Luigi Documentation

Release 1.0

Erik Bernhardsson and Elias Freider

June 22, 2015

Contents

1 Getting Started 3

2 Documentation 5

3 More background 7

4 Dependency graph example 9

5 Background 11

6 Who uses Luigi? 13

7 Getting Help 15

8 External links 17

9 Authors 19

10 Table of Contents 21
- 10.1 Example – Top Artists . . . 21
- 10.2 Building workflows . . . 24
- 10.3 Tasks . . . 28
- 10.4 Parameters . . . 33
- 10.5 Running from the Command Line . . . 35
- 10.6 Programmatic Execution . . . 36
- 10.7 Using the Central Scheduler . . . 36
- 10.8 Execution Model . . . 40
- 10.9 Luigi Patterns . . . 41
- 10.10 Configuration . . . 43
- 10.11 More Info . . . 49

11 API Reference 51
- 11.1 luigi package . . . 51
- 11.2 luigi.contrib package . . . 135
- 11.3 luigi.tools package . . . 177
- 11.4 Indices and tables . . . 182

Python Module Index 183

Luigi is a Python package that helps you build complex pipelines of batch jobs. It handles dependency resolution, workflow management, visualization, handling failures, command line integration, and much more.

Run `pip install luigi` to install the latest stable version from PyPI.

For bleeding edge code, `git clone https://github.com/spotify/luigi` and `python setup.py install`. Bleeding edge documentation can be found here.

If you want to run the central scheduler (highly recommended), you need to install Tornado which you can do from PyPI as well: `pip install tornado`.

CHAPTER 1

Getting Started

Take a look at the Example workflow and Building workflows which explains some of the most important concepts.

CHAPTER 2

Documentation

Full documentation is available at Read the Docs, including the Luigi package documentation.

See Configuration for how to configure Luigi.

CHAPTER 3

More background

The purpose of Luigi is to address all the plumbing typically associated with long-running batch processes. You want to chain many tasks, automate them, and failures *will* happen. These tasks can be anything, but are typically long running things like Hadoop jobs, dumping data to/from databases, running machine learning algorithms, or anything else.

There are other software packages that focus on lower level aspects of data processing, like Hive, Pig, or Cascading. Luigi is not a framework to replace these. Instead it helps you stitch many tasks together, where each task can be a Hive query, a Hadoop job in Java, a Spark job in Scala or Python a Python snippet, dumping a table from a database, or anything else. It's easy to build up long-running pipelines that comprise thousands of tasks and take days or weeks to complete. Luigi takes care of a lot of the workflow management so that you can focus on the tasks themselves and their dependencies.

You can build pretty much any task you want, but Luigi also comes with a *toolbox* of several common task templates that you use. It includes support for running Python mapreduce jobs in Hadoop, as well as Hive, and Pig, jobs. It also comes with file system abstractions for HDFS, and local files that ensures all file system operations are atomic. This is important because it means your data pipeline will not crash in a state containing partial data.

CHAPTER 4

Dependency graph example

Just to give you an idea of what Luigi does, this is a screen shot from something we are running in production. Using Luigi's visualizer, we get a nice visual overview of the dependency graph of the workflow. Each node represents a task which has to be run. Green tasks are already completed whereas yellow tasks are yet to be run. Most of these tasks are Hadoop jobs, but there are also some things that run locally and build up data files.

CHAPTER 5

Background

We use Luigi internally at Spotify to run thousands of tasks every day, organized in complex dependency graphs. Most of these tasks are Hadoop jobs. Luigi provides an infrastructure that powers all kinds of stuff including recommendations, toplists, A/B test analysis, external reports, internal dashboards, etc. Luigi grew out of the realization that powerful abstractions for batch processing can help programmers focus on the most important bits and leave the rest (the boilerplate) to the framework.

Conceptually, Luigi is similar to GNU Make where you have certain tasks and these tasks in turn may have dependencies on other tasks. There are also some similarities to Oozie and Azkaban. One major difference is that Luigi is not just built specifically for Hadoop, and it's easy to extend it with other kinds of tasks.

Everything in Luigi is in Python. Instead of XML configuration or similar external data files, the dependency graph is specified *within Python*. This makes it easy to build up complex dependency graphs of tasks, where the dependencies can involve date algebra or recursive references to other versions of the same task. However, the workflow can trigger things not in Python, such as running Pig scripts or scp'ing files.

