch counters from logs...
Looking for step log in /mnt/var/log/hadoop/steps/s-SGVW9B5LEXF5 on ec2-52-37-112-240.us-west-2.compu
  Parsing step log: ssh://ec2-52-37-112-240.us-west-2.compute.amazonaws.com/mnt/var/log/hadoop/steps/
Counters: 9
    Job Counters
            Data-local map tasks=1
            Failed map tasks=4
            Launched map tasks=4
            Other local map tasks=3
            Total megabyte-seconds taken by all map tasks=58125312
            Total time spent by all map tasks (ms)=75684
            Total time spent by all maps in occupied slots (ms)=227052
            Total time spent by all reduces in occupied slots (ms)=0
            Total vcore-seconds taken by all map tasks=75684
Scanning logs for probable cause of failure...
Looking for task logs in /mnt/var/log/hadoop/userlogs/application_1458861299388_0001 on ec2-52-37-112
  Parsing task syslog: ssh://ec2-52-37-112-240.us-west-2.compute.amazonaws.com/mnt/var/log/hadoop/use
  Parsing task stderr: ssh://ec2-52-37-112-240.us-west-2.compute.amazonaws.com/mnt/var/log/hadoop/use
Probable cause of failure:

PipeMapRed failed!
java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess failed with code 1
    at org.apache.hadoop.streaming.PipeMapRed.waitOutputThreads(PipeMapRed.java:330)
    at org.apache.hadoop.streaming.PipeMapRed.mapRedFinished(PipeMapRed.java:543)
    at org.apache.hadoop.streaming.PipeMapper.close(PipeMapper.java:130)
    at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:81)
    at org.apache.hadoop.streaming.PipeMapRunner.run(PipeMapRunner.java:34)
    at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:432)
    at org.apache.hadoop.mapred.MapTask.run(MapTask.java:343)
    at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild.java:175)
    at java.security.AccessController.doPrivileged(Native Method)
    at javax.security.auth.Subject.doAs(Subject.java:415)
    at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1548)
    at org.apache.hadoop.mapred.YarnChild.main(YarnChild.java:170)

(from lines 37-50 of ssh://ec2-52-37-112-240.us-west-2.compute.amazonaws.com/mnt/var/log/hadoop/user

caused by:
```

```
Traceback (most recent call last):
  File "mr_boom.py", line 10, in <module>
    MRBoom.run()
  File "/usr/lib/python3.4/dist-packages/mrjob/job.py", line 430, in run
    mr_job.execute()
  File "/usr/lib/python3.4/dist-packages/mrjob/job.py", line 439, in execute
    self.run_mapper(self.options.step_num)
  File "/usr/lib/python3.4/dist-packages/mrjob/job.py", line 499, in run_mapper
    for out_key, out_value in mapper_init() or ():
  File "mr_boom.py", line 7, in mapper_init
    raise Exception('BOOM')
Exception: BOOM

(from lines 1-12 of ssh://ec2-52-37-112-240.us-west-2.compute.amazonaws.com/mnt/var/log/hadoop/userlo

while reading input from s3://mrjob-35cdec11663cb1cb/tmp/mr_boom.davidmarin.20160324.231045.501027/fi


Step 1 of 1 failed
Killing our SSH tunnel (pid 52847)
```

Now you can fix the bug and try again, without having to wait for a new cluster to bootstrap.

Note: mrjob *can* fetch logs from persistent jobs even without SSH set up, but it has to pause 10 minutes to wait for EMR to transfer logs to S3, which defeats the purpose of rapid iteration.

1.17.6 Advanced EMR usage

Spot Instances

You can potentially save money purchasing EC2 instances for your EMR clusters from AWS's spot market. The catch is that if someone bids more for instances that you're using, they can be taken away from your cluster. If this happens, you aren't charged, but your job may fail.

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

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

or in *mrjob.conf*:

```
runners:
  emr:
    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.

Cluster 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

See *Installing Python packages with pip* and *Installing System Packages*.

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"
```

Note: This doesn't work on AMI 4.0.0 and later.

Manually Reusing Clusters

In some cases, it may be useful to have more fine-grained control than *Cluster Pooling* provides; for example, to run several related jobs on the same cluster.

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

```
$ mrjob create-cluster --num-core-instances=11
...
j-CLUSTERID
```

You can then add jobs to the cluster with the `--cluster-id` switch or the cluster_id option in *mrjob.conf* (see `EMRJobRunner.__init__()`):

```
$ python mr_my_job.py -r emr --cluster-id=j-CLUSTERID input_file.txt > out
...
Adding our job to existing cluster j-CLUSTERID
...
```

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 cluster terminates.

1.18 Python 2 vs. Python 3

1.18.1 Raw protocols

Both because we don't want to break mrjob for Python 2 users, and to make writing jobs simple, jobs read their input as `str`s by default (even though `str` means bytes in Python 2 and unicode in Python 3).

The way this works in mrjob is that *`RawValueProtocol`* is actually an alias for one of two classes, *`BytesValueProtocol`* if you're in Python 2, and *`TextValueProtocol`* if you're in Python 3.

If you care about this distinction, you may want to explicitly set *`INPUT_PROTOCOL`* to one of these. If your input has a well-defined encoding, probably you want *`BytesValueProtocol`*, and if it's a bunch of text that's mostly ASCII, with like, some stuff that... might be UTF-8? (i.e. most log files), you probably want *`TextValueProtocol`*. But most of the time it'll just work.

1.18.2 Bytes vs. strings

The following things are bytes in any version of Python (which means you need to use the `bytes` type and/or `b'...'` constant

- data read or written by *Protocols*
- lines yielded by *`cat_output()`*
- anything read from *`cat()`*

The `stdin`, `stdout`, and `stderr` attributes of *`MRJob`*s are always bytestreams (so, for example, `self.stderr` defaults to `sys.stderr.buffer` in Python 3).

Everything else (including file paths, URIs, arguments to commands, and logging messages) are strings; that is, `str`s on Python 3, and either `unicode`s or ASCII `str`s on Python 2. Like with *`RawValueProtocol`*, most of the time it'll just work even if you don't think about it.

1.18.3 python_bin

python_bin defaults to **`python3`** in Python 3, and **`python`** in Python 2 (except on EMR AMIs prior to 4.3.0, where we use **`python2.7`**)

1.18.4 Your Hadoop cluster

Whatever version of Python you use, you'll have to have a compatible version of Python installed on your Hadoop cluster. mrjob does its best to make this work on Elastic MapReduce (see bootstrap_python), but if you're running on your own Hadoop cluster, this is up to you.

1.19 Contributing to mrjob

1.19.1 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:

- Add command line switches that allow full control over the option
- Document the option and its switches in the appropriate file under `docs`

1.19.2 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
  - Google Cloud Dataproc
    - *mrjob.dataproc*: Submit jobs to Dataproc
  - Amazon Elastic MapReduce
    - *mrjob.emr*: Submit jobs to EMR
    - `mrjob.pool`: Utilities for cluster pooling functionality
    - *mrjob.retry*: Wrapper for S3 and EMR connections to handle recoverable errors
- Interacting with different "filesystems"
  - *mrjob.fs.base*: Common functionality
  - `mrjob.fs.composite`: Support multiple filesystems; if one fails, "fall through" to another
  - `mrjob.fs.gcs`: Google Cloud Storage
  - `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.logs: Log interpretation (counters, probable cause of job failure)
  - *mrjob.parse*: Parsing utilities for URIs, command line options, etc.
  - *mrjob.util*: Utilities for dealing with files, command line options, various other things

CHAPTER 2

Reference

2.1 mrjob.ami - building custom AMIs

Utilities for creating custom AMIs.

mrjob.ami.**describe_base_emr_images**(*ec2_client*)

Fetch a list of Amazon Linux AMI images that are usable with EMR, with the most recent first. This can take several seconds.

Parameters **ec2_client** – a boto3 EC2 client, which can be obtained from *mrjob.emr.EMRJobRunner.make_ec2_client()* or `boto3.client('ec2')`

For the sake of consistency, we have somewhat stricter requirements than the AWS documentation. Specifically:

•Amazon Linux (not Amazon Linux 2)

•HVM virtualization

•x86_64 architecture

•single EBS volume * standard volume type (not GP2)

•stable version (no "testing" or "rc", only numbers and dots)

This only returns images going back to September 2016 (prior to that, EC2 used a different naming convention).

This returns a dictionary for each image, in the same response format as ec2_client.describe_images(). The *ImageId* field contains the AMI ID, and *Description* contains a human-readable description.

2.2 mrjob.cat - decompress files based on extension

Emulating the way Hadoop handles input files, decompressing compressed files based on their file extension.

This module also functions as a **cat** substitute that can handle compressed files. It it used by *local* mode and can function without the rest of the mrjob library.

mrjob.cat.**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 decompressed chunks; it does *not* split on lines. To get lines, wrap this in `to_lines()`.

`mrjob.cat.`**`decompress`**(*readable*, *path*, *bufsize=1024*)
Take a *readable* which supports the `.read()` method correponding to the given path and returns an iterator that yields chunks of bytes, possibly decompressing based on *path*.

if *readable* appears to be a fileobj, pass it through as-is.

if *readable* does not have a `read()` method, assume that it's a generator that yields chunks of bytes

`mrjob.cat.`**`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 `to_lines()`.

`mrjob.cat.`**`to_chunks`**(*readable*, *bufsize=1024*)
Convert *readable*, which is any object supporting `read()` (e.g. fileobjs) to a stream of non-empty `bytes`.

If *readable* has an `__iter__` method but not a `read` method, pass through as-is.

2.3 mrjob.cmd: The `mrjob` command-line utility

The **`mrjob`** command provides a number of sub-commands that help you run and monitor jobs.

The **`mrjob`** command comes with Python-version-specific aliases (e.g. **`mrjob-3`**, **`mrjob-3.4`**), in case you choose to install `mrjob` for multiple versions of Python. You may also run it as **`python -m mrjob.cmd <subcommand>`**.

2.3.1 audit-emr-usage

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

Usage:

```
mrjob audit-emr-usage > report
```

Options:

```
-c CONF_PATHS, --conf-path CONF_PATHS
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--ec2-endpoint EC2_ENDPOINT
                      Force mrjob to connect to EC2 on this endpoint (e.g.
                      ec2.us-west-1.amazonaws.com). Default is to infer this
                      from region.
--emr-endpoint EMR_ENDPOINT
                      Force mrjob to connect to EMR on this endpoint (e.g.
                      us-west-1.elasticmapreduce.amazonaws.com). Default is
```

```
                        to infer this from region.
    -h, --help          show this help message and exit
    --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)
    -q, --quiet         Don't print anything to stderr
    --region REGION     GCE/AWS region to run Dataproc/EMR jobs in.
    --s3-endpoint S3_ENDPOINT
                        Force mrjob to connect to S3 on this endpoint (e.g. s3
                        -us-west-1.amazonaws.com). You usually shouldn't set
                        this; by default mrjob will choose the correct
                        endpoint for each S3 bucket based on its location.
    -v, --verbose       print more messages to stderr
```

2.3.2 boss

Run a command on every node of a cluster. Store stdout and stderr for results in OUTPUT_DIR.

Usage:

```
mrjob boss CLUSTER_ID [options] "command string"
```

Options:

```
    -c CONF_PATHS, --conf-path CONF_PATHS
                        Path to alternate mrjob.conf file to read from
    --no-conf           Don't load mrjob.conf even if it's available
    --ec2-endpoint EC2_ENDPOINT
                        Force mrjob to connect to EC2 on this endpoint (e.g.
                        ec2.us-west-1.amazonaws.com). Default is to infer this
                        from region.
    --ec2-key-pair-file EC2_KEY_PAIR_FILE
                        Path to file containing SSH key for EMR
    --emr-endpoint EMR_ENDPOINT
                        Force mrjob to connect to EMR on this endpoint (e.g.
                        us-west-1.elasticmapreduce.amazonaws.com). Default is
                        to infer this from region.
    -h, --help          show this help message and exit
    -o OUTPUT_DIR, --output-dir OUTPUT_DIR
                        Specify an output directory (default: CLUSTER_ID)
    -q, --quiet         Don't print anything to stderr
    --region REGION     GCE/AWS region to run Dataproc/EMR jobs in.
    --s3-endpoint S3_ENDPOINT
                        Force mrjob to connect to S3 on this endpoint (e.g. s3
                        -us-west-1.amazonaws.com). You usually shouldn't set
                        this; by default mrjob will choose the correct
                        endpoint for each S3 bucket based on its location.
    --ssh-bin SSH_BIN   Name/path of ssh binary. Arguments are allowed (e.g.
                        --ssh-bin 'ssh -v')
    -v, --verbose       print more messages to stderr
```

2.3.3 create-cluster

Create a persistent EMR cluster to run clusters in, and print its ID to stdout.

Usage:

```
mrjob create-cluster
```

Options:

```
--additional-emr-info ADDITIONAL_EMR_INFO
                      A JSON string for selecting additional features on EMR
--applications APPLICATIONS, --application APPLICATIONS
                      Additional applications to run on 4.x and 5.x AMIs,
                      separated by commas (e.g. "Ganglia,Spark")
--bootstrap BOOTSTRAP
                      A shell command to set up libraries etc. before any
                      steps (e.g. "sudo apt-get -qy install python3"). You
                      may interpolate files available via URL or locally
                      with Hadoop Distributed Cache syntax ("sudo yum
                      install -y foo.rpm#")
--bootstrap-action BOOTSTRAP_ACTIONS
                      Raw bootstrap action scripts to run before any of the
                      other bootstrap steps. You can use --bootstrap-action
                      more than once. Local scripts will be automatically
                      uploaded to S3. To add arguments, just use quotes:
                      "foo.sh arg1 arg2"
--bootstrap-mrjob     Automatically zip up the mrjob library and install it
                      when we run the mrjob. This is the default. Use --no-
                      bootstrap-mrjob if you've already installed mrjob on
                      your Hadoop cluster.
--no-bootstrap-mrjob  Don't automatically zip up the mrjob library and
                      install it when we run this job. Use this if you've
                      already installed mrjob on your Hadoop cluster.
--bootstrap-python    Attempt to install a compatible version of Python at
                      bootstrap time. Currently this only does anything for
                      Python 3, for which it is enabled by default.
--no-bootstrap-python
                      Don't automatically try to install a compatible
                      version of Python at bootstrap time.
--bootstrap-spark     Auto-install Spark on the cluster (even if not
                      needed).
--no-bootstrap-spark  Don't auto-install Spark on the cluster.
--cloud-fs-sync-secs CLOUD_FS_SYNC_SECS
                      How long to wait for remote FS to reach eventual
                      consistency. This is typically less than a second but
                      the default is 5.0 to be safe.
--cloud-log-dir CLOUD_LOG_DIR
                      URI on remote FS to write logs into
--cloud-part-size-mb CLOUD_PART_SIZE_MB
                      Upload files to cloud FS in parts no bigger than this
                      many megabytes. Default is 100 MiB. Set to 0 to
                      disable multipart uploading entirely.
--cloud-upload-part-size CLOUD_PART_SIZE_MB
                      Deprecated alias for --cloud-part-size-mb
--cloud-tmp-dir CLOUD_TMP_DIR
                      URI on remote FS to use as our temp directory.
-c CONF_PATHS, --conf-path CONF_PATHS
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--core-instance-bid-price CORE_INSTANCE_BID_PRICE
                      Bid price to specify for core nodes when setting them
                      up as EC2 spot instances (you probably only want to do
                      this for task instances).
```