CHAPTER 6

Who uses Luigi?

Several companies have written blog posts or presentation about Luigi:

- Spotify (NYC Data Science)
- Foursquare
- Mortar Data
- Stripe
- Asana
- Buffer
- SeatGeek
- Treasure Data

Please let us know if your company wants to be featured on this list!

CHAPTER 7

Getting Help

- Find us on *#luigi* on freenode.
- Subscribe to the luigi-user group and ask a question.

CHAPTER 8

External links

- Mailing List (Google Groups)
- Releases (PyPi)
- Source code (Github)

CHAPTER 9

Authors

Luigi was built at Spotify, mainly by Erik Bernhardsson and Elias Freider. Many other people have contributed since open sourcing in late 2012. Arash Rouhani is currently the chief maintainer of Luigi.

CHAPTER 10

Table of Contents

10.1 Example – Top Artists

This is a very simplified case of something we do at Spotify a lot. All user actions are logged to HDFS where we run a bunch of Hadoop jobs to transform the data. At some point we might end up with a smaller data set that we can bulk ingest into Cassandra, Postgres, or some other format.

For the purpose of this exercise, we want to aggregate all streams, find the top 10 artists and then put the results into Postgres.

This example is also available in examples/top_artists.py.

10.1.1 Step 1 - Aggregate Artist Streams

```
class AggregateArtists(luigi.Task):
    date_interval = luigi.DateIntervalParameter()

    def output(self):
        return luigi.LocalTarget("data/artist_streams_%s.tsv" % self.date_interval)

    def requires(self):
        return [Streams(date) for date in self.date_interval]

    def run(self):
        artist_count = defaultdict(int)

        for input in self.input():
            with input.open('r') as in_file:
                for line in in_file:
                    timestamp, artist, track = line.strip().split()
                    artist_count[artist] += 1

        with self.output().open('w') as out_file:
            for artist, count in artist_count.iteritems():
                print >> out_file, artist, count
```

Note that this is just a portion of the file *examples/top_artists.py*. In particular, `Streams` is defined as a *`Task`*, acting as a dependency for `AggregateArtists`. In addition, `luigi.run()` is called if the script is executed directly, allowing it to be run from the command line.

There are several pieces of this snippet that deserve more explanation.

- Any `Task` may be customized by instantiating one or more `Parameter` objects on the class level.
- The `output()` method tells Luigi where the result of running the task will end up. The path can be some function of the parameters.
- The `requires()` tasks specifies other tasks that we need to perform this task. In this case it's an external dump named *Streams* which takes the date as the argument.
- For plain Tasks, the `run()` method implements the task. This could be anything, including calling subprocesses, performing long running number crunching, etc. For some subclasses of `Task` you don't have to implement the `run` method. For instance, for the `JobTask` subclass you implement a *mapper* and *reducer* instead.
- `HdfsTarget` is a built in class that makes it easy to read/write from/to HDFS. It also makes all file operations atomic, which is nice in case your script crashes for any reason.

10.1.2 Running this Locally

Try running this using eg.

```
$ python examples/top_artists.py AggregateArtists --local-scheduler --date-interval 2012-06
```

You can also try to view the manual using *–help* which will give you an overview of the options:

```
usage: wordcount.py [-h] [--local-scheduler] [--scheduler-host SCHEDULER_HOST]
                    [--lock] [--lock-pid-dir LOCK_PID_DIR] [--workers WORKERS]
                    [--date-interval DATE_INTERVAL]

optional arguments:
  -h, --help            show this help message and exit
  --local-scheduler     Use local scheduling
  --scheduler-host SCHEDULER_HOST
                        Hostname of machine running remote scheduler [default:
                        localhost]
  --lock                Do not run if the task is already running
  --lock-pid-dir LOCK_PID_DIR
                        Directory to store the pid file [default:
                        /var/tmp/luigi]
  --workers WORKERS     Maximum number of parallel tasks to run [default: 1]
  --date-interval DATE_INTERVAL
                        AggregateArtists.date_interval
```

Running the command again will do nothing because the output file is already created. In that sense, any task in Luigi is *idempotent* because running it many times gives the same outcome as running it once. Note that unlike Makefile, the output will not be recreated when any of the input files is modified. You need to delete the output file manually.

The *–local-scheduler* flag tells Luigi not to connect to a scheduler server. This is not recommended for other purpose than just testing things.

10.1.3 Step 1b - Running this in Hadoop

Luigi comes with native Python Hadoop mapreduce support built in, and here is how this could look like, instead of the class above.

```
class AggregateArtistsHadoop(luigi.contrib.hadoop.JobTask):
    date_interval = luigi.DateIntervalParameter()

    def output(self):
        return luigi.contrib.hdfs.HdfsTarget("data/artist_streams_%s.tsv" % self.date_interval)
    def requires(self):
        return [StreamsHdfs(date) for date in self.date_interval]

    def mapper(self, line):
        timestamp, artist, track = line.strip().split()
        yield artist, 1

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

Note that *luigi.contrib.hadoop.JobTask* doesn't require you to implement a *run()* method. Instead, you typically implement a *mapper()* and *reducer()* method.

10.1.4 Step 2 – Find the Top Artists

At this point, we've counted the number of streams for each artists, for the full time period. We are left with a large file that contains mappings of artist -> count data, and we want to find the top 10 artists. Since we only have a few hundred thousand artists, and calculating artists is nontrivial to parallelize, we choose to do this not as a Hadoop job, but just as a plain old for-loop in Python.

```
class Top10Artists(luigi.Task):
    date_interval = luigi.DateIntervalParameter()
    use_hadoop = luigi.BoolParameter()

    def requires(self):
        if self.use_hadoop:
            return AggregateArtistsHadoop(self.date_interval)
        else:
            return AggregateArtists(self.date_interval)

    def output(self):
        return luigi.LocalTarget("data/top_artists_%s.tsv" % self.date_interval)

    def run(self):
        top_10 = nlargest(10, self._input_iterator())
        with self.output().open('w') as out_file:
            for streams, artist in top_10:
                print >> out_file, self.date_interval.date_a, self.date_interval.date_b, artist, stre

    def _input_iterator(self):
        with self.input().open('r') as in_file:
            for line in in_file:
                artist, streams = line.strip().split()
                yield int(streams), int(artist)
```

The most interesting thing here is that this task (*Top10Artists*) defines a dependency on the previous task (*AggregateArtists*). This means that if the output of *AggregateArtists* does not exist, the task will run before *Top10Artists*.

```
$ python examples/top_artists.py Top10Artists --local-scheduler --date-interval 2012-07
```

This will run both tasks.

10.1.5 Step 3 - Insert into Postgres

This mainly serves as an example of a specific subclass *Task* that doesn't require any code to be written. It's also an example of how you can define task templates that you can reuse for a lot of different tasks.

```
class ArtistToplistToDatabase(luigi.postgres.CopyToTable):
    date_interval = luigi.DateIntervalParameter()
    use_hadoop = luigi.BoolParameter()

    host = "localhost"
    database = "toplists"
    user = "luigi"
    password = "abc123"  # ;)
    table = "top10"

    columns = [("date_from", "DATE"),
               ("date_to", "DATE"),
               ("artist", "TEXT"),
               ("streams", "INT")]

    def requires(self):
        return Top10Artists(self.date_interval, self.use_hadoop)
```

Just like previously, this defines a recursive dependency on the previous task. If you try to build the task, that will also trigger building all its upstream dependencies.

10.1.6 Using the Central Planner

The *–local-scheduler* flag tells Luigi not to connect to a central scheduler. This is recommended in order to get started and or for development purposes. At the point where you start putting things in production we strongly recommend running the central scheduler server. In addition to providing locking so that the same task is not run by multiple processes at the same time, this server also provides a pretty nice visualization of your current work flow.

If you drop the *–local-scheduler* flag, your script will try to connect to the central planner, by default at localhost port 8082. If you run