```
    --core-instance-type CORE_INSTANCE_TYPE
                          Type of GCE/EC2 core instance(s) to launch
    --ebs-root-volume-gb EBS_ROOT_VOLUME_GB
                          Size of root EBS volume, in GiB. Must be an
                          integer.Set to 0 to use the default
    --ec2-endpoint EC2_ENDPOINT
                          Force mrjob to connect to EC2 on this endpoint (e.g.
                          ec2.us-west-1.amazonaws.com). Default is to infer this
                          from region.
    --ec2-key-pair EC2_KEY_PAIR
                          Name of the SSH key pair you set up for EMR
    --emr-action-on-failure EMR_ACTION_ON_FAILURE
                          Action to take when a step fails (e.g.
                          TERMINATE_CLUSTER, CANCEL_AND_WAIT, CONTINUE)
    --emr-configuration EMR_CONFIGURATIONS
                          Configuration to use on 4.x AMIs as a JSON-encoded
                          dict; see http://docs.aws.amazon.com/ElasticMapReduce/
                          latest/ReleaseGuide/emr-configure-apps.html for
                          examples
    --emr-endpoint EMR_ENDPOINT
                          Force mrjob to connect to EMR on this endpoint (e.g.
                          us-west-1.elasticmapreduce.amazonaws.com). Default is
                          to infer this from region.
    --enable-emr-debugging
                          Enable storage of Hadoop logs in SimpleDB
    --disable-emr-debugging
                          Disable storage of Hadoop logs in SimpleDB (the
                          default)
    --extra-cluster-param EXTRA_CLUSTER_PARAMS
                          extra parameter to pass to cloud API when creating a
                          cluster, to access features not currently supported by
                          mrjob. Takes the form <param>=<value>, where value is
                          JSON or a string. Use <param>=null to unset a
                          parameter
    -h, --help            show this help message and exit
    --iam-endpoint IAM_ENDPOINT
                          Force mrjob to connect to IAM on this endpoint (e.g.
                          iam.us-gov.amazonaws.com)
    --iam-instance-profile IAM_INSTANCE_PROFILE
                          EC2 instance profile to use for the EMR cluster -- see
                          "Configure IAM Roles for Amazon EMR" in AWS docs
    --iam-service-role IAM_SERVICE_ROLE
                          IAM service role to use for the EMR cluster -- see
                          "Configure IAM Roles for Amazon EMR" in AWS docs
    --image-id IMAGE_ID   ID of custom AWS machine image (AMI) to use
    --image-version IMAGE_VERSION
                          version of EMR/Dataproc machine image to run
    --instance-fleets INSTANCE_FLEETS
                          detailed JSON list of instance fleets, including EBS
                          configuration. See docs for --instance-fleets at
                          http://docs.aws.amazon.com/cli/latest/reference/emr
                          /create-cluster.html
    --instance-groups INSTANCE_GROUPS
                          detailed JSON list of EMR instance configs, including
                          EBS configuration. See docs for --instance-groups at
                          http://docs.aws.amazon.com/cli/latest/reference/emr
                          /create-cluster.html
    --instance-type INSTANCE_TYPE
```

```
                      Type of GCE/EC2 instance(s) to launch GCE - e.g.
                      n1-standard-1, n1-highcpu-4, n1-highmem-4 -- See
                      https://cloud.google.com/compute/docs/machine-types
                      EC2 - e.g. m1.medium, c3.xlarge, r3.xlarge -- See
                      http://aws.amazon.com/ec2/instance-types/
  --label LABEL       Alternate label for the job, to help us identify it.
  --master-instance-bid-price MASTER_INSTANCE_BID_PRICE
                      Bid price to specify for the master node when setting
                      it up as an EC2 spot instance (you probably only want
                      to do this for task instances).
  --master-instance-type MASTER_INSTANCE_TYPE
                      Type of GCE/EC2 master instance to launch
  --max-mins-idle MAX_MINS_IDLE
                      If we create a cluster, have it automatically
                      terminate itself after it's been idle this many
                      minutes
  --num-core-instances NUM_CORE_INSTANCES
                      Total number of core instances to launch
  --num-task-instances NUM_TASK_INSTANCES
                      Total number of task instances to launch
  --owner OWNER       User who ran the job (default is the current user)
  --pool-clusters     Add to an existing cluster or create a new one that
                      does not terminate when the job completes.
  --no-pool-clusters  Don't run job on a pooled cluster (the default)
  --pool-name POOL_NAME
                      Specify a pool name to join. Default is "default"
  -q, --quiet         Don't print anything to stderr
  --region REGION     GCE/AWS region to run Dataproc/EMR jobs in.
  --release-label RELEASE_LABEL
                      Release Label (e.g. "emr-4.0.0"). Overrides --image-
                      version
  --s3-endpoint S3_ENDPOINT
                      Force mrjob to connect to S3 on this endpoint (e.g. s3
                      -us-west-1.amazonaws.com). You usually shouldn't set
                      this; by default mrjob will choose the correct
                      endpoint for each S3 bucket based on its location.
  --subnet SUBNET     ID of Amazon VPC subnet/URI of Google Compute Engine
                      subnetwork to launch cluster in.
  --subnets SUBNET    Like --subnet, but with a comma-separated list, to
                      specify multiple subnets in conjunction with
                      --instance-fleets (EMR only)
  --tag TAGS          Metadata tags to apply to the EMR cluster; should take
                      the form KEY=VALUE. You can use --tag multiple times
  --task-instance-bid-price TASK_INSTANCE_BID_PRICE
                      Bid price to specify for task nodes when setting them
                      up as EC2 spot instances
  --task-instance-type TASK_INSTANCE_TYPE
                      Type of GCE/EC2 task instance(s) to launch
  -v, --verbose       print more messages to stderr
  --zone ZONE         GCE zone/AWS availability zone to run Dataproc/EMR
                      jobs in.
```

2.3.4 diagnose

Print probable cause of error for a failed step.

Currently this only works on EMR.

Usage:

```
mrjob diagnose [opts] j-CLUSTERID
```

Options:

```
-c CONF_PATHS, --conf-path CONF_PATHS
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--ec2-endpoint EC2_ENDPOINT
                      Force mrjob to connect to EC2 on this endpoint (e.g.
                      ec2.us-west-1.amazonaws.com). Default is to infer this
                      from region.
--emr-endpoint EMR_ENDPOINT
                      Force mrjob to connect to EMR on this endpoint (e.g.
                      us-west-1.elasticmapreduce.amazonaws.com). Default is
                      to infer this from region.
-h, --help            show this help message and exit
-q, --quiet           Don't print anything to stderr
--region REGION       GCE/AWS region to run Dataproc/EMR jobs in.
--s3-endpoint S3_ENDPOINT
                      Force mrjob to connect to S3 on this endpoint (e.g. s3
                      -us-west-1.amazonaws.com). You usually shouldn't set
                      this; by default mrjob will choose the correct
                      endpoint for each S3 bucket based on its location.
--step-id STEP_ID     ID of a particular failed step to diagnose
-v, --verbose         print more messages to stderr
```

New in version 0.6.1.

2.3.5 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
```

Options:

```
-c CONF_PATHS, --conf-path CONF_PATHS
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--ec2-endpoint EC2_ENDPOINT
                      Force mrjob to connect to EC2 on this endpoint (e.g.
                      ec2.us-west-1.amazonaws.com). Default is to infer this
                      from region.
--emr-endpoint EMR_ENDPOINT
                      Force mrjob to connect to EMR on this endpoint (e.g.
                      us-west-1.elasticmapreduce.amazonaws.com). Default is
                      to infer this from region.
-x EXCLUDE, --exclude EXCLUDE
                      Exclude clusters that match the specified tags.
                      Specifed in the form TAG_KEY,TAG_VALUE.
-h, --help            show this help message and exit
--min-hours MIN_HOURS
                      Minimum number of hours a job can run before we report
                      it. Default: 24.0
```

```
    -q, --quiet           Don't print anything to stderr
    --region REGION       GCE/AWS region to run Dataproc/EMR jobs in.
    --s3-endpoint S3_ENDPOINT
                          Force mrjob to connect to S3 on this endpoint (e.g. s3
                          -us-west-1.amazonaws.com). You usually shouldn't set
                          this; by default mrjob will choose the correct
                          endpoint for each S3 bucket based on its location.
    -v, --verbose         print more messages to stderr
```

2.3.6 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/
```

Usage:

```
    mrjob s3-tmpwatch [options] <time-untouched> <URIs>
```

Options:

```
    -c CONF_PATHS, --conf-path CONF_PATHS
                          Path to alternate mrjob.conf file to read from
    --no-conf             Don't load mrjob.conf even if it's available
    -h, --help            show this help message and exit
    -q, --quiet           Don't print anything to stderr
    --region REGION       GCE/AWS region to run Dataproc/EMR jobs in.
    --s3-endpoint S3_ENDPOINT
                          Force mrjob to connect to S3 on this endpoint (e.g. s3
                          -us-west-1.amazonaws.com). You usually shouldn't set
                          this; by default mrjob will choose the correct
                          endpoint for each S3 bucket based on its location.
    -t, --test            Don't actually delete any files; just log that we
                          would
    -v, --verbose         print more messages to stderr
```

2.3.7 spark-submit

A drop-in replacement for **`spark-submit`** that can use mrjob's runners. For example, you can submit your spark job to EMR just by adding `-r emr`.

This also adds a few mrjob features that are not standard with **`spark-submit`**, such as `--cmdenv`, `--dirs`, and `--setup`.

New in version 0.6.7.

Changed in version 0.6.8: added `local`, `spark` runners, made `spark` the default (was `hadoop`)

Changed in version 0.7.1: `--archives` and `--dirs` are supported on all masters (except local)

Usage:

```
mrjob spark-submit [-r <runner>] [options] <python file | app jar>
[app arguments]
```

Options:

```
All runners:
 -r {emr,hadoop,local,spark}, --runner {emr,hadoop,local,spark}
                       Where to run the job (default: "spark")
 --class MAIN_CLASS    Your application's main class (for Java / Scala apps).
 --name NAME           The name of your application.
 --jars LIBJARS        Comma-separated list of jars to include on the
                       driverand executor classpaths.
 --packages PACKAGES   Comma-separated list of maven coordinates of jars to
                       include on the driver and executor classpaths. Will
                       search the local maven repo, then maven central and
                       any additional remote repositories given by
                       --repositories. The format for the coordinates should
                       be groupId:artifactId:version.
 --exclude-packages EXCLUDE_PACKAGES
                       Comma-separated list of groupId:artifactId, to exclude
                       while resolving the dependencies provided in
                       --packages to avoid dependency conflicts.
 --repositories REPOSITORIES
                       Comma-separated list of additional remote repositories
                       to search for the maven coordinates given with
                       --packages.
 --py-files PY_FILES   Comma-separated list of .zip, .egg, or .py files to
                       placed on the PYTHONPATH for Python apps.
 --files UPLOAD_FILES  Comma-separated list of files to be placed in the
                       working directory of each executor. Ignored on
                       local[*] master.
 --archives UPLOAD_ARCHIVES
                       Comma-separated list of archives to be extracted into
                       the working directory of each executor.
 --dirs UPLOAD_DIRS    Comma-separated list of directors to be archived and
                       then extracted into the working directory of each
                       executor.
 --cmdenv CMDENV       Arbitrary environment variable to set inside Spark, in
                       the format NAME=VALUE.
 --conf JOBCONF        Arbitrary Spark configuration property, in the format
                       PROP=VALUE.
 --setup SETUP         A command to run before each Spark executor in the
                       shell ("touch foo"). In cluster mode, runs before the
                       Spark driver as well. You may interpolate files
                       available via URL or on your local filesystem using
                       Hadoop Distributed Cache syntax (". setup.sh#"). To
                       interpolate archives (YARN only), use #/: "cd
                       foo.tar.gz#/; make.
 --properties-file PROPERTIES_FILE
                       Path to a file from which to load extra properties. If
                       not specified, this will look for conf/spark-
                       defaults.conf.
 --driver-memory DRIVER_MEMORY
                       Memory for driver (e.g. 1000M, 2G) (Default: 1024M).
 --driver-java-options DRIVER_JAVA_OPTIONS
                       Extra Java options to pass to the driver.
 --driver-library-path DRIVER_LIBRARY_PATH
                       Extra library path entries to pass to the driver.
```

```
  --driver-class-path DRIVER_CLASS_PATH
                        Extra class path entries to pass to the driver. Note
                        that jars added with --jars are automatically included
                        in the classpath.
  --executor-memory EXECUTOR_MEMORY
                        Memory per executor (e.g. 1000M, 2G) (Default: 1G).
  --proxy-user PROXY_USER
                        User to impersonate when submitting the application.
                        This argument does not work with --principal /
                        --keytab.
  -c CONF_PATHS, --conf-path CONF_PATHS
                        Path to alternate mrjob.conf file to read from
  --no-conf             Don't load mrjob.conf even if it's available
  -q, --quiet           Don't print anything to stderr
  -v, --verbose         print more messages to stderr
  -h, --help            show this message and exit
```

Spark and Hadoop runners only:

--master SPARK_MASTER spark://host:port, mesos://host:port, yarn,k8s://https://host:port, or local. Defaults to local[*] on spark runner, yarn on hadoop runner.

--deploy-mode SPARK_DEPLOY_MODE Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster") (Default: client).

Cluster deploy mode only:

--driver-cores DRIVER_CORES Number of cores used by the driver (Default: 1).

Spark standalone or Mesos with cluster deploy mode only:

--supervise If given, restarts the driver on failure.

Spark standalone and Mesos only:

--total-executor-cores TOTAL_EXECUTOR_CORES Total cores for all executors.

Spark standalone and YARN only:

--executor-cores EXECUTOR_CORES Number of cores per executor. (Default: 1 in YARN mode, or all available cores on the worker in standalone mode)

YARN-only:

--queue QUEUE_NAME The YARN queue to submit to (Default: "default").

--num-executors NUM_EXECUTORS Number of executors to launch (Default: 2). If dynamic allocation is enabled, the initial number of executors will be at least NUM.

--principal PRINCIPAL Principal to be used to login to KDC, while running onsecure HDFS.

--keytab KEYTAB The full path to the file that contains the keytab for the principal specified above. This keytab will be copied to the node running the Application Master via the Secure Distributed Cache, for renewing the login tickets and the delegation tokens periodically.

This also supports the same runner-specific switches as *MRJob*s (e.g. `--hadoop-bin`, `--region`).

2.3.8 terminate-cluster

Terminate an existing EMR cluster.

Usage:

```
mrjob terminate-cluster [options] CLUSTER_ID
```

Terminate an existing EMR cluster.

Options:

```
-c CONF_PATHS, --conf-path CONF_PATHS
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--ec2-endpoint EC2_ENDPOINT
                      Force mrjob to connect to EC2 on this endpoint (e.g.
                      ec2.us-west-1.amazonaws.com). Default is to infer this
                      from region.
--emr-endpoint EMR_ENDPOINT
                      Force mrjob to connect to EMR on this endpoint (e.g.
                      us-west-1.elasticmapreduce.amazonaws.com). Default is
                      to infer this from region.
-h, --help            show this help message and exit
-q, --quiet           Don't print anything to stderr
--region REGION       GCE/AWS region to run Dataproc/EMR jobs in.
--s3-endpoint S3_ENDPOINT
                      Force mrjob to connect to S3 on this endpoint (e.g.
                      s3-us-west-1.amazonaws.com). You usually shouldn't set
                      this; by default mrjob will choose the correct
                      endpoint for each S3 bucket based on its location.
-t, --test            Don't actually delete any files; just log that we
                      would
-v, --verbose         print more messages to stderr
```

2.3.9 terminate-idle-clusters

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

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

```
*/30 * * * * mrjob terminate-idle-clusters -q
```

Changed in version 0.6.4: Skips termination-protected idle clusters, rather than crashing. (This was also backported to mrjob v0.5.12.)

Options:

```
-c CONF_PATHS, --conf-path CONF_PATHS
                      Path to alternate mrjob.conf file to read from
--no-conf             Don't load mrjob.conf even if it's available
--dry-run             Don't actually kill idle jobs; just log that we would
--ec2-endpoint EC2_ENDPOINT
                      Force mrjob to connect to EC2 on this endpoint (e.g.
                      ec2.us-west-1.amazonaws.com). Default is to infer this
```