```
luigid
```

in the background and then run

```
$ python wordcount.py --date 2012-W03
```

then in fact your script will now do the scheduling through a centralized server. You need Tornado for this to work.

Launching *http://localhost:8082* should show something like this:

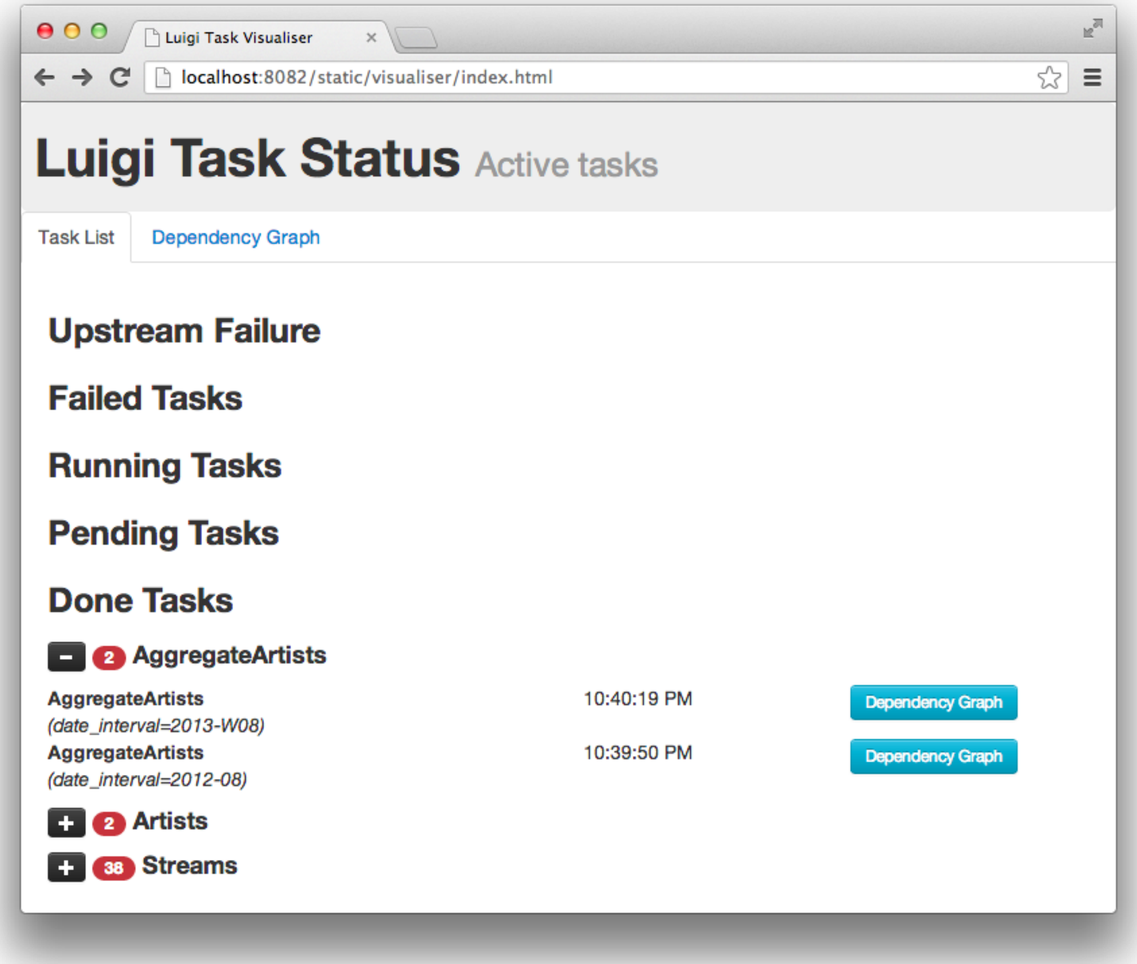
Web server screenshot Looking at the dependency graph for any of the tasks yields something like this:

Aggregate artists screenshot

In production, you'll want to run the centralized scheduler. See: Using the Central Scheduler for more information.

10.2 Building workflows

There are two fundamental building blocks of Luigi - the `Task` class and the `Target` class. Both are abstract classes and expect a few methods to be implemented. In addition to those two concepts, the `Parameter` class is an important concept that governs how a Task is run.



Luigi Task Status Active tasks

Upstream Failure

Failed Tasks

Running Tasks

Pending Tasks

Done Tasks

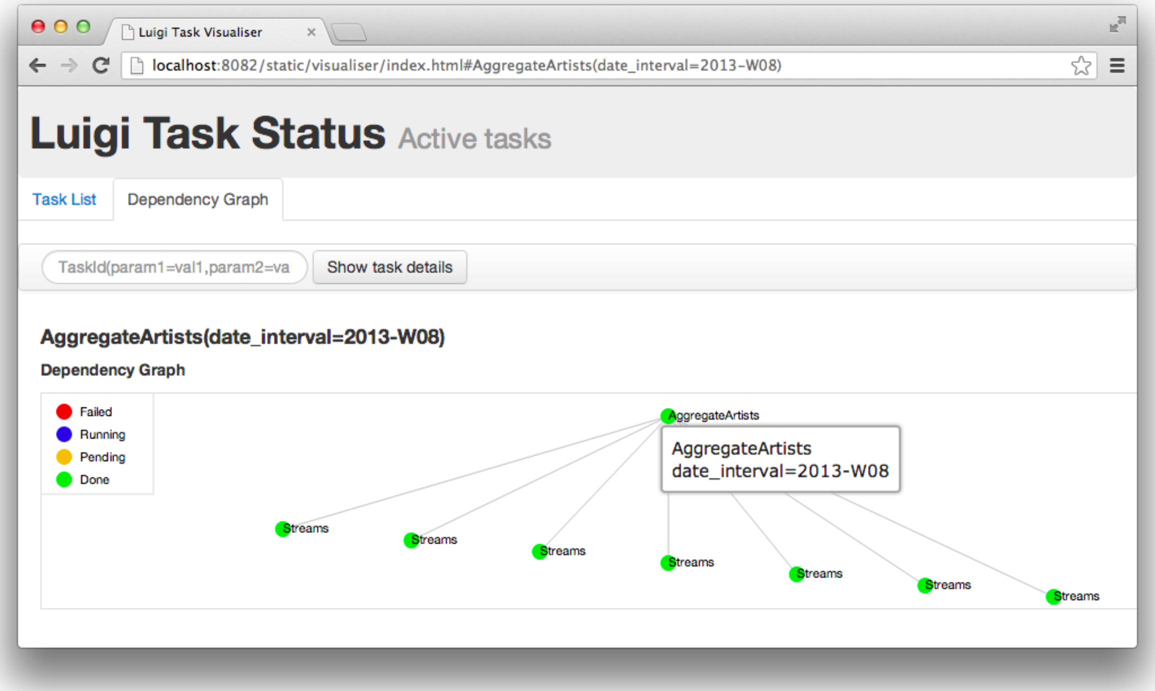
− 2 AggregateArtists