```
                        from region.
    --emr-endpoint EMR_ENDPOINT
                        Force mrjob to connect to EMR on this endpoint (e.g.
                        us-west-1.elasticmapreduce.amazonaws.com). Default is
                        to infer this from region.
    -h, --help          show this help message and exit
    --max-mins-idle MAX_MINS_IDLE
                        Max number of minutes a cluster 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.
    --max-mins-locked MAX_MINS_LOCKED
                        Deprecated, does nothing
    --pool-name POOL_NAME
                        Only terminate clusters in the given named pool.
    --pooled-only       Only terminate pooled clusters
    -q, --quiet         Don't print anything to stderr
    --region REGION     GCE/AWS region to run Dataproc/EMR jobs in.
    --s3-endpoint S3_ENDPOINT
                        Force mrjob to connect to S3 on this endpoint (e.g.
                        s3-us-west-1.amazonaws.com). You usually shouldn't set
                        this; by default mrjob will choose the correct
                        endpoint for each S3 bucket based on its location.
    --unpooled-only     Only terminate un-pooled clusters
    -v, --verbose       print more messages to stderr
```

2.4 mrjob.compat - Hadoop version compatibility

Utility functions for compatibility with different version of hadoop.

`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 2, `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 2.0, `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.`**`map_version`**(*version*, *version_map*)
Allows you to look up something by version (e.g. which jobconf variable to use, specifying only the versions where that value changed.

version is a string

version_map is a map from version (as a string) that a value changed to the new value.

For efficiency, *version_map* can also be a list of tuples of `(LooseVersion(version_as_string), value)`, with oldest versions first.

If *version* is less than any version in *version_map*, use the value for the earliest version in *version_map*.

`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.`**`translate_jobconf_dict`**(*jobconf*, *hadoop_version=None*)
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.`**`translate_jobconf_for_all_versions`**(*variable*)
Get all known variants of the given jobconf variable. Unlike *`translate_jobconf()`*, returns a list.

`mrjob.compat.`**`uses_yarn`**(*version*)
Basically, is this Hadoop 2? This also handles versions in the zero series (0.23+) where YARN originated.

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

2.5 mrjob.conf - parse and write config files

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

2.5.1 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, resolving includes. 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 real (according to `os.path.realpath()`) conf paths that have already been loaded (used by *load_opts_from_mrjob_confs()*).

Relative `include:` paths are relative to the real (after resolving symlinks) path of the including conf file

This will only load each config file once, even if it's referenced from multiple paths due to symlinks.

`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 (see *find_mrjob_conf()*).

Parameters

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

This will only load each config file once, even if it's referenced from multiple paths due to symlinks.

2.5.2 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_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_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_envs**(**envs*)

Combine zero or more dictionaries containing environment variables. Environment variable values may be wrapped in `ClearedValue`.

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. Wrapping a path value in `ClearedValue` disables this behavior.

Environment set to `ClearedValue(None)` will *delete* environment variables earlier in the list, rather than setting them to `None`.

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

`mrjob.conf.`**combine_jobconfs**(**jobconfs*)

Like combine_dicts(), but non-string values are converted to Java-readable string (e.g. True becomes 'true'). Keys whose value is None are blanked out.

`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.

Strings, bytes, and non-sequence objects (e.g. numbers) are treated as single-item lists.

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_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.

Can take single strings as well as lists.

mrjob.conf.**combine_paths**(**paths*)
Returns the last value in *paths* that is not `None`. Resolve ~ (home dir) and environment variables.

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).

2.6 mrjob.dataproc - run on Dataproc

2.6.1 Job Runner

class mrjob.dataproc.**DataprocJobRunner**(***kwargs*)
Runs an *MRJob* on Google Cloud Dataproc. Invoked when you run your job with `-r dataproc`.

DataprocJobRunner runs your job in an Dataproc cluster, which is basically a temporary Hadoop cluster.

Input, support, and jar files can be either local or on GCS; use `gs://...` URLs to refer to files on GCS.

This class has some useful utilities for talking directly to GCS and Dataproc, so you may find it useful to instantiate it without a script:

```
from mrjob.dataproc import DataprocJobRunner
...
```

2.6.2 GCS Utilities

class mrjob.dataproc.**GCSFilesystem**(*credentials=None*, *project_id=None*, *part_size=None*, *location=None*, *object_ttl_days=None*)
Filesystem for Google Cloud Storage (GCS) URIs

Parameters

- **credentials** – an optional `google.auth.credentials.Credentials`, used to initialize the storage client
- **project_id** – an optional project ID, used to initialize the storage client
- **part_size** – Part size for multi-part uploading, in bytes, or `None`
- **location** – Default location to use when creating a bucket
- **object_ttl_days** – Default object expiry for newly created buckets

Changed in version 0.7.0: removed *local_tmp_dir*

Changed in version 0.6.8: deprecated *local_tmp_dir*, added *part_size*, *location*, *object_ttl_days*

2.7 mrjob.emr - run on EMR

2.7.1 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 cluster, which is basically a temporary Hadoop cluster. Normally, it creates a cluster just for your job; it's also possible to run your job in a specific cluster by setting *cluster_id* or to automatically choose a waiting cluster, creating one if none exists, by setting *pool_clusters*.

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_client = EMRJobRunner().make_emr_client()
clusters = emr_client.list_clusters()
...
```

2.7.2 EMR Utilities

EMRJobRunner.**get_cluster_id**()

Get the ID of the cluster our job is running on, or `None`.

EMRJobRunner.**get_image_version**()

Get the version of the AMI that our cluster is running, or `None`.

EMRJobRunner.**get_job_steps**()

Fetch the steps submitted by this runner from the EMR API.

Deprecated since version 0.7.4.

New in version 0.6.1.

EMRJobRunner.**make_emr_client**()

Create a `boto3` EMR client.

Returns a `botocore.client.EMR` wrapped in a *mrjob.retry.RetryWrapper*

2.7.3 S3 Utilities

class mrjob.fs.s3.**S3Filesystem**(*aws_access_key_id=None*, *aws_secret_access_key=None*, *aws_session_token=None*, *s3_endpoint=None*, *s3_region=None*, *part_size=None*)

Filesystem for Amazon S3 URIs. Typically you will get one of these via `EMRJobRunner().fs`, composed with `SSHFilesystem` and `LocalFilesystem`.

Parameters

- **aws_access_key_id** – Your AWS access key ID
- **aws_secret_access_key** – Your AWS secret access key
- **aws_session_token** – session token for use with temporary AWS credentials
- **s3_endpoint** – If set, always use this endpoint

- **s3_region** – Default region for connections to the S3 API and newly created buckets.
- **part_size** – Part size for multi-part uploading, in bytes, or None

Changed in version 0.6.8: added *part_size*

S3Filesystem.**create_bucket**(*bucket_name*, *region=None*)
Create a bucket on S3 with a location constraint matching the given region.

S3Filesystem.**get_all_bucket_names**()
Get a list of the names of all buckets owned by this user on S3.

S3Filesystem.**get_bucket**(*bucket_name*)
Get the (boto3) bucket, connecting through the appropriate endpoint.

S3Filesystem.**make_s3_client**(*region_name=None*)
Create a boto3 S3 client, wrapped in a *mrjob.retry.RetryWrapper*

Parameters **region** – region to use to choose S3 endpoint.

S3Filesystem.**make_s3_resource**(*region_name=None*)
Create a boto3 S3 resource, with its client wrapped in a *mrjob.retry.RetryWrapper*

Parameters **region** – region to use to choose S3 endpoint

It's best to use *get_bucket()* because it chooses the appropriate S3 endpoint automatically. If you are trying to get bucket metadata, use *make_s3_client()*.

2.7.4 Other AWS clients

EMRJobRunner.**make_ec2_client**()
Create a boto3 EC2 client.

Returns a botocore.client.EC2 wrapped in a *mrjob.retry.RetryWrapper*

EMRJobRunner.**make_iam_client**()
Create a boto3 IAM client.

Returns a botocore.client.IAM wrapped in a *mrjob.retry.RetryWrapper*

2.8 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*.

2.8.1 Utilities

mrjob.hadoop.**fully_qualify_hdfs_path**(*path*)
If path isn't an hdfs:// URL, turn it into one.

2.9 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).

Unlike other runners, `InlineMRJobRunner`'s `run()` method raises the actual exception that caused a step to fail (rather than `StepFailedException`).

To more accurately simulate your environment prior to running on Hadoop/EMR, use `-r local` (see *LocalMRJobRunner*).

New in version 0.6.8: can run *SparkStep*s via the `pyspark` library.

InlineMRJobRunner.**__init__**(*mrjob_cls=None*, ***kwargs*)
InlineMRJobRunner takes the same keyword args as *MRJobRunner*. However, please note that *hadoop_input_format*, *hadoop_output_format*, 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`.

2.10 mrjob.job - defining your job

class mrjob.job.**MRJob**(*args=None*)
The base class for all MapReduce jobs. See *__init__()* for details.

2.10.1 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"'
```

MRJob.**mapper_raw**(*input_path*, *input_uri*)
Re-define this to make Hadoop pass one input file to each mapper.

Parameters

- **input_path** – a local path that the input file has been copied to
- **input_uri** – the URI of the input file on HDFS, S3, etc

New in version 0.6.3.

MRJob.**spark**(*input_path*, *output_path*)
Re-define this with Spark code to run. You can read input with *input_path* and output with *output_path*.

Warning: Prior to v0.6.8, to pass job methods into Spark (`rdd.flatMap(self.some_method)`), you first had to call *self.sandbox()*; otherwise Spark would error because *self* was not serializable.

2.10.2 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 [MRStep(mapper=self.transform_input,
                       reducer=self.consolidate_1),
                MRStep(reducer_init=self.log_mapper_init,
                       reducer=self.consolidate_2)]
```

Returns a list of steps constructed with *MRStep* or other classes in *mrjob.step*.

2.10.3 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:

•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 `sys.argv[1:]`

For a full list of command-line arguments, run: `python -m mrjob.job --help`

Parameters **args** – Arguments to your script (switches and input files)

Changed in version 0.7.0: Previously, *args* set to `None` was equivalent to `[]`.

`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*

2.10.4 Parsing output

`MRJob.parse_output`(*chunks*)

Parse the final output of this MRJob (as a stream of byte chunks) into a stream of `(key, value)`.

2.10.5 Counters and status messages

`MRJob.increment_counter`(*group*, *counter*, *amount=1*)

Increment a counter in Hadoop streaming by printing to stderr.

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.

2.10.6 Setting protocols

`MRJob.INPUT_PROTOCOL = <class 'mrjob.protocol.BytesValueProtocol'>`

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.StandardJSONProtocol'>**
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.StandardJSONProtocol'>**
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.

2.10.7 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

2.10.8 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_args**()
Define arguments for this script. Called from *__init__()*.

Re-define to define custom command-line arguments or pass through existing ones:

```
def configure_args(self):
    super(MRYourJob, self).configure_args()

    self.add_passthru_arg(...)
    self.add_file_arg(...)
    self.pass_arg_through(...)
    ...
```

MRJob.**add_passthru_arg**(**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 `argparse.ArgumentParser.add_argument()`:

```
def configure_args(self):
    super(MRYourJob, self).configure_args()
    self.add_passthru_arg(
        '--max-ngram-size', type=int, default=4, help='...')
```

If you want to pass files through to the mapper/reducer, use *add_file_arg()* instead.

If you want to pass through a built-in option (e.g. `--runner`, use *pass_arg_through()* instead.

MRJob.**add_file_arg**(**args*, ***kwargs*)
Add a command-line option that sends an external file (e.g. a SQLite DB) to Hadoop:

```
def configure_args(self):
   super(MRYourJob, self).configure_args()
   self.add_file_arg('--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.

Note: If you pass a file to a job, best practice is to lazy-load its contents (e.g. make a method that opens the file the first time you call it) rather than loading it in your job's constructor or *load_args()*. Not only is this more efficient, it's necessary if you want to run your job in a Spark executor (because the file may not be in the same place in a Spark driver).

Note: 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.

Changed in version 0.6.6: now accepts explicit `type=str`

Changed in version 0.6.8: fully supported on Spark, including `local[*]` master

`MRJob.`**`pass_arg_through`**(*opt_str*)
Pass the given argument through to the job.

`MRJob.`**`load_args`**(*args*)
Load command-line options into `self.options`.

Called from *`__init__()`* after *`configure_args()`*.

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_args(self, args):
    super(MRYourJob, self).load_args(args)

    self.stop_words = self.options.stop_words.split(',')
    ...
```

`MRJob.`**`is_task`**()
True if this is a mapper, combiner, reducer, or Spark script.

This is mostly useful inside *`load_args()`*, to disable loading args when we aren't running inside Hadoop.

2.10.9 Uploading support files

`MRJob.`**`FILES = []`**
Optional list of files to upload to the job's working directory. These can be URIs or paths on the local filesystem.

Relative paths will be interpreted as relative to the directory containing the script (not the current working directory). Environment variables and ~ in paths will be expanded.

If you want a file to be uploaded to a filename other than it's own, append `#<name>` (e.g. `data/foo.json#bar.json`).

If you need to dynamically generate a list of files, override *`files()`* instead.

New in version 0.6.4.

`MRJob.`**`DIRS = []`**
Optional list of directories to upload to the job's working directory. These can be URIs or paths on the local filesystem.

Relative paths will be interpreted as relative to the directory containing the script (not the current working directory). Environment variables and ~ in paths will be expanded.

If you want a directory to be copied with a name other than it's own, append `#<name>` (e.g. `data/foo#bar`).

If you need to dynamically generate a list of files, override *`dirs()`* instead.

New in version 0.6.4.

`MRJob.`**`ARCHIVES = []`**
Optional list of archives to upload and unpack in the job's working directory. These can be URIs or paths on the local filesystem.

Relative paths will be interpreted as relative to the directory containing the script (not the current working directory). Environment variables and ~ in paths will be expanded.

By default, the directory will have the same name as the archive (e.g. `foo.tar.gz/`). To change the directory's name, append `#<name>`:

```
ARCHIVES = ['data/foo.tar.gz#foo']
```

If you need to dynamically generate a list of files, override *archives()* instead.

New in version 0.6.4.

`MRJob.`**`files`**`()`
Like *FILES*, except that it can return a dynamically generated list of files to upload. Overriding this method disables *FILES*.

Paths returned by this method are relative to the working directory (not the script). Note that the job runner will *always* expand environment variables and ~ in paths returned by this method.

You do not have to worry about inadvertently disabling `--files`; this switch is handled separately.

New in version 0.6.4.

`MRJob.`**`dirs`**`()`
Like *DIRS*, except that it can return a dynamically generated list of directories to upload. Overriding this method disables *DIRS*.

Paths returned by this method are relative to the working directory (not the script). Note that the job runner will *always* expand environment variables and ~ in paths returned by this method.

You do not have to worry about inadvertently disabling `--dirs`; this switch is handled separately.

New in version 0.6.4.

`MRJob.`**`archives`**`()`
Like *ARCHIVES*, except that it can return a dynamically generated list of archives to upload and unpack. Overriding this method disables *ARCHIVES*.

Paths returned by this method are relative to the working directory (not the script). Note that the job runner will *always* expand environment variables and ~ in paths returned by this method.

You do not have to worry about inadvertently disabling `--archives`; this switch is handled separately.

New in version 0.6.4.

2.10.10 Job runner configuration

classmethod `MRJob.`**`mr_job_script`**`()`
Path of this script. This returns the file containing this class, or `None` if there isn't any (e.g. it was defined from the command line interface.)

2.10.11 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)

Called from *run()*. You'd probably only want to call this directly from automated tests.

MRJob.**map_pairs**(*pairs*, *step_num=0*)
Runs *mapper_init()*, *mapper()*/*mapper_raw()*, and *mapper_final()* for one map task in one step.

Takes in a sequence of (key, value) pairs as input, and yields (key, value) pairs as output.

run_mapper() essentially wraps this method with code to handle reading/decoding input and writing/encoding output.

New in version 0.6.7.

MRJob.**run_reducer**(*step_num=0*)
Run the reducer for the given step.

Parameters **step_num** (*int*) – which step to run (0-indexed)

Called from *run()*. You'd probably only want to call this directly from automated tests.

MRJob.**reduce_pairs**(*pairs*, *step_num=0*)
Runs *reducer_init()*, *reducer()*, and *reducer_final()* for one reduce task in one step.

Takes in a sequence of (key, value) pairs as input, and yields (key, value) pairs as output.

run_reducer() essentially wraps this method with code to handle reading/decoding input and writing/encoding output.

New in version 0.6.7.

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.**combine_pairs**(*pairs*, *step_num=0*)
Runs *combiner_init()*, *combiner()*, and *combiner_final()* for one reduce task in one step.

Takes in a sequence of (key, value) pairs as input, and yields (key, value) pairs as output.

run_combiner() essentially wraps this method with code to handle reading/decoding input and writing/encoding output.

New in version 0.6.7.

2.10.12 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**()
`-D` args to pass to hadoop streaming. This should be a map from property name to value. By default, returns *JOBCONF*.

Changed in version 0.6.6: re-defining longer clobbers command-line `--jobconf` options.

MRJob.**LIBJARS = []**
Optional list of paths of jar files to run our job with using Hadoop's `-libjars` option.

~ and environment variables in paths be expanded, and relative paths will be interpreted as relative to the directory containing the script (not the current working directory).

If you require more sophisticated behavior, try overriding *libjars()*.

MRJob.**libjars**()
Optional list of paths of jar files to run our job with using Hadoop's `-libjars` option. Normally setting *LIBJARS* is sufficient. Paths from *LIBJARS* are interpreted as relative to the the directory containing the script (paths from the command-line are relative to the current working directory).

Note that ~ and environment variables in paths will always be expanded by the job runner (see libjars).

Changed in version 0.6.6: re-defining this no longer clobbers the command-line `--libjars` option

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 *PARTITIONER*.

You probably don't need to re-define this; it's just here for completeness.

2.10.13 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 `BytesIO` 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=BytesIO(b'some_data\n'))
```

or, equivalently:

```
mr_job.sandbox(stdin=[b'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 BytesIO import BytesIO

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=BytesIO(fake_input))

mr_job.run_reducer(link_num=0)

self.assertEqual(mrjob.stdout.getvalue(), ...)
self.assertEqual(parse_mr_job_stderr(mr_job.stderr), ...)
```

Note: If you are using Spark, it's recommended you only pass in `io.BytesIO` or other serializable alternatives to file objects. *stdin*, *stdout*, and *stderr* get stored as job attributes, which means if they aren't serializable, neither is the job instance or its methods.

2.11 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.

It's rare to need to instantiate this class directly (see *__init__()* for details).

New in version 0.6.8: can run Spark steps as well, on the `local-cluster` Spark master.

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_input_format*, *hadoop_output_format*, 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`.

2.12 mrjob.parse - log parsing

Utilities for parsing errors, counters, and status messages.

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 a URI and contains `://` (we only care about URIs that can describe files)

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 (bytes), or bytes
- **counters** – Counters so far, to update; a map from group (string to counter name (string) 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 (strings) that aren't either counters or status messages

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

mrjob.parse.**to_uri**(*path_or_uri*)

If *path_or_uri* is not a URI already, convert it to a `file:///` URI.

2.13 mrjob.protocol - input and output

Protocols translate raw bytes into key, value pairs.

Typically, protocols encode a key and value into bytes, and join them together with a tab character.

However, protocols with `Value` in their name ignore keys and simply read/write values (with key read in as `None`), allowing you to read and write data in arbitrary formats.

For more information, see *Protocols* and *Writing custom protocols*.

2.13.1 Strings

class `mrjob.protocol.`**RawValueProtocol**
Just output `value` (a `str`), and discard `key` (`key` is read in as `None`).

This is the default protocol used by jobs to read input.

This is an alias for *RawValueProtocol* on Python 2 and *TextValueProtocol* on Python 3.

class `mrjob.protocol.`**BytesValueProtocol**
Read line (without trailing newline) directly into `value` (`key` is always `None`). Output `value` (bytes) directly, discarding `key`.

This is the default protocol used by jobs to read input on Python 2.

> **Warning:** Typical usage on Python 2 is to have your mapper parse (byte) strings out of your input files, and then include them in the output to the reducer. Since this output is then (by default) JSON-encoded, encoding will fail if the bytestrings are not UTF-8 decodable. If this is an issue, consider using *TextValueProtocol* instead.

class `mrjob.protocol.`**TextValueProtocol**
Attempt to UTF-8 decode line (without trailing newline) into `value`, falling back to latin-1. (`key` is always `None`). Output `value` UTF-8 encoded, discarding `key`.

This is the default protocol used by jobs to read input on Python 3.

This is a good solution for reading text files which are mostly ASCII but may have some other bytes of unknown encoding (e.g. logs).

If you wish to enforce a particular encoding, use *BytesValueProtocol* instead:

```
class MREncodingEnforcer(MRJob):

    INPUT_PROTOCOL = BytesValueProtocol

    def mapper(self, _, value):
        value = value.decode('utf_8')
        ...
```

class `mrjob.protocol.`**RawProtocol**
Output `key` (`str`) and `value` (`str`), separated by a tab character.

This is an alias for *BytesProtocol* on Python 2 and *TextProtocol* on Python 3.

class `mrjob.protocol.`**BytesProtocol**
Encode `(key, value)` (bytestrings) as `key` and `value` separated by a tab.

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.`**TextProtocol**
UTF-8 encode `key` and `value` (unicode strings) and join them with a tab character. When reading input, we fall back to latin-1 if we can't UTF-8 decode the line.

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.

2.13.2 JSON

class `mrjob.protocol.`**`JSONProtocol`**
Encode `(key, value)` as two JSONs separated by a tab.

This is the default protocol used by jobs to write output and communicate between steps.

This is an alias for the first one of *`UltraJSONProtocol`*, *`RapidJSONProtocol`*, *`SimpleJSONProtocol`*, or *`StandardJSONProtocol`* for which the underlying library is available.

class `mrjob.protocol.`**`UltraJSONProtocol`**
Implements *`JSONProtocol`* using the `ujson` library.

> **Warning:** `ujson` is about five times faster than the standard implementation, but is more willing to encode things that aren't strictly JSON-encodable, including sets, dictionaries with tuples as keys, UTF-8 encoded bytes, and objects (!). Relying on this behavior won't stop your job from working, but it can make your job *dependent* on `ujson`, rather than just using it as a speedup.

Note: `ujson` also differs from the standard implementation in that it doesn't add spaces to its JSONs (`{"foo":"bar"}` versus `{"foo": "bar"}`). This probably won't affect anything but test cases and readability.

---

class `mrjob.protocol.`**`RapidJSONProtocol`**
Implements *`JSONProtocol`* using the `rapidjson` library.

class `mrjob.protocol.`**`SimpleJSONProtocol`**
Implements *`JSONProtocol`* using the `simplejson` library.

class `mrjob.protocol.`**`StandardJSONProtocol`**
Implements *`JSONProtocol`* using Python's built-in JSON library.

Note: The built-in `json` library is (appropriately) strict about the JSON standard; it won't accept dictionaries with non-string keys, sets, or (on Python 3) bytestrings.

---

class `mrjob.protocol.`**`JSONValueProtocol`**
Encode `value` as a JSON and discard `key` (`key` is read in as `None`).

This is an alias for the first one of *`UltraJSONValueProtocol`*, *`RapidJSONValueProtocol`*, *`SimpleJSONValueProtocol`*, or *`StandardJSONValueProtocol`* for which the underlying library is available.

class `mrjob.protocol.`**`UltraJSONValueProtocol`**
Implements *`JSONValueProtocol`* using the `ujson` library.

class `mrjob.protocol.`**`RapidJSONValueProtocol`**
Implements *`JSONValueProtocol`* using the `rapidjson` library.

class `mrjob.protocol.`**`SimpleJSONValueProtocol`**
Implements *`JSONValueProtocol`* using the `simplejson` library.

class `mrjob.protocol.`**`StandardJSONValueProtocol`**
Implements *`JSONValueProtocol`* using Python's built-in JSON library.

2.13.3 Repr

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()*).

> **Warning:** The repr format changes between different versions of Python (for example, braces for sets in Python 2.7, and different string contants in Python 3). Plan accordingly.

class mrjob.protocol.**ReprValueProtocol**
Encode `value` as a repr and discard `key` (`key` is read in as None).

See *ReprProtocol* for details.

2.13.4 Pickle

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.

> **Warning:** Pickling is only *backwards*-compatible across Python versions. If your job uses this as an output protocol, you should use at least the same version of Python to parse the job's output. Vice versa for using this as an input protocol.

class mrjob.protocol.**PickleValueProtocol**
Encode `value` as a string-escaped pickle and discard `key` (`key` is read in as `None`).

See *PickleProtocol* for details.

2.14 mrjob.spark.runner - run on any Spark cluster

2.14.1 Job Runner

class mrjob.spark.runner.**SparkMRJobRunner**(*max_output_files=None*, *mrjob_cls=None*, ***kwargs*)
Runs a *MRJob* on your Spark cluster (with or without Hadoop). Invoked when you run your job with `-r spark`.

See *Running on your Spark cluster* for more information.

The Spark runner can also run "classic" MRJobs directly on Spark, without using Hadoop streaming. See *Running "classic" MRJobs on Spark*.

New in version 0.6.8.

2.15 mrjob.retry - retry on transient errors

class mrjob.retry.**RetryWrapper**(*wrapped*, *retry_if*, *backoff=15*, *multiplier=1.5*, *max_tries=10*, *max_backoff=1200*, *unwrap_methods=()*)

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*, *max_backoff=1200*, *unwrap_methods=()*)

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
- **max_backoff** (*float*) – cap the backoff at this number of seconds
- **unwrap_methods** (*sequence*) – names of methods to call with this object as *self* rather than retrying on transient errors (e.g. methods that return a paginator)

2.16 mrjob.runner - base class for all runners

class mrjob.runner.**MRJobRunner**(*mr_job_script=None*, *conf_paths=None*, *extra_args=None*, *hadoop_input_format=None*, *hadoop_output_format=None*, *input_paths=None*, *output_dir=None*, *partitioner=None*, *sort_values=None*, *stdin=None*, *steps=None*, *step_output_dir=None*, ***opts*)

Abstract base class for all runners

MRJobRunner.**__init__**(*mr_job_script=None*, *conf_paths=None*, *extra_args=None*, *hadoop_input_format=None*, *hadoop_output_format=None*, *input_paths=None*, *output_dir=None*, *partitioner=None*, *sort_values=None*, *stdin=None*, *steps=None*, *step_output_dir=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_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.
- **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 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 partitioner 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.
- **sort_values** (*bool*) – if true, set partitioners and jobconf variables so that reducers to receive the values associated with any key in sorted order (sorted by their *encoded* value). Also known as secondary sort.
- **stdin** – an iterable (can be a `BytesIO` 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.
- **steps** – a list of descriptions of steps to run (see mrjob.step - represent Job Steps for description formats)
- **step_output_dir** (*str*) – An empty/non-existent directory where Hadoop should put output from all steps other than the last one (this only matters for multi-step jobs). Currently ignored by local runners.

2.16.1 Running your job

`MRJobRunner.`**`run`**`()`

Run the job, and block until it finishes.

Raise `StepFailedException` if there are any problems (except on *`InlineMRJobRunner`*, where we raise the actual exception that caused the step to fail).

`MRJobRunner.`**`cat_output`**`()`

Stream the job's output, as a stream of `bytes`. If there are multiple output files, there will be an empty bytestring (`b''`) between them.

Like Hadoop input formats, we ignore files and subdirectories whose names start with `"_"` or `"."` (e.g. `_SUCCESS`, `_logs/`, `.part-00000.crc`.

Changed in version 0.6.8: Ignore file/dirnames starting with `"."` as well as `"_"`.

`MRJobRunner.`**`cleanup`**`(`*mode=None*`)`

Clean up running jobs, temp files, 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.options.**CLEANUP_CHOICES = ['ALL', 'CLOUD_TMP', 'CLUSTER', 'HADOOP_TMP', 'JOB', 'LOCAL_TMP',**
cleanup options:

- `'ALL'`: delete logs and local and remote temp files; stop cluster if on EMR and the job is not done when cleanup is run.
- `'CLOUD_TMP'`: delete temp files on cloud storage (e.g. S3) only
- **`'CLUSTER'`: terminate the cluster if on EMR and the job is not done** on cleanup
- `'HADOOP_TMP'`: delete temp files on HDFS only
- `'JOB'`: stop job if on EMR and the job is not done when cleanup runs
- `'LOCAL_TMP'`: delete local temp files only
- `'LOGS'`: delete logs only
- `'NONE'`: delete nothing
- `'TMP'`: delete local, HDFS, and cloud storage temp files, but not logs

2.16.2 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.

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 cluster. *HadoopJobRunner* gets this from `hadoop version`. *LocalMRJobRunner* has an additional *hadoop_version* option to specify which version it simulates. *InlineMRJobRunner* does not simulate Hadoop at all.

MRJobRunner.**get_job_key**()
Get the unique key for the job run by this runner. This has the format `label.owner.date.time.microseconds`

2.16.3 Configuration

MRJobRunner.**get_opts**()
Get options set for this runner, as a dict.

2.16.4 File management

`MRJobRunner.`**`fs`**
Filesystem object for the local filesystem.

class `mrjob.fs.base.`**`Filesystem`**
Some simple filesystem operations that are common across the local filesystem, S3, GCS, HDFS, and remote machines via SSH.

Different runners provide functionality for different filesystems via their *fs* attribute. Generally a runner will wrap one or more filesystems with `mrjob.fs.composite.CompositeFilesystem`.

Schemes supported:

- `mrjob.fs.gcs.GCSFilesystem`: `gs://`
- `mrjob.fs.hadoop.HadoopFilesystem`: `hdfs://` and other URIs
- `mrjob.fs.local.LocalFilesystem`: paths and `file://` URIs
- *mrjob.fs.s3.S3Filesystem*: `s3://`, `s3a://`, `s3n://`,
- `mrjob.fs.ssh.SSHFilesystem`: `ssh://`

Changed in version 0.6.12: *LocalFilesystem* added support for `file://` URIs

`can_handle_path`(*path*)
Can we handle this path at all?

`cat`(*path_glob*)
cat all files matching **path_glob**, decompressing if necessary

This yields bytes, which don't necessarily correspond to lines (see #1544). If multiple files are catted, yields `b''` between each file.

`du`(*path_glob*)
Get the total size of files matching `path_glob`

Corresponds roughly to: `hadoop fs -du path_glob`

`exists`(*path_glob*)
Does the given path/URI exist?

Corresponds roughly to: `hadoop fs -test -e path_glob`

`join`(*path*, **paths*)
Join *paths* onto *path* (which may be a URI)

`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 -ls -R path_glob`

`md5sum`(*path*)
Generate the md5 sum of the file at *path*

`mkdir`(*path*)
Create the given dir and its subdirs (if they don't already exist). On cloud filesystems (e.g. S3), also create the corresponding bucket as needed

Corresponds roughly to: `hadoop fs -mkdir -p path`

New in version 0.6.8: creates buckets on cloud filesystems

put (*src*, *path*)
Upload a file on the local filesystem (*src*) to *path*. Like with `shutil.copyfile()`, *path* should be the full path of the new file, not a directory which should contain it.

Corresponds roughly to `hadoop fs -put src path`.

New in version 0.6.8.

rm (*path_glob*)
Recursively delete the given file/directory, if it exists

Corresponds roughly to: `hadoop fs -rm -R 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`

2.17 mrjob.step - represent Job Steps

Representations of job steps, to use in your *MRJob*'s *steps()* method.

Because *the runner* just needs to know how to invoke your MRJob script, not how it works insternally, each step instance's `description()` method produces a simplified, JSON-able description of the step, to pass to the runner.

2.17.1 Steps

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.

Takes the following keyword arguments: *combiner*, *combiner_cmd*, *combiner_final*, *combiner_init*, *combiner_pre_filter*, *mapper*, *mapper_cmd*, *mapper_final*, *mapper_init*, *mapper_pre_filter*, *mapper_raw*, *reducer*, *reducer_cmd*, *reducer_final*, *reducer_init*, *reducer_pre_filter*. These should be set to `None` or a function with the same signature as the corresponding method in *MRJob*.

Also accepts *jobconf*, a dictionary with custom jobconf arguments to pass to hadoop.

A MRStep's description looks like:

```
{
    'type': 'streaming',
    'mapper': { ... },
    'combiner': { ... },
    'reducer': { ... },
    'jobconf': { ... },  # dict of Hadoop configuration properties
}
```

At least one of `mapper`, `combiner` and `reducer` need be included. `jobconf` is completely optional.

`mapper`, `combiner`, and `reducer` are either handled by the script containing your job definition, in which case they look like:

```
{
    'type': 'script',
    'pre_filter': 'grep -v bad', # optional cmd to filter input
}
```

or they simply run a command, which looks like:

```
{
    'type': 'command',
    'command': 'cut -f 1-2', # command to run, as a string
}
```

class mrjob.step.**JarStep**(*jar*, ***kwargs*)
Represents a running a custom Jar as a step.

Accepts the following keyword arguments:

Parameters

- **jar** – The local path to the Jar. On EMR, this can also be an `s3://` URI, or `file://` to reference a jar on the local filesystem of your EMR instance(s).
- **args** – (optional) A list of arguments to the jar. Use *mrjob.step.INPUT* and *OUTPUT* to interpolate input and output paths.
- **jobconf** – (optional) A dictionary of Hadoop properties
- **main_class** – (optional) The main class to run from the jar. If not specified, Hadoop will use the main class in the jar's manifest file.

jar can also be passed as a positional argument

See *Jar steps* for sample usage.

Sample description of a JarStep:

```
{
    'type': 'jar',
    'jar': 'binks.jar.jar',
    'main_class': 'MyMainMan',  # optional
    'args': ['argh', 'argh']  # optional
    'jobconf': { ... }  # optional
}
```

To give your jar access to input files, an empty output directory, configuration properties, and libjars managed by mrjob, you may include *INPUT*, *OUTPUT*, and *GENERIC_ARGS* in *args*.

class mrjob.step.**SparkStep**(*spark*, ***kwargs*)
Represents running a Spark step defined in your job.

Accepts the following keyword arguments:

Parameters

- **spark** – function containing your Spark code with same function signature as *spark()*
- **jobconf** – (optional) A dictionary of Hadoop properties
- **spark_args** – (optional) an array of arguments to pass to spark-submit (e.g. `['--executor-memory', '2G']`).

Sample description of a SparkStep:

```
{
    'type': 'spark',
    'jobconf': { ... },  # optional
    'spark_args': ['--executor-memory', '2G'],  # optional
}
```

class mrjob.step.**SparkJarStep**(*jar*, *main_class*, ***kwargs*)
Represents a running a separate Jar through Spark

Accepts the following keyword arguments:

Parameters

- **jar** – The local path to the Python script to run. On EMR, this can also be an `s3://` URI, or `file://` to reference a jar on the local filesystem of your EMR instance(s).
- **main_class** – Your application's main class (e.g. `'org.apache.spark.examples.SparkPi'`)
- **args** – (optional) A list of arguments to the script. Use *mrjob.step.INPUT* and *OUTPUT* to interpolate input and output paths.
- **jobconf** – (optional) A dictionary of Hadoop properties
- **spark_args** – (optional) an array of arguments to pass to spark-submit (e.g. `['--executor-memory', '2G']`).

jar and *main_class* can also be passed as positional arguments

Sample description of a SparkJarStep:

```
{
    'type': 'spark_jar',
    'jar': 'binks.jar.jar',
    'main_class': 'MyMainMan',  # optional
    'args': ['argh', 'argh'],  # optional
    'jobconf': { ... },  # optional
    'spark_args': ['--executor-memory', '2G'],  # optional
}
```

To give your Spark JAR access to input files and an empty output directory managed by mrjob, you may include *INPUT* and *OUTPUT* in *args*.

class mrjob.step.**SparkScriptStep**(*script*, ***kwargs*)
Represents a running a separate Python script through Spark

Accepts the following keyword arguments:

Parameters

- **script** – The local path to the Python script to run. On EMR, this can also be an `s3://` URI, or `file://` to reference a jar on the local filesystem of your EMR instance(s).
- **args** – (optional) A list of arguments to the script. Use *mrjob.step.INPUT* and *OUTPUT* to interpolate input and output paths.
- **jobconf** – (optional) A dictionary of Hadoop properties
- **spark_args** – (optional) an array of arguments to pass to spark-submit (e.g. `['--executor-memory', '2G']`).

script can also be passed as a positional argument

Sample description of a ScriptStep:

```
{
     'type': 'spark_script',
     'script': 'my_spark_script.py',
     'args': ['script_arg1', 'script_arg2'],
     'jobconf': { ... },  # optional
```

```
        'spark_args': ['--executor-memory', '2G'],  # optional
    }
```

To give your Spark script access to input files and an empty output directory managed by mrjob, you may include *INPUT* and *OUTPUT* in *args*.

2.17.2 Argument interpolation

Use these constants in your step's *args* and mrjob will automatically replace them before running your step.

`mrjob.step.`**`INPUT`** **= '<input>'**
If passed as an argument to *JarStep*, *SparkJarStep*, or *SparkScriptStep*, it'll be replaced with the step's input path(s). If there are multiple paths, they'll be joined with commas.

`mrjob.step.`**`OUTPUT`** **= '<output>'**
If this is passed as an argument to *JarStep*, *SparkJarStep*, or *SparkScriptStep*, it'll be replaced with the step's output path

`mrjob.step.`**`GENERIC_ARGS`** **= '<generic args>'**
If this is passed as an argument to *JarStep*, it'll be replaced with generic hadoop args (-D and -libjars)

2.18 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.`**`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** (*archive_file_suffix=''*)
Represents the working directory of hadoop/Spark tasks (or bootstrap commands in the cloud).

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.

if *type* is archive, we'll also add *path* as an auto-named archive_file. This reserves space in the working dir in case we need to copy the archive into the working dir and un-archive it ourselves.

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=None*)
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 (*type=None*)
Get a set of all paths tracked by this WorkingDirManager.

mrjob.setup.**name_uniquely** (*path*, *names_taken=()*, *proposed_name=None*, *unhide=False*, *strip_ext=False*, *suffix=''*)
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.
- **unhide** – make sure final name doesn't start with periods or underscores

- **strip_ext** – if we propose a name, it shouldn't have a file extension
- **suffix** – if set to a string, add this to the end of any filename we propose. Should include the `.`

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.txt', {'foo.txt'})
'foo-1.txt'
>>> name_uniquely('bar.tar.gz', {'bar'}, strip_ext=True)
'bar-1'
```

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.

If *path* is followed by a trailing slash, that indicates *path* is a directory and should be tarballed and later unarchived into a directory on the remote system. The trailing slash will also be kept as part of the original command. You may optionally include a slash after *name* as well (this will only result in a single slash in the final command).

Parsed hash paths are dicitionaries with the keys `path`, `name`, and `type` (either `'file'`, `'archive'`, or `'dir'`).

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 quotes or whitespace
- *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.

2.19 mrjob.util - general utility functions

Utility functions for MRJob

`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.`**`file_ext`**(*filename*)
return the file extension, including the `.`

```
>>> file_ext('foo.tar.gz')
'.tar.gz'
```

```
>>> file_ext('.emacs')
''
```

```
>>> file_ext('.mrjob.conf')
'.conf'
```

`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
- **`stream`** (*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.`**`random_identifier`**()
A random 16-digit hex string.

`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.**save_sys_path**(**args*, ***kwds*)
Context manager that saves sys.path and restores it after execution.

mrjob.util.**save_sys_std**(**args*, ***kwds*)
Context manager that saves the current values of *sys.stdin*, *sys.stdout*, and *sys.stderr*, and flushes these filehandles before and after switching them out.

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.**to_lines**(*chunks*)
Take in data as a sequence of bytes, and yield it, one line at a time.

Only breaks lines on `\n` (not `\r`), and does not add a trailing newline.

For efficiency, passes through anything with a `readline()` attribute.

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.

mrjob.util.**unique**(*items*)
Yield items from *item* in order, skipping duplicates.

mrjob.util.**which**(*cmd*, *path=None*)
Like the UNIX which command: search in *path* for the executable named *cmd*. *path* defaults to `PATH`. Returns `None` if no such executable found.

This is basically `shutil.which()` (which was introduced in Python 3.3) without the *mode* argument. Best practice is to always specify *path* as a keyword argument.

mrjob.util.**zip_dir**(*dir*, *out_path*, *filter=None*, *prefix=''*)
Compress the given *dir* into a zip file 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'`)

CHAPTER 3

What's New

For a complete list of changes, see CHANGES.txt

3.1 0.7.4

3.1.1 Docker on EMR

This release adds support for Docker on EMR, which was released with AMI version 6.0.0. This is enabled by setting docker_image to point at your image.

There is also a docker_mounts option, and, if you want to host your image on a private ECR repo instead of Docker Hub, a docker_client_config option (though with AMIs 6.1.0 and later, you can also auto-authenticate to ECR; see this page).

As a result of adding Docker support, the default image_version on EMR is 6.0.0. Also, on EMR and Dataproc we used to literally bootstrap mrjob by copying it to Python's root package directory, but as this won't put mrjob into a Docker image, mrjob is now bootstrapped via py_files, like on every other runner.

3.1.2 Concurrent Steps on EMR clusters

This release also supports concurrent steps on EMR clusters, a feature introduced in AMI 5.28.0. The max_concurrent_steps option controls both the concurrency level of a newly launched cluster, and how much concurrency we will accept when joining a pooled cluster.

To prevent steps from the same job attempting to run simultaneously, mrjob will now submit steps of a multi-step one at a time (after the previous one completes) on clusters running AMI 5.28.0 or later. This can be changed with the add_steps_in_batch option.

`get_job_steps()` is now deprecated, as it can't fetch steps before they're submitted.

3.1.3 Cluster Pooling

Cluster pooling can now join pooled clusters based on available CPU and memory reported by the YARN resource manager, rather than looking at number and type of instances in the cluster. You can enable this by setting min_available_mb and/or min_available_virtual_cores. For this feature to work, you must enable SSH (the ec2_key_pair and ec2_key_pair_file options).

You can now control the size of your cluster pool with the max_clusters_in_pool option. If a job wants to launch a new cluster in the pool but the pool is already "full," it will wait and try again until the pool is no longer full or it can join a cluster.

Once a job determines that it is okay to add another cluster to the pool, it will wait a random number of seconds and try again. This way, if several pooled jobs launch simultaneously, they will be likely to stay within the maximum number of clusters rather than all launching their own. The random wait time can be controlled with pool_jitter_seconds.

By default, a job will wait forever to either join an existing cluster or create new one. You can make jobs give up and raise an exception with the pool_timeout_minutes option.

mrjob will now bypass the pool_wait_minutes option if there is not a matching, active cluster to join. Basically, it won't wait if there is not a cluster to wait for. As with max_clusters_in_pool, if a job determines there are no clusters to wait for, it will wait a random number of seconds and double-check before launching a new cluster.

3.1.4 Library requirements

To support concurrent steps, `boto3` must be at least version 1.10.0 and `botocore` must be at least version 1.13.16. The `google-cloud-dataproc` library must be no greater than 1.1.0, to maintain compatibility with our code.

3.2 0.7.3

Made many long-overdue changes to *Cluster Pooling*, to reduce the potential for throttling by the EMR API. Pooling now puts most information a job needs to tell if it can join a cluster into the cluster name, meaning most non-matching clusters can be filtered out when we call `ListClusters`. Pooling also no longer needs to list cluster steps. Finally, if pool_wait_minutes is set, and there are multiple clusters we can join, we try them all, rather than just trying the "best" one and then requesting more information from the API.

This update resulted in a few minor changes to pooling. When a job has the choice of multiple clusters, it chooses solely on based on CPU capacity, using `NormalizedInstanceHours` in the cluster summary returned by the `ListClusters` API call. mrjob version and applications must now match exactly in all cases.

We also re-worked the "locking" mechanism that keeps multiple jobs from joining the same cluster. Formerly, this used S3 (which may only be eventually consistent), and locks had no fixed expiration time. Now, EMR tags are used for locking, locks always expire after one minute, and every job uses the same timing when locking clusters, reducing the potential for race conditions.

`mrjob terminate-idle-clusters` no longer attempts to lock clusters before terminating them, so its `--max-mins-locked` option is deprecated and does nothing.

The Spark harness now emulates counters correctly in local mode.

If you use *`mapper_raw()`*, and your setup script has an error, it will be correctly reported, even if your underlying shell is **`dash`** and not **`bash`**.

3.3 0.7.2

Spark normally only supports archives if you're running on YARN. However, mrjob now seamlessly emulates archives on all Spark masters (other than `local`). This means you can now use `--archives` or `--dirs` with `mrjob spark-submit`, as well as using archives in your `--setup` script.

As a result of this change, mrjob is somewhat better at recognizing file extensions; it ignores `.` at the end of filenames, and can now recognize that a file with a name like `mrjob-0.7.0.tar.gz` is a `.tar.gz` file, not a `.7.0.tar.gz` file.

Also, if you don't specify a name for an archive (e.g. `--setup 'cd foo.tar.gz#/'`) mrjob no longer includes the file extension in the resulting directory name (`foo/`, not `foo.tar.gz/`).

Patched a long-standing security issue on EMR where we were copying the SSH key to the master node when reading logs from other nodes, which are only accessible via the master node. mrjob now correctly uses **`ssh-add`** and the SSH agent instead of copying the key. As a result, mrjob now has a ssh_add_bin option.

The extra_cluster_params option now recursively merges dict params into existing ones. For example, you can now do this:

```
runners:
  emr:
    extra_cluster_params:
      Instances:
        EmrManagedMasterSecurityGroup: sg-foo
```

without obliterating the rest of the `Instances` API parameter.

Python 2 has reached end-of-life, so if you're using Python 2, the default python_bin is `python2.7` rather than `python`, which now means Python 3 on some systems (for example, 6.x EMR AMIs).

Finally, we ensure that if you're installing mrjob on Python 3.4, we'll install a Python 3.4-compatible version of PyYAML.

3.4 0.7.1

3.4.1 EMR

Fixed a bug to set default value of *VisibleToAllUsers* to *True*.

You can set sub-parameters with extra_cluster_params to set it *False*. For example, you can now do:

```
--extra-cluster-param VisibleToAllUsers=false
```

Added logging for mrjob to show invoked runner with keyword arguments. Contents of archives are now used during bootstrapping to ensure clusters have same setup.

3.5 0.7.0

3.5.1 AWS and Google are now optional dependencies

Amazon Web Services (EMR/S3) and Google Cloud are now optional dependencies, `aws` and `google` respectively. For example, to install mrjob with AWS support, run:

```
pip install mrjob[aws]
```

3.5.2 non-Python mrjobs are no longer supported

Fully removed support for writing MRJob scripts in other languages and then running them with the mrjob library. (This capability so little used that chances are you never knew it existed.)

As a result the *interpreter* and *steps_interpreter* options are gone, the **`mrjob run`** subcommand is gone, and the *MRJobLauncher* class has been merged back into *MRJob*. Also removed `mr_wc.rb` from `mrjob/examples/`

3.5.3 MRSomeJob() means read from sys.argv

In prior versions, if you initialized a *MRJob* subclass with no arguments (`MRSomeJob()`), that meant the same thing as passing in an empty argument list (`MRSomeJob(args=[])`). It now means to read *args* directly from `sys.argv[1:]`.

In practice, it's rare to see `MRJob` subclass intialized this way outside of test cases. Running a `MRJob` script directly, or initializing it with an argument list works this same as in previous versions.

3.5.4 mrjob/examples/ love

The mrjob.examples package has been updated. Some examples that were difficult to test or maintain were removed, and the remainder were tested and fixed if necessary.

`mrjob.examples.mr_text_classifier` no longer needs you to encode documents in JSON format, and instead operates directly on text files with names like `doc_id-cat_id_1-not_cat_id_2-etc.txt`. Try it out:

```
python -m mrjob.examples.mr_text_classifier docs-to-classify/*.txt
```

3.5.5 miscellanous tweaks

The **`mrjob audit-emr-usage`** subcommand no longer attempts to read cluster pool names from clusters launched by mrjob v0.5.x.

Method arguments in filesystem classes (in `mrjob.fs`) are now consistenly named. This probably won't matter in practice, as `runner.fs <mrjob.runner.MRJobRunner.fs>` is always a `CompositeFilesystem` anyhow.

3.5.6 removed deprecated code

Check your deprecation warnings! Everything marked deprecated in mrjob v0.6.x has been removed.

The following runner config options no longer exist: *emr_api_params*, *interpreter*, *max_hours_idle*, *mins_to_end_of_hour*, *steps_interpreter*, *steps_python_bin*, *visible_to_all_users*.

The following singular switches have been removed in favor of their plural alternative (e.g. **`--archives`**): **`--archive`**, **`--dir`**, **`--file`**, **`--hadoop-arg`**, **`--libjar`**, **`--py-file`**, **`--spark-arg`**.

The **`--steps`** switch is gone. This means **`--help --steps`** no longer works; use **`--help -v`** to see help for **`--mapper`**, etc.

Support for simulating `optparse` has been removed from *MRJob*. This includes `add_file_option()`, `add_passthrough_option()`, `configure_options()`, `load_options()`, `pass_through_option()`, `self.args`, `self.OPTION_CLASS`.

`mrjob.job.MRJobRunner.stream_output()` and `mrjob.job.MRJob.parse_output_line()` have been removed.

The constructor for `MRJobRunner` no longer has a *file_upload_args* keyword argument.

`parse_and_save_options()`, `read_file()`, and `read_input()` have all been removed from *mrjob.util*.

`CompositeFilesystem` no longer takes filesystems as arguments to its constructor; use `add_fs()`. The useless *local_tmp_dir* option to the `GCSFilesystem` constructor and the *chunk_size* arg to its `put()` method have been removed.

3.6 0.6.12

Updated the Dataproc's runner default image_version to `1.3`, as the old default, `1.0` no longer works.

The local and inline runners can now handle `file://` URIs as input paths and as files/archives uploaded to the working directory. The local filesystem (available as `runner.fs` from all runners) can now handle `file://` URIs as well.

3.7 0.6.11

Adds support for parsing Spark logs and driver output to determine why a job failed. This works with with the local, Hadoop, EMR, and Spark runners.

The Spark runner no longer needs `pyspark` in the `$PYTHONPATH` to launch scripts with **`spark-submit`** (it still needs `pyspark` to use the Spark harness).

On Python 3.7, you can now intermix positional arguments to *MRJob* with switches, similar to how you could back when mrjob used `optparse`. For example: **`mr_your_script.py file1 -v file2`**.

On EMR, the default image_version (AMI) is now 5.27.0.

Restored `m4.large` as the default instance type pre-5.13.0 AMIs, as they do not support `m5.xlarge`. (`m5.xlarge` is still the default for AMI 5.13.0 and later.)

mrjob can now retry on transient AWS API errors (e.g. throttling) or network errors when making API calls that use pagination (e.g. listing clusters).

The emr_configurations opt now supports the `!clear` tag rather than crashing. You may also override individual configs by setting a config with the same `Classification`.

This version restores official support for Python 3.4, as it's the version of Python 3 installed on EMR AMIs prior to 5.20.0. In order to make this work, mrjob drops support for Google Cloud services in Python 3.4, as the recent Google libraries appear to need a later Python version.

3.8 0.6.10

Adds official support for PyPy (that is any version of it compatible with Python 2.7/3.5+). If you launch a job in PyPy python_bin will automatically default to `pypy` or `pypy3` as appropriate.

Note that mrjob does not auto-install PyPy for you on EMR (Amazon Linux does not provide a PyPy package). Installing PyPy yourself at bootstrap time is fairly straightforward, see *Installing PyPy*.

The Spark harness can now be used on EMR, allowing you to run "classic" MRJobs in Spark, which is often faster. Essentially, you launch jobs in the Spark runner with `--spark-submit-bin 'mrjob spark-submit -r emr'`; see *Running classic MRJobs on Spark on EMR* for details.