AggregateArtists (date_interval=2013-W08) 10:40:19 PM Dependency Graph

AggregateArtists (date_interval=2012-08) 10:39:50 PM Dependency Graph

+ 2 Artists

+ 38 Streams



Luigi Task Status Active tasks

AggregateArtists(date_interval=2013-W08)

Dependency Graph

10.2.1 Target

The *Target* class corresponds to a file on a disk, a file on HDFS or some kind of a checkpoint, like an entry in a database. Actually, the only method that Targets have to implement is the *exists* method which returns True if and only if the Target exists.

In practice, implementing Target subclasses is rarely needed. Luigi comes a toolbox of several useful Targets. In particular, *LocalTarget* and `HdfsTarget`, but there is also support for other file systems: *luigi.s3.S3Target*, *luigi.contrib.ssh.RemoteTarget*, `luigi.ftp.RemoteTarget`, *luigi.contrib.mysqldb.MySqlTarget*, `luigi.redshift.RedshiftTarget`, and several more.

Most of these targets, are file system-like. For instance, *LocalTarget* and `HdfsTarget` map to a file on the local drive or a file in HDFS. In addition these also wrap the underlying operations to make them atomic. They both implement the *open()* method which returns a stream object that could be read (`mode='r'`) from or written to (`mode='w'`).

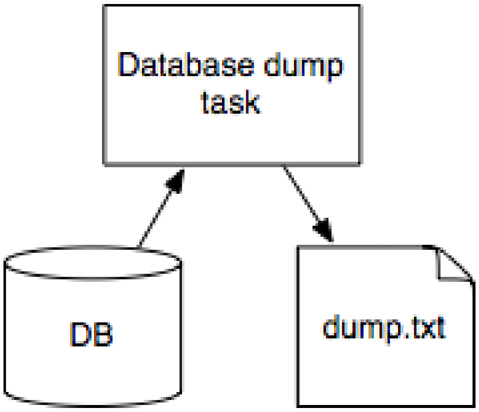
Luigi comes with Gzip support by providing `format=format.Gzip`. Adding support for other formats is pretty simple.

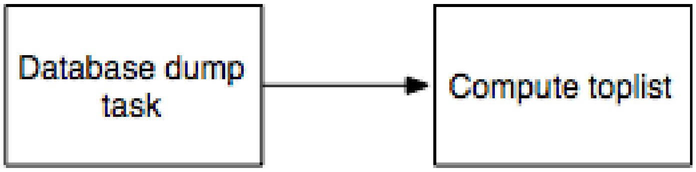
10.2.2 Task

The *Task* class is a bit more conceptually interesting because this is where computation is done. There are a few methods that can be implemented to alter its behavior, most notably *run()*, *output()* and *requires()*.

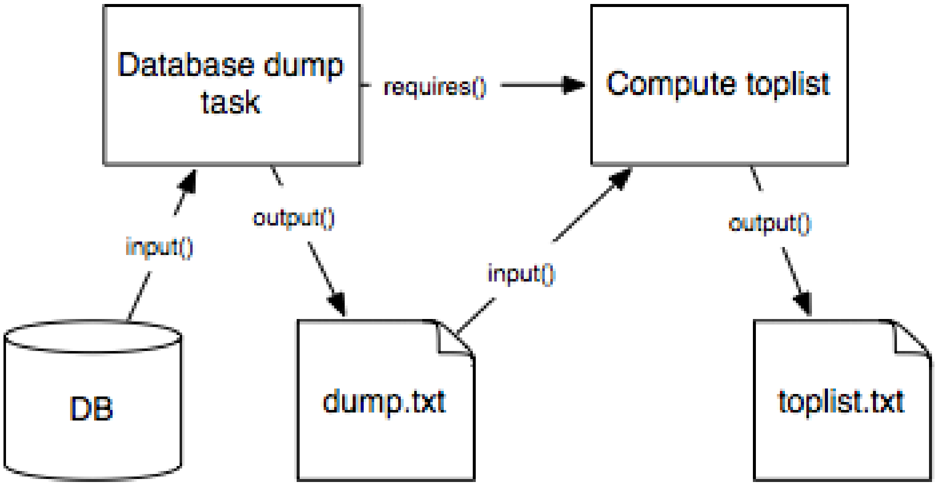
Tasks consume Targets that were created by some other task. They usually also output targets:

You can define dependencies between *Tasks* using the *requires()* method. See Tasks for more info.



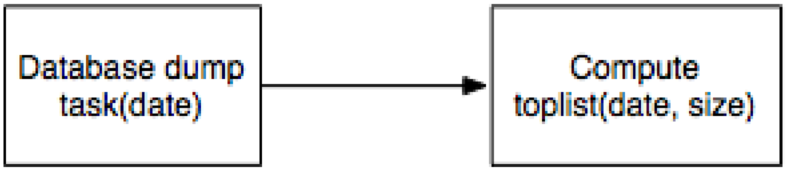


Each task defines its outputs using the `output()` method. Additionally, there is a helper method `input()` that returns the corresponding Target classes for each Task dependency.