The Spark runner can now optionally disable internal protocols when running "classic" MRJobs, eliminating the (usually) unnecessary effort of encoding data structures into JSON or other string representations and then decoding them. See skip_internal_protocol for details.

The EMR runner's default instance type is now `m5.xlarge`, which works with newer reasons and should make it easier to run Spark jobs. The EMR runner also now logs the DNS of the master node as soon as it is available, to make it easier to SSH in.

Finally, mrjob gives a much clearer error message if you attempt to read a YAML mrjob.conf file without `PyYAML` installed.

3.9 0.6.9

Drops support for Python 3.4.

Fixes a bug introduced in *0.6.8* that could break archives or directories uploaded into Hadoop or Spark if the name of the unpacked archive didn't have an archive extension (e.g. `.tar.gz`).

The Spark runner can now optionally emulate Hadoop's `mapreduce.map.input.file` configuration property when running the mapper of the first step of a streaming job if you enable emulate_map_input_file. This means that jobs that depend on *`jobconf_from_env('mapreduce.map.input.file')`* will still work.

The Spark runner also now uses the correct argument names when emulating *`increment_counter()`*, and logs a warning if spark_tmp_dir doesn't match spark_master.

mrjob spark-submit can now pass switches to the Spark script/JAR without explicitly separating them out with `--`.

The local and inline runners now more correctly emulate the *mapreduce.map.input.file* config property by making it a `file://` URL.

Deprecated methods `add_file_option()` and `add_passthrough_option()` can now take a type (e.g. `int`) as their `type` argument, to better emulate `optparse`.

3.10 0.6.8

3.10.1 Nearly full support for Spark

This release adds nearly full support for Spark, including mrjob-specific features like setup scripts and *passthrough options*. See *Why use mrjob with Spark?* for everything mrjob can do with Spark.

This release adds a *`SparkMRJobRunner`* (`-r spark`), which works with any Spark installation, does not require Hadoop, and can access any filesystem supported by both mrjob and Spark (HDFS, S3, GCS). The Spark runner is now the default for *mrjob spark-submit*.

What's *not* supported? mrjob does not yet support Spark on Google Cloud Dataproc. The Spark runner does not yet parse logs to determine probable cause of failure when your job fails (though it does give you the Spark driver output).

3.10.2 Spark Hadoop Streaming emulation

Not only does the Spark runner not need Hadoop to run Spark jobs, it doesn't need Hadoop to run most *Hadoop Streaming* jobs, as it knows how to run them directly on Spark. This means if you want to migrate to a non-Hadoop Spark cluster, you can take all your old *`MRJob`*s with you. See *Running "classic" MRJobs on Spark* for details.

The "experimental harness script" mentioned in *0.6.7* is now fully integrated into the Spark runner and is no longer supported as a separate feature.

3.10.3 Local runner support for Spark

The `local` and `inline` runner can now run Spark scripts locally for testing, analogous to the way they've supported Hadoop streaming scripts (except that they *do* require a local Spark installation). See *Other ways to run on Spark*.

3.10.4 Other Spark improvements

*MRJob*s are now Spark-serializable without calling *sandbox()* (there used to be a problematic reference to `sys.stdin`). This means you can always pass job methods to `rdd.flatMap()` etc.

setup scripts are no longer a YARN-specific feature, working on all Spark masters (except `local[*]`, which doesn't give executors a separate working directory).

Likewise, you can now specify a different name for files in the job's working directory (e.g. `--file foo#bar`) on all Spark masters.

Note: Uploading archives and directories still only works on YARN for now; Spark considers `--archives` a YARN-specific feature.

When running on a local Spark cluster, uses `file://...` rather than just the path of the file when necessary (e.g. with `--py-files`).

cat_output() now ignores files and subdirectories starting with `"."` (used to only be `"_"`). This allows mrjob to ignore Spark's checksum files (e.g. `.part-00000.crc`), and also brings mrjob in closer compliance to the way Hadoop input formats read directories.

`spark.yarn.appMasterEnv.*` config properties are only set if you're actually running on YARN.

The values of spark_master and spark_deploy_mode can no longer be overridden with configuration properties (`-D spark.master=...`). While not exactly a "feature," this means that mrjob always knows what Spark platform it's running on.

3.10.5 Filesystems

Every runner has an `fs` attribute that gives access to all the filesystems that runner supports.

Added a *put()* method to all filesystems, which allows uploading a single file (it used to be that each runner had custom logic for uploads).

It also used to be that if you wanted to create a bucket on S3 or GCS, you had to call `create_bucket(...)` explicitly. Now *mkdir()* will automatically create buckets as needed.

If you still need to access methods specific to a filesystem, you should do so through `fs.<name>`, where `<name>` is the (lowercase) name of the storage service. For example the Spark runner's filesystem offers both `runner.fs.s3.create_bucket()` and `runner.fs.gcs.create_bucket()`. The old style of implicitly passing through FS-specific methods (`runner.fs.create_bucket(...)`) is deprecated and going away in v0.7.0.

`GCSFilesystem`'s constructor had a useless `local_tmp_dir` argument, which is now deprecated and going away in v0.7.0.

3.10.6 EMR

Fixed a bad bug introduced in *0.6.7* that could prevent mrjob from running on EMR with a non-default temp bucket.

You can now set sub-parameters with extra_cluster_params. For example, you can now do:

```
--extra-cluster-param Instances.EmrManagedMasterSecurityGroup=...
```

without clobbering the zone or instance group/fleet configs specified in `Instances`.

Running your job with `--subnet ''` now un-sets a subnet specified in your config file (used to be ignored).

If you are using cluster pooling with retries (pool_wait_minutes), mrjob now retains information about clusters that is immutable (e.g. AMI version), saving API calls.

3.10.7 Dependency upgrades

Bumped the required versions of several Google Cloud Python libraries to be more compatible with current versions of their sub-dependencies (Google libraries pin a fairly narrow range of dependencies). `mrjob` now requires:

- `google-cloud-dataproc` at least 0.3.0,
- `google-cloud-logging` at least 1.9.0, and
- `google-cloud-storage` at least 1.13.1.

Also dropped support for `PyYAML` 3.08; now we require at least `PyYAML` 3.10 (which came out in 2011).

Note: We are aware that the Google libraries' extensive dependencies can be a nuisance for mrjob users who don't use Google Cloud. Our tentative plan is to make dependencies specific to a third-party service (including `google-cloud-*` and `boto3`) optional starting in v0.7.0.

3.10.8 Other bugfixes

Fixed a long-standing bug that would cause the Hadoop runner to hang or raise cryptic errors if hadoop_bin or spark_submit_bin is not executable.

Support files for `mrjob.examples` (e.g. `stop_words.txt` for `MRMostUsedWord`) are now installed along with `mrjob`.

Setting a **_bin* option to an empty value (e.g. `--hadoop-bin`) now always instructs mrjob to use the default, rather than disabling core features or creating cryptic errors. This affects gcloud_bin, hadoop_bin, sh_bin, and ssh_bin; the various **python_bin* options already worked this way.

3.11 0.6.7

setup commands now work on Spark (at least on YARN).

Added the *mrjob spark-submit* subcommand, which works as a drop-in replacement for **`spark-submit`** but with mrjob runners (e.g EMR) and mrjob features (e.g. setup, cmdenv).

Fixed a bug that was causing idle timeout scripts to silently fail on 2.x EMR AMIs.

Fixed a bug that broke `create_bucket()` on `us-east-1`, preventing new mrjob installations from launching on EMR in that region.

Fixed an `ImportError` from attempting to import `os.SIGKILL` on Windows.

The default instance type on EMR is now `m4.large`.

EMR's cluster pooling now knows the CPU and memory capacity of `c5` and `m5` instances, allowing it to join "better" clusters.

Added the plural form of several switches (separate multiple values with commas):

- `--applications`
- `--archives`
- `--dirs`

- `--files`
- `--libjars`
- `--py-files`

Except for `--application`, the singular version of these switches (`--archive`, `--dir`, `--file`, `--libjar`, `--py-file`) is deprecated for consistency with Hadoop and Spark

sh_bin is now fully qualified by default (`/bin/sh -ex`, not `sh -ex`). sh_bin may no longer be empty, and a warning is issued if it has more than one argument, to properly support shell script shebangs (e.g. `#!/bin/sh -ex`) on Linux.

Runners no longer call *MRJob*s with `--steps`; instead the job passes its step description to the runner on instantiation. `--steps` and *steps_python_bin* are now deprecated.

The Hadoop and EMR runner can now set `SPARK_PYTHON` and `SPARK_DRIVER_PYTHON` to different values if need be (e.g. to match task_python_bin, or to support setup scripts in client mode).

The inline runner no longer attempts to run command substeps.

The inline and local runner no longer silently pretend to run non-streaming steps.

The Hadoop runner no longer has the bootstrap_spark option, which did nothing.

interpreter and *steps_interpreter* are deprecated, in anticipation in removing support for writing MRJobs in other programming languages.

Runners now issue a warning if they receive options that belong to other runners (e.g. passing image_version to the Hadoop runner).

`mrjob create-cluster` now supports `--emr-action-on-failure`.

Updated deprecate escape sequences in mrjob code that would break on Python 3.8.

`--help` message for mrjob subcommands now correctly includes the subcommand in `usage`.

mrjob no longer raises `AssertionError`, instead raising `ValueError`.

Added an experimental harness script (in `mrjob/spark`) to run basic MRJobs on Spark, potentially without Hadoop:

```
spark-submit mrjob_spark_harness.py module.of.YourMRJob input_path output_dir
```

Added *`map_pairs()`*, *`reduce_pairs()`*, and *`combine_pairs()`* methods to *`MRJob`*, to enable the Spark harness script.

3.12 0.6.6

Fixes a longstanding bug where boolean jobconf values were passed to Hadoop in Python format (`True` instead of `true`). You can now do safely do something like this:

```
runners:
  emr:
    jobconf:
      mapreduce.output.fileoutputformat.compress: true
```

whereas in prior versions of mrjob, you had to use `"true"` in quotes.

Added `-D` as a synonym for `--jobconf`, to match Hadoop.

On EMR, if you have SSH set up (see *Configuring SSH credentials*) mrjob can fetch your history log directly from HDFS, allowing it to more quickly diagnose why your job failed.

Added a `--local-tmp-dir` switch. If you set local_tmp_dir to empty string, mrjob will use the system default.

You can now pass multiple arguments to Hadoop `--hadoop-args` (for example, `--hadoop-args='-fs hdfs://namenode:port'`), rather than having to use `--hadoop-arg` one argument at time. `--hadoop-arg` is now deprecated.

Similarly, you can use `--spark-args` to pass arguments to `spark-submit` in place of the now-deprecated `--spark-arg`.

mrjob no longer automatically passes generic arguments (`-D` and `-libjars`) to *JarStep*s, because this confuses some JARs. If you want mrjob to pass generic arguments to a JAR, add *GENERIC_ARGS* to your *JarStep*'s *args* keyword argument, like you would with *INPUT* and *OUTPUT*.

The Hadoop runner now has a spark_deploy_mode option.

Fixed the `usage:  usage:` typo in `--help` messages.

mrjob.job.MRJob.add_file_arg() can now take an explicit `type=str` (used to cause an error).

The deprecated `optparse` emulation methods `add_file_option()` and `add_passthrough_option()` now support `type='str'` (used to only accept `type='string'`).

Fixed a permissions error that was breaking `inline` and `local` mode on some versions of Windows.

3.13 0.6.5

This release fixes an issue with self-termination of idle clusters on EMR (see max_mins_idle) where the master node sometimes simply ignored `sudo shutdown -h now`. The idle self termination script now logs to `bootstrap-actions/mrjob-idle-termination.log`.

Note: If you are using *Cluster Pooling*, it's highly recommended you upgrade to this version to fix the self-termination issue.

You can now turn off log parsing (on all runners) by setting read_logs to false. This can speed up cases where you don't care why a job failed (e.g. integration tests) or where you'd rather use the *diagnose* tool after the fact.

You may specify custom AMIs with the image_id option. To find Amazon Linux AMIs compatible with EMR that you can use as a base for your custom image, use *describe_base_emr_images()*.

The default AMI on EMR is now 5.16.0.

New EMR clusters launched by mrjob will be automatically tagged with `__mrjob_label` (filename of your mrjob script) and `__mrjob_owner` (your username), to make it easier to understand your mrjob usage in CloudWatch etc. You can change the value of these tags with the label and owner options.

You may now set the root EBS volume size for EMR clusters directly with ebs_root_volume_gb (you used to have to use instance_groups or instance_fleets).

API clients returned by *EMRJobRunner* now retry on SSL timeouts. EMR clients returned by *mrjob.emr.EMRJobRunner.make_emr_client()* won't retry faster than check_cluster_every, to prevent throttling.

Cluster pooling recovery (relaunching a job when your pooled cluster self-terminates) now works correctly on single-node clusters.

3.14 0.6.4

This release makes it easy to attach static files to your *MRJob* with the *FILES*, *DIRS*, and *ARCHIVES* attributes.

In most cases, you no longer need setup scripts to access other python modules or packages from your job because you can use `DIRS` instead. For more details, see *Using other python modules and packages*.

For completeness, also added `files()`, `dirs()`, and `archives()` methods.

terminate-idle-clusters now skips termination-protected idle clusters, rather than crashing (this is fixed in *0.5.12*, but not previous 0.6.x versions).

Python 3.3 is no longer supported.

mrjob now requires `google-cloud-dataproc` 0.2.0+ (this library used to be vendored).

3.15 0.6.3

3.15.1 Read arbitrary file formats

You can now pass entire files in any format to your mapper by defining `mapper_raw()`. See *Passing entire files to the mapper* for an example.

3.15.2 Google Cloud Datatproc parity

mrjob now offers feature parity between Google Cloud Dataproc and Amazon Elastic MapReduce. Support for Spark and libjars will be added in a future release. (There is no plan to introduce *Cluster Pooling* with Dataproc.)

Specifically, `DataprocJobRunner` now supports:

- fetching and parsing counters
- parsing logs for probable cause of failure
- job progress messages (% complete)
- *Jar steps*
- these config options:
  - cloud_part_size_mb (chunked uploading)
  - core_instance_config, master_instance_config, task_instance_config
  - hadoop_streaming_jar
  - network/subnet (running in a VPC)
  - service_account (custom IAM account)
  - service_account_scopes (fine-grained permissions)
  - ssh_tunnel/ssh_tunnel_is_open (resource manager)

Improvements to existing Dataproc features:

- bootstrap scripts run in a temp dir, rather than `/`
- uses Dataproc's built-in auto-termination feature, rather than a script
- GCS filesystem:
  - `cat()` streams data rather than dumping to a temp file
  - `exists()` no longer swallows all exceptions

To get started, read *Getting started with Google Cloud*.

3.15.3 Other changes

mrjob no longer streams your job output to the command line if you specify output_dir. You can control this with the **--cat-output** and **--no-cat-output** switches (**--no-output** is deprecated).

cloud_upload_part_size has been renamed to cloud_part_size_mb (the old name will work until v0.7.0).

mrjob can now recognize "not a valid JAR" errors from Hadoop and suggest them as probable cause of job failure.

mrjob no longer depends on `google-cloud` (which implies several other Google libraries). Its current Google library dependencies are `google-cloud-logging` 1.5.0+ and `google-cloud-storage` 1.9.0+. Future versions of mrjob will depend on `google-cloud-dataproc` 0.11.0+ (currently included with mrjob because it hasn't yet been released).

`RetryWrapper` now sets `__name__` when wrapping methods, making for easier debugging.

3.16 0.6.2

mrjob is now orders of magnitude quicker at parsing logs, making it practical to diagnose rare errors from very large jobs. However, on some AMIs, it can no longer parse errors without waiting for logs to transfer to S3 (this may be fixed in a future version).

To run jobs on Google Cloud Dataproc, mrjob no longer requires you to install the **gcloud** util (though if you do have it installed, mrjob can read credentials from its configs). For details, see Dataproc Quickstart.

mrjob no longer requires you to select a Dataproc zone prior to running jobs. Auto zone placement (just set region and let Dataproc pick a zone) is now enabled, with the default being auto zone placement in `us-west1`. mrjob no longer reads zone and region from **gcloud**'s compute engine configs.

mrjob's Dataproc code has been ported from the `google-python-api-client` library (which is in maintenance mode) to `google-cloud-sdk`, resulting in some small changes to the GCS filesystem API. See CHANGES.txt for details.

Local mode now has a num_cores option that allow you to control how tasks it handles simultaneously.

3.17 0.6.1

Added the *diagnose* tool (run **mrjob diagnose j-CLUSTERID**), which determines why a previously run job failed.

Fixed a serious bug that made mrjob unable to properly parse error logs in some cases.

Added the *`get_job_steps()`* method to *`EMRJobRunner`*.

3.18 0.6.0

3.18.1 Dropped Python 2.6

mrjob now supports Python 2.7 and Python 3.3+. (Some versions of PyPy also work but are not officially supported.)

3.18.2 boto3, not boto

mrjob now uses `boto3` rather than `boto` to talk to AWS. This makes it much simpler to pass user-defined data structures directly to the API, enabling a number of features.

At least version 1.4.6 of `boto3` is required to run jobs on EMR.

It is now possible to fully configure instances (including EBS volumes). See instance_groups for an example.

mrjob also now supports Instance Fleets, which may be fully configured (including EBS volumes) through the instance_fleets option.

Methods that took or returned `boto` objects (for example, `make_emr_conn()`) have been completely removed as there as no way to make a deprecated shim for them without keeping `boto` as a dependency. See *EMRJobRunner* and *S3Filesystem* for new method names.

Note that `boto3` reads temporary credentials from `$AWS_SESSION_TOKEN`, not `$AWS_SECURITY_TOKEN` as in `boto` (see aws_session_token for details).

3.18.3 argparse, not optparse

mrjob now uses `argparse` to parse options, rather than `optparse`, which has been deprecated since Python 2.7.

`argparse` has slightly different option-parsing logic. A couple of things you should be aware of:

- everything that starts with `-` is assumed to be a switch. `--hadoop-arg=-verbose` works, but `--hadoop-arg -verbose` does not.
- positional arguments may not be split. `mr_wc.py CHANGES.txt LICENSE.txt -r local` will work, but `mr_wc.py CHANGES.txt -r local LICENSE.txt` will not.

Passthrough options, file options, etc. are now handled with *add_file_arg()*, *add_passthru_arg()*, *configure_args()*, *load_args()*, and *pass_arg_through()*. The old methods with "option" in their name are deprecated but still work.

As part of this refactor, *OptionStore* and its subclasses have been removed; options are now handled by runners directly.

3.18.4 Chunks, not lines

mrjob no longer assumes that job output will be line-based. If you *run your job programmatically*, you should read your job output with *cat_output()*, which yields bytestrings which don't necessarily correspond to lines, and run these through *parse_output()*, which will convert them into key/value pairs.

`runner.fs.cat()` also now yields arbitrary bytestrings, not lines. When it yields from multiple files, it will yield an empty bytestring (`b''`) between the chunks from each file.

`read_file()` and `read_input()` are now deprecated because they are line-based. Try *decompress()*, *to_chunks()*, and *to_lines()*.

3.18.5 Better local/inline mode

The sim runners (`inline` and `local` mode) have been completely rewritten, making it possible to fix a number of outstanding issues.

Local mode now runs one mapper/reducer per CPU, using `multiprocesssing`, for faster results.

We only sort by reducer key (not the full line) unless `SORT_VALUES` is set, exposing bad assumptions sooner.

The step_output_dir option is now supported, making it easier to debug issues in intermediate steps.

Files in tasks' (e.g. mappers') working directories are marked user-executable, to better imitate Hadoop Distributed Cache. When possible, we also symlink to a copy of each file/archive in the "cache," rather than copying them.

If `os.symlink()` raises an exception, we fall back to copying (this can be an issue in Python 3 on Windows).

Tasks are run more like they are in Hadoop; input is passed through stdin, rather than as script arguments. *`mrjob.cat`* is no longer executable because local mode no longer needs it.

3.18.6 Cloud runner improvements

Much of the common code for the "cloud" runners (Dataproc and EMR) has been merged, so that new features can be rolled out in parallel.

The bootstrap option (for both Dataproc and EMR) can now take archives and directories as well as files, like the setup option has since version *0.5.8*.

The extra_cluster_params option allows you to pass arbitrary JSON to the API at cluster create time (in Dataproc and EMR). The old *emr_api_params* option is deprecated and disabled.

max_hours_idle has been replaced with max_mins_idle (the old option is deprecated but still works). The default is 10 minutes. Due to a bug, smaller numbers of minutes might cause the cluster to terminate before the job runs.

It is no longer possible for mrjob to launch a cluster that sits idle indefinitely (except by setting max_mins_idle to an unreasonably high value). It is still a good idea to run *report-long-jobs* because mrjob can't tell if a running job is doing useful work or has stalled.

3.18.7 EMR now bills by the second, not the hour

Elastic MapReduce recently stopped billing by the full hour, and now bills by the second. This means that *Cluster Pooling* is no longer a cost-saving strategy, though developers might find it handy to reduce wait times when testing.

The *mins_to_end_of_hour* option no longer makes sense, and has been deprecated and disabled.

audit-emr-usage has been updated to use billing by the second when approximating time billed and waste.

Note: Pooling was enabled by default for some development versions of v0.6.0, prior to the billing change. This did not make it into the release; you must still explicitly turn on *cluster pooling*.

3.18.8 Other EMR changes

The default AMI is now 5.8.0. Note that this means you get Spark 2 by default.

Regions are now case-sensitive, and the `EU` alias for `eu-west-1` no longer works.

Pooling no longer adds dummy arguments to the master bootstrap script, instead setting the `__mrjob_pool_hash` and `__mrjob_pool_name` tags on the cluster.

mrjob automatically adds the `__mrjob_version` tag to clusters it creates.

Jobs will not add tags to clusters they join rather than create.

enable_emr_debugging now works on AMI 4.x and later.

AMI 2.4.2 and earlier are no longer supported (no Python 2.7). There is no longer any special logic for the "latest" AMI alias (which the API no longer supports).

The SSH filesystem no longer dumps file contents to memory.

Pooling will only join a cluster with enough *running* instances to meet its specifications; *requested* instances no longer count.

Pooling is now aware of EBS (disk) setup.

Pooling won't join a cluster that has extra instance types that don't have enough memory or disk space to run your job.

Errors in bootstrapping scripts are no longer dumped as JSON.

visible_to_all_users is deprecated.

3.18.9 Massive purge of deprecated code

About a hundred functions, methods, options, and more that were deprecated in v0.5.x have been removed. See CHANGES.txt for details.

3.19 0.5.12

This release came out after v0.6.3. It was mostly a backport from v0.6.x.

Python 2.6 and 3.3 are no longer supported.

`mrjob.parse.parse_s3_uri()` handles `s3a://` URIs.

terminate-idle-clusters now skips termination-protected idle clusters, rather than crashing.

Since Amazon no longer bills by the full hour, the *mins_to_end_of_hour* option now defaults to 60, effectively disabling it.

When mrjob passes an environment dictionary to subprocesses, it ensures that the keys and values are always `str`s (this mostly affects Python 2 on Windows).

3.20 0.5.11

The *report-long-jobs* utility can now ignore certain clusters based on EMR tags.

This version deals more gracefully with clusters that use instance fleets, preventing crashes that may occur in some rare edge cases.

3.21 0.5.10

Fixed an issue where bootstrapping mrjob on Dataproc or EMR could stall if mrjob was already installed.

The *aws_security_token* option has been renamed to aws_session_token. If you want to set it via environment variable, you still have to use `$AWS_SECURITY_TOKEN` because that's what boto uses.

Added protocol support for `rapidjson`; see `RapidJSONProtocol` and `RapidJSONValueProtocol`. If available, `rapidjson` will be used as the default JSON implementation if `ujson` is not installed.

The master bootstrap script on EMR and Dataproc now has the correct file extension (`.sh`, not `.py`).

3.22 0.5.9

Fixed a bug that prevented setup scripts from working on EMR AMIs 5.2.0 and later. Our workaround should be completely transparent unless you use a custom shell binary; see sh_bin for details.

The EMR runner now correctly re-starts the SSH tunnel to the job tracker/resource manager when a cluster it tries to run a job on auto-terminates. It also no longer requires a working SSH tunnel to fetch job progress (you still a working SSH; see ec2_key_pair_file).

The *emr_applications* option has been renamed to applications.

The *terminate-idle-clusters* utility is now slightly more robust in cases where your S3 temp directory is an different region from your clusters.

Finally, there a couple of changes that probably only matter if you're trying to wrap your Hadoop tasks (mappers, reducers, etc.) in **`docker`**:

- You can set *just* the python binary for tasks with task_python_bin. This allows you to use a wrapper script in place of Python without perturbing setup scripts.
- Local mode now no longer relies on an absolute path to access the *`mrjob.cat`* utility it uses to handle compressed input files; copying the job's working directory into Docker is enough.

3.23 0.5.8

You can now pass directories to jobs, either directly with the upload_dirs option, or through setup commands. For example:

```
--setup 'export PYTHONPATH=$PYTHONPATH:your-src-code/#'
```

mrjob will automatically tarball these directories and pass them to Hadoop as archives.

For multi-step jobs, you can now specify where inter-step output goes with step_output_dir (`--step-output-dir`), which can be useful for debugging.

All *`job step types`* now take the *jobconf* keyword argument to set Hadoop properties for that step.

Jobs' `--help` printout is now better-organized and less verbose.

Made several fixes to pre-filters (commands that pipe into streaming steps):

- you can once again add pre-filters to a single step job by re-defining *`mapper_pre_filter()`*, *`combiner_pre_filter()`*, and/or *`reducer_pre_filter()`*
- local mode now ignores non-zero return codes from pre-filters (this matters for BSD grep)
- local mode can now run pre-filters on compressed input files

mrjob now respects sh_bin when it needs to wrap a command in `sh` before passing it to Hadoop (e.g. to support pipes)

On EMR, mrjob now fetches logs from task nodes when determining probable cause of error, not just core nodes (the ones that run tasks and host HDFS).

Several unused functions in *`mrjob.util`* are now deprecated:

- `args_for_opt_dest_subset()`
- `bash_wrap()`
- `populate_option_groups_with_options()`
- `scrape_options_and_index_by_dest()`

- `tar_and_gzip()`

bunzip2_stream() and *gunzip_stream()* have been moved from *mrjob.util* to *mrjob.cat*.

`SSHFilesystem.ssh_slave_hosts()` has been deprecated.

Option group attributes in *MRJob*s have been deprecated, as has the `get_all_option_groups()` method.

3.24 0.5.7

3.24.1 Spark and related changes

mrjob now supports running Spark jobs on your own Hadoop cluster or Elastic MapReduce. mrjob provides significant benefits over Spark's built-in Python support; see *Why use mrjob with Spark?* for details.

Added the py_files option, to put *.zip* or *.egg* files in your job's `PYTHONPATH`. This is based on a Spark feature, but it works with streaming jobs as well. mrjob is now bootstrapped (see bootstrap_mrjob) as a *.zip* file rather than a tarball. If for some reason, the bootstrapped mrjob library won't compile, you'll get much cleaner error messages.

The default AMI version on EMR (see image_version) has been bumped from 3.11.0 to 4.8.2, as 3.11.0's Spark support is spotty.

On EMR, mrjob now defaults to the cheapest instance type that will work (see instance_type). In most cases, this is `m1.medium`, but it needs to be `m1.large` for Spark worker nodes.

3.24.2 Cluster pooling

mrjob can now add up to 1,000 steps on *pooled clusters* on EMR (except on very old AMIs). mrjob now prints debug messages explaining why your job matched a particular pooled cluster when running in verbose mode (the `-v` option). Fixed a bug that caused pooling to fail when there was no need for a master bootstrap script (e.g. when running with `--no-bootstrap-mrjob`).

3.24.3 Other improvements

Log interpretation is much more efficient at determining a job's probable cause of failure (this works with Spark as well).

When running custom JARs (see *JarStep*) mrjob now repects libjars and jobconf.

The hadoop_streaming_jar option now supports environment variables and `~`.

The *terminate-idle-clusters* tool now works with all step types, including Spark. (It's still recommended that you rely on the *max_hours_idle* option rather than this tool.)

mrjob now works in Anaconda3 Jupyter Notebook.

3.24.4 Bugfixes

Added several missing command-line switches, including `--no-bootstrap-python` on Dataproc. Made a major refactor that should prevent these kinds of issues in the future.

Fixed a bug that caused mrjob to crash when the ssh binary (see ssh_bin) was missing or not executable.

Fixed a bug that erroneously reported failed or just-started jobs as 100% complete.

Fixed a bug where timestamps were erroneously recognized as URIs. mrjob now only recognizes strings containing `://` as URIs (see *is_uri()*).

3.24.5 Deprecation

The following are deprecated and will be removed in v0.6.0:

- *JarStep*.``INPUT``; use *mrjob.step.INPUT* instead
- *JarStep*.``OUTPUT``; use *mrjob.step.OUTPUT* instead
- non-strict protocols (see *strict_protocols*)
- the *python_archives* option (try *this* instead)
- `is_windows_path()`
- `parse_key_value_list()`
- `parse_port_range_list()`
- `scrape_options_into_new_groups()`

3.25 0.5.6

Fixed a critical bug that caused Dataproc runner to always crash when determining Hadoop version.

Log interpretation now prioritizes task errors (e.g. a traceback from your Python script) as probable cause of failure, even if they aren't the most recent error. Log interpretation will now continue to download and parse task logs until it finds a non-empty stderr log.

Log interpretation also strips the "subprocess failed" Java stack trace that appears in task stderr logs from Hadoop 1.

3.26 0.5.5

Functionally equivalent to *0.5.4*, except that it restores the deprecated *ami_version* option as an alias for image_version, making it easier to upgrade from earlier versions of mrjob.

Also slightly improves *Cluster Pooling* on EMR with updated information on memory and CPU power of various EC2 instance types, and by treating application names (e.g. "Spark") as case-insensitive.

3.27 0.5.4

3.27.1 Pooling and idle cluster self-termination

> **Warning:** This release accidentally removed the *ami_version* option instead of merely deprecating it. If you are upgrading from an earlier version of mrjob, use version *0.5.5* or later.

This release resolves a long-standing EMR API race condition that made it difficult to use *Cluster Pooling* and idle cluster self-termination (see *max_hours_idle*) together. Now if your pooled job unknowingly runs on a cluster that was in the process of shutting down, it will detect that and re-launch the job on a different cluster.

This means pretty much *everyone* running jobs on EMR should now enable pooling, with a configuration like this:

```
runners:
  emr:
    max_hours_idle: 1
    pool_clusters: true
```

You may *also* run the *terminate-idle-clusters* script periodically, but (barring any bugs) this shouldn't be necessary.

3.27.2 Generic EMR option names

Many options to the EMR runner have been made more generic, to make it easier to share code with the Dataproc runner (in most cases, the new names are also shorter and easier to remember):

old option name	new option name
ami_version	image_version
aws_availablity_zone	zone
aws_region	region
check_emr_status_every	check_cluster_every
ec2_core_instance_bid_price	core_instance_bid_price
ec2_core_instance_type	core_instance_type
ec2_instance_type	instance_type
ec2_master_instance_bid_price	master_instance_bid_price
ec2_master_instance_type	master_instance_type
ec2_slave_instance_type	core_instance_type
ec2_task_instance_bid_price	task_instance_bid_price
ec2_task_instance_type	task_instance_type
emr_tags	tags
num_ec2_core_instances	num_core_instances
num_ec2_task_instances	num_task_instances
s3_log_uri	cloud_log_dir
s3_sync_wait_time	cloud_fs_sync_secs
s3_tmp_dir	cloud_tmp_dir
s3_upload_part_size	*cloud_upload_part_size*

The old option names and command-line switches are now deprecated but will continue to work until v0.6.0. (Exception: *ami_version* was accidentally removed; if you need it, use *0.5.5* or later.)

num_ec2_instances has simply been deprecated (it's just num_core_instances plus one).

hadoop_streaming_jar_on_emr has also been deprecated; in its place, you can now pass a `file://` URI to hadoop_streaming_jar to reference a path on the master node.

3.27.3 Log interpretation

Log interpretation (counters and probable cause of job failure) on Hadoop is more robust, handing a wider variety of log4j formats and recovering more gracefully from permissions errors. This includes fixing a crash that could happen on Python 3 when attempting to read data from HDFS.

Log interpretation used to be partially broken on EMR AMI 4.3.0 and later due to a permissions issue; this is now fixed.

3.27.4 pass_through_option()

You can now pass through *existing* command-line switches to your job; for example, you can tell a job which runner launched it. See `pass_through_option()` for details.

If you *don't* do this, `self.options.runner` will now always be `None` in your job (it used to confusingly default to `'inline'`).

3.27.5 Stop logging credentials

When mrjob is run in verbose mode (the `-v` option), the values of all runner options are debug-logged to stderr. This has been the case since the very early days of mrjob.

Unfortunately, this means that if you set your AWS credentials in `mrjob.conf`, they get logged as well, creating a surprising potential security vulnerability. (This doesn't happen for AWS credentials set through environment variables.)

Starting in this version, the values of aws_secret_access_key and *aws_security_token* are shown as `'...'` if they are set, and all but the last four characters of aws_access_key_id are blanked out as well (e.g. `'...YNDR'`).

3.27.6 Other improvements and bugfixes

The ssh tunnel to the resource manager on EMR (see ssh_tunnel) now connects to its correct *internal* IP; this resolves a firewall issue that existed on some VPC setups.

Uploaded files will no longer be given names starting with _ or `.`, since Hadoop's input processing treats these files as "hidden".

The EMR idle cluster self-termination script (see *max_hours_idle*) now only runs on the master node.

The *audit-emr-usage* command-line tool should no longer constantly trigger throttling warnings.

bootstrap_python no longer bothers trying to install Python 3 on EMR AMI 4.6.0 and later, since it is already installed.

The `--ssh-bind-ports` command-line switch was broken (starting in *0.4.5*!), and is now fixed.

3.28 0.5.3

This release adds support for custom libjars (such as nicknack), allowing easy access to custom input and output formats. This works on Hadoop and EMR (including on a cluster that's already running).

In addition, jobs can specify needed libjars by setting the *`LIBJARS`* attribute or overriding the *`libjars()`* method. For examples, see *Input and output formats*.

The Hadoop runner now tries *even harder* to find your log files without needing additional configuration (see hadoop_log_dirs).

The EMR runner now supports Amazon VPC subnets (see subnet), and, on 4.x AMIs, Application Configurations (see emr_configurations).

If your EMR cluster fails during bootstrapping, mrjob can now determine the probable cause of failure.

There are also some minor improvements to SSH tunneling and a handful of small bugfixes; see CHANGES.txt for details.

3.29 0.5.2

This release adds basic support for Google Cloud Dataproc which is Google's Hadoop service, roughly analogous to EMR. See Dataproc Quickstart. Some features are not yet implemented:

- fetching counters
- finding probable cause of errors
- running Java JARs as steps

Added the *emr_applications* option, which helps you configure 4.x AMIs.

Fixed an EMR bug (introduced in v0.5.0) where we were waiting for steps to complete in the wrong order (in a multi-step job, we wouldn't register that the first step had finished until the last one had).

Fixed a bug in SSH tunneling (introduced in v0.5.0) that made connections to the job tracker/resource manager on EMR time out when running on a 2.x AMI inside a VPC (Virtual Private Cluster).

Fixed a bug (introduced in v0.4.6) that kept mrjob from interpreting ~ (home directory) in includes in `mrjob.conf`.

It is now again possible to run tool modules deprecated in v0.5.0 directly (e.g. **`python -m mrjob.tools.emr.create_job_flow`**). This is still a deprecated feature; it's recommended that you use the appropriate **`mrjob`** subcommand instead (e.g. **`mrjob create-cluster`**).

3.30 0.5.1

Fixes a bug in the previous relase that broke *`SORT_VALUES`* and any other attempt by the job to set the partitioner. The `--partitioner` switch is now deprecated (the choice of partitioner is part of your job semantics).

Fixes a bug in the previous release that caused *strict_protocols* and check_input_paths to be ignored in `mrjob.conf`. (We would much prefer you fixed jobs that are using "loose protocols" rather than setting `strict_protocols: false` in your config file, but we didn't break this on purpose, we promise!)

`mrjob terminate-idle-clusters` now correctly handles EMR debugging steps (see enable_emr_debugging) set up by boto 2.40.0.

Fixed a bug that could result in showing a blank probable cause of error for pre-YARN (Hadoop 1) jobs.

ssh_bind_ports now defaults to a `range` object (`xrange` on Python 2), so that when you run on emr in verbose mode (`-r emr -v`), debug logging devotes one line to the value of `ssh_bind_ports` rather than 840.

3.31 0.5.0

3.31.1 Python versions

mrjob now fully supports Python 3.3+ in a way that should be transparent to existing Python 2 users (you don't have to suddenly start handling `unicode` instead of `str`). For more information, see Python 2 vs. Python 3.

If you run a job with Python 3, mrjob will automatically install Python 3 on ElasticMapreduce AMIs (see bootstrap_python).

When you run jobs on EMR in Python 2, mrjob attempts to match your minor version of Python as well (either **`python2.6`** or **`python2.7`**); see python_bin for details.

Note: If you're currently running Python 2.7, and *using yum to install python libraries*, you'll want to use the Python 2.7 version of the package (e.g. `python27-numpy` rather than `python-numpy`).

The **`mrjob`** command is now installed with Python-version-specific aliases (e.g. **`mrjob-3`**, **`mrjob-3.4`**), in case you install mrjob for multiple versions of Python.

3.31.2 Hadoop

mrjob should now work out-of-the box on almost any Hadoop setup. If **`hadoop`** is in your path, or you set any commonly-used `$HADOOP_*` environment variable, mrjob will find the Hadoop binary, the streaming jar, and your logs, without any help on your part (see hadoop_bin, hadoop_log_dirs, hadoop_streaming_jar).

mrjob has been updated to fully support Hadoop 2 (YARN), including many updates to `HadoopFilesystem`. Hadoop 1 is still supported, though anything prior to Hadoop 0.20.203 is not (mrjob is actually a few months older than Hadoop 0.20.203, so this used to matter).

3.31.3 3.x and 4.x AMIs

mrjob now fully supports the 3.x and 4.x Elastic MapReduce AMIs, including SSH tunneling to the resource manager, fetching counters and finding probable cause of job failure.

The default *ami_version* (see image_version) is now `3.11.0`. Our plan is to continue updating this to the lastest (non-broken) 3.x AMI for each 0.5.x release of mrjob.

The default instance_type is now `m1.medium` (`m1.small` is too small for the 3.x and 4.x AMIs)

You can specify 4.x AMIs with either the new release_label option, or continue using *ami_version*; both work.

mrjob continues to support 2.x AMIs. However:

> **Warning:** 2.x AMIs are deprecated by AWS, and based on a very old version of Debian (squeeze), which breaks **`apt-get`** and exposes you to security holes.

Please, please switch if you haven't already.

3.31.4 AWS Regions

The new default *aws_region* (see region) is `us-west-2` (Oregon). This both matches the default in the EMR console and, according to Amazon, is carbon neutral.

An edge case that might affect you: EC2 key pairs (i.e. SSH credentials) are region-specific, so if you've set up SSH but not explicitly specified a region, you may get an error saying your key pair is invalid. The fix is simply to *create new SSH keys* for the `us-west-2` (Oregon) region.

3.31.5 S3

mrjob is much smarter about the way it interacts with S3:

- automatically creates temp bucket in the same region as jobs
- connects to S3 buckets on the endpoint matching their region (no more 307 errors)
  - *`EMRJobRunner`* and *`S3Filesystem`* methods no longer take `s3_conn` args (passing around a single S3 connection no longer makes sense)
- no longer uses the temp bucket's location to choose where you run your job
- `rm()` no longer has special logic for `*_$folder$` keys
- `ls()` recurses "subdirectories" even if you pass it a URI without a trailing slash

3.31.6 Log interpretation

The part of mrjob that fetches counters and tells you what probably caused your job to fail was basically unmaintainable and has been totally rewritten. Not only do we now have solid support across Hadoop and EMR AMI versions, but if we missed anything, it should be straightforward to add it.

Once casualty of this change was the **`mrjob fetch-logs`** command, which means mrjob no longer offers a way to fetch or interpret logs from a *past* job. We do plan to re-introduce this functionality.

3.31.7 Protocols

Protocols are now strict by default (they simply raise an exception on unencodable data). "Loose" protocols can be re-enabled with the `--no-strict-protocols` switch; see *strict_protocols* for why this is a bad idea.

Protocols will now use the much faster `ujson` library, if installed, to encode and decode JSON. This is especially recommended for simple jobs that spend a significant fraction of their time encoding and data.

Note: If you're using EMR, try out *this bootstrap recipe* to install `ujson`.

mrjob will fall back to the `simplejson` library if `ujson` is not installed, and use the built-in `json` module if neither is installed.

You can now explicitly specify which JSON implementation you wish to use (e.g. *`StandardJSONProtocol`*, *`SimpleJSONProtocol`*, *`UltraJSONProtocol`*).

3.31.8 Status messages

We've tried to cut the logging messages that your job prints as it runs down to the basics (either useful info, like where a temp directory is, or something that tells you why you're waiting). If there are any messages you miss, try running your job with `-v`.

When a step in your job fails, mrjob no longer prints a useless stacktrace telling you where in the code the runner raised an exception about your step failing. This is thanks to `StepFailedException`, which you can also catch and interpret if you're *running jobs programmatically*.

3.31.9 Deprecation

Many things that were deprecated in 0.4.6 have been removed:

- options:
  - `IF_SUCCESSFUL` cleanup option (use `ALL`)
  - *iam_job_flow_role* (use iam_instance_profile)
- functions and methods:
  - positional arguments to `mrjob.job.MRJob.mr()` (don't even use `mr()`; use *`mrjob.step.MRStep`*)
  - `mrjob.job.MRJob.jar()` (use *`mrjob.step.JarStep`*)
  - *step_args* and *name* arguments to *`mrjob.step.JarStep`* (use *args* instead of *step_args*, and don't use *name* at all)
  - `mrjob.step.MRJobStep` (use *`mrjob.step.MRStep`*)
  - `mrjob.compat.get_jobconf_value()` (use to *`jobconf_from_env()`*)

  - `mrjob.job.MRJob.parse_counters()`
  - *`mrjob.job.MRJob.parse_output()`*
  - `mrjob.conf.combine_cmd_lists()`
  - `mrjob.fs.s3.S3Filesystem.get_s3_folder_keys()`

`mrjob.compat` functions `supports_combiners_in_hadoop_streaming()`, `supports_new_distributed_cache_options()`, and `uses_generic_jobconf()`, which only existed to support very old versions of Hadoop, were removed without deprecation warnings (sorry!).

To avoid a similar wave of deprecation warnings in the future, the name of every part of mrjob that isn't meant to be a stable interface provided by the library now starts with an underscore. You can still use these things (or copy them; it's Open Source), but there's no guarantee they'll exist in the next release.

If you want to get ahead of the game, here is a list of things that are deprecated starting in mrjob 0.5.0 (do these *after* upgrading mrjob):

- options:
  - *base_tmp_dir* is now local_tmp_dir
  - cleanup options `LOCAL_SCRATCH` and `REMOTE_SCRATCH` are now `LOCAL_TMP` and `REMOTE_TMP`
  - *emr_job_flow_id* is now cluster_id
  - *emr_job_flow_pool_name* is now pool_name
  - *hdfs_scratch_dir* is now hadoop_tmp_dir
  - *pool_emr_job_flows* is now pool_clusters
  - *s3_scratch_uri* is now cloud_tmp_dir
  - *ssh_tunnel_to_job_tracker* is now simply ssh_tunnel
- functions and methods:
  - `mrjob.job.MRJob.is_mapper_or_reducer()` is now *`is_task()`*
  - *`Filesystem`* method `path_exists()` is now simply *`exists()`*
  - *`Filesystem`* method `path_join()` is now simply *`join()`*
  - Use `runner.fs` explicitly when accessing filesystem methods (e.g. `runner.fs.ls()`, not `runner.ls()`)
- **`mrjob`** subcommands - **`mrjob create-job-flow`** is now **`mrjob create-cluster`** - **`mrjob terminate-idle-job-flows`** is now **`mrjob terminate-idle-clusters`** - **`mrjob terminate-job-flow`** is now **`mrjob temrinate-cluster`**

3.31.10 Other changes

- mrjob now requires `boto` 2.35.0 or newer (chances are you're already doing this). Later 0.5.x releases of mrjob may require newer versions of `boto`.
- *visible_to_all_users* now defaults to `True`
- `HadoopFilesystem.rm()` uses `-skipTrash`
- new iam_endpoint option
- custom hadoop_streaming_jars are properly uploaded

- `JOB` cleanup on EMR is temporarily disabled
- mrjob now follows symlinks when `ls()`ing the local filesystem (beware recursive symlinks!)
- The *interpreter* option disables bootstrap_mrjob by default (*interpreter* is meant for non-Python jobs)
- *Cluster Pooling* now respects ec2_key_pair
- cluster self-termination (see *max_hours_idle*) now respects non-streaming jobs
- `LocalFilesystem` now rejects URIs rather than interpreting them as local paths
- `local` and `inline` runners no longer have a default hadoop_version, instead handling jobconf in a version-agnostic way
- *steps_python_bin* now defaults to the current Python interpreter.
- minor changes to *`mrjob.util`*:
  - *`file_ext()`* takes filename, not path
  - `gunzip_stream()` now yields chunks of bytes, not lines
  - moved *`random_identifier()`* method here from `mrjob.aws`
  - `buffer_iterator_to_line_iterator()` is now named *`to_lines()`*, and no longer appends a trailing newline to data.

3.32 0.4.6

`include:` in conf files can now use relative paths in a meaningful way. See *Relative includes*.

List and environment variable options loaded from included config files can be totally overridden using the `!clear` tag. See *Clearing configs*.

Options that take lists (e.g. setup) now treat scalar values as single-item lists. See *this example*.

Fixed a bug that kept the `pool_wait_minutes` option from being loaded from config files.

3.33 0.4.5

This release moves mrjob off the deprecated DescribeJobFlows EMR API call.

Warning: AWS *again* broke older versions mrjob for at least some new accounts, by returning 400s for the deprecated DescribeJobFlows API call. If you have a newer AWS account (circa July 2015), you must use at least this version of mrjob.

The new API does not provide a way to tell when a job flow (now called a "cluster") stopped provisioning instances and started bootstrapping, so the clock for our estimates of when we are close to the end of a billing hour now start at cluster creation time, and are thus more conservative.

Related to this change, `terminate_idle_job_flows` no longer considers job flows in the `STARTING` state idle; use `report_long_jobs` to catch jobs stuck in this state.

`terminate_idle_job_flows` performs much better on large numbers of job flows. Formerly, it collected all job flow information first, but now it terminates idle job flows as soon as it identifies them.

`collect_emr_stats` and `job_flow_pool` have *not* been ported to the new API and will be removed in v0.5.0.

Added an *aws_security_token* option to allow you to run mrjob on EMR using temporary AWS credentials.

Added an *emr_tags* (see tags) option to allow you to tag EMR job flows at creation time.

EMRJobRunner now has a `get_ami_version()` method.

The hadoop_version option no longer has any effect in EMR. This option only every did anything on the 1.x AMIs, which mrjob no longer supports.

Added many missing switches to the EMR tools (accessible from the **`mrjob`** command). Formerly, you had to use a config file to get at these options.

You can now access the `mrboss` tool from the command line: **`mrjob boss <args>`**.

Previous 0.4.x releases have worked with boto as old as 2.2.0, but this one requires at least boto 2.6.0 (which is still more than two years old). In any case, it's recommended that you just use the latest version of boto.

This branch has a number of additional deprecation warnings, to help prepare you for mrjob v0.5.0. Please heed them; a lot of deprecated things really are going to be completely removed.

3.34 0.4.4

mrjob now automatically creates and uses IAM objects as necessary to comply with new requirements from Amazon Web Services.

(You do not need to install the AWS CLI or run `aws emr create-default-roles` as the link above describes; mrjob takes care of this for you.)

> **Warning:** The change that AWS made essentially broke all older versions of mrjob for all new accounts. If the first time your AWS account created an Elastic MapReduce cluster was on or after April 6, 2015, you should use at least this version of mrjob.
> If you *must* use an old version of mrjob with a new AWS account, see this thread for a possible workaround.

`--iam-job-flow-role` has been renamed to `--iam-instance-profile`.

New `--iam-service-role` option.

3.35 0.4.3

This release also contains many, many bugfixes, one of which probably affects you! See CHANGES.txt for details.

Added a new subcommand, `mrjob collect-emr-active-stats`, to collect stats about active jobflows and instance counts.

`--iam-job-flow-role` option allows setting of a specific IAM role to run this job flow.

You can now use `--check-input-paths` and `--no-check-input-paths` on EMR as well as Hadoop.

Files larger than 100MB will be uploaded to S3 using multipart upload if you have the *filechunkio* module installed. You can change the limit/part size with the `--s3-upload-part-size` option, or disable multipart upload by setting this option to 0. You can now require protocols to be strict from *mrjob.conf*; this means unencodable input/output will result in an exception rather than the job quietly incrementing a counter. It is recommended you set this for all runners:

```
runners:
  emr:
    strict_protocols: true
  hadoop:
    strict_protocols: true
```

```
  inline:
    strict_protocols: true
  local:
    strict_protocols: true
```

You can use `--no-strict-protocols` to turn off strict protocols for a particular job.

Tests now support pytest and tox.

Support for Python 2.5 has been dropped.

3.36 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.

3.37 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).

3.38 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!

3.39 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.

3.40 0.3.3

You can now *include one config file from another*.

3.41 0.3.2

The EMR instance type/number options have changed to support spot instances:

- *core_instance_bid_price*
- *core_instance_type*
- *master_instance_bid_price*
- *master_instance_type*
- *slave_instance_type* (alias for *core_instance_type*)
- *task_instance_bid_price*
- *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.

3.42 0.3

3.42.1 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.

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: *Cluster Pooling*

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*

3.42.2 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.

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 `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 converts one 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

Symbols

A

B

C

D

E

N

O

P

R

S

T

U

V

W

Z