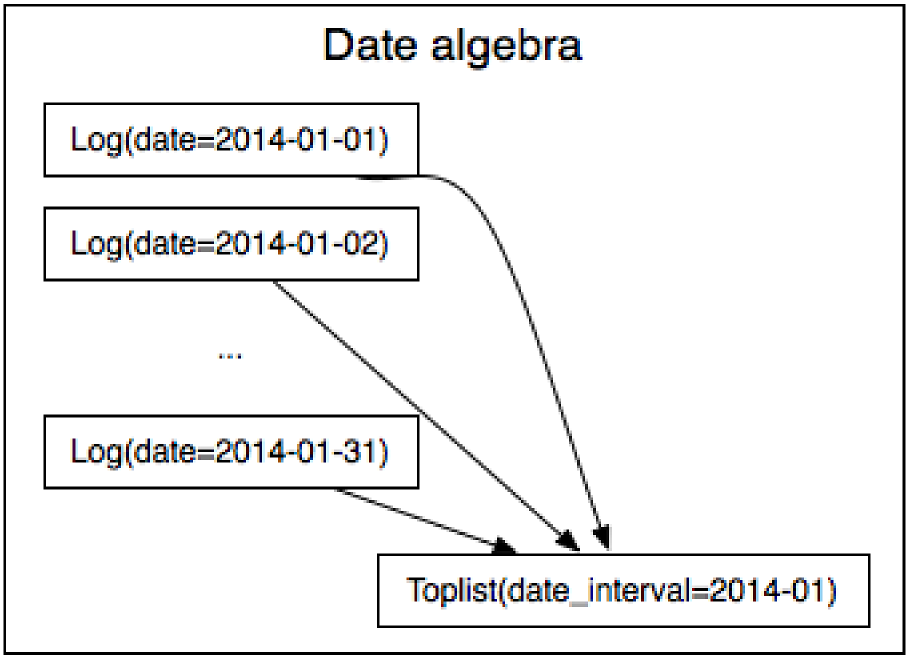
10.2.3 Parameter

The Task class corresponds to some type of job that is run, but in general you want to allow some form of parametrization of it. For instance, if your Task class runs a Hadoop job to create a report every night, you probably want to make the date a parameter of the class. See Parameters for more info.



10.2.4 Dependencies

Using tasks, targets, and parameters, Luigi lets you express arbitrary dependencies in *code*, rather than using some kind of awkward config DSL. This is really useful because in the real world, dependencies are often very messy. For instance, some examples of the dependencies you might encounter:



(These diagrams are from a Luigi presentation in late 2014 at NYC Data Science meetup)

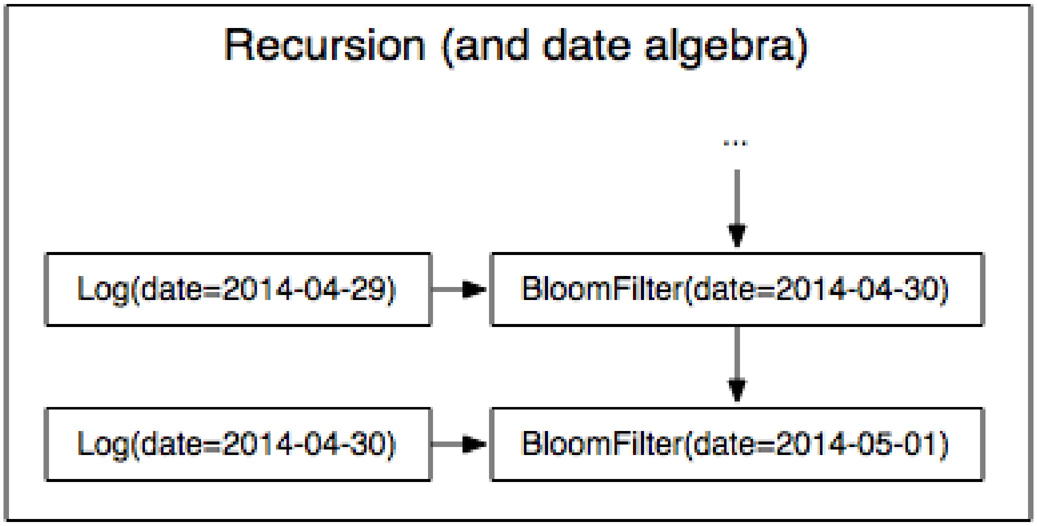
10.3 Tasks

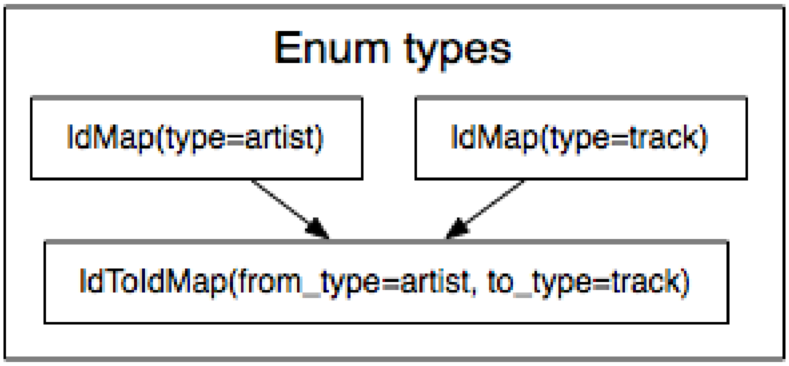
Tasks are where the execution takes place. Tasks depend on each other and output targets.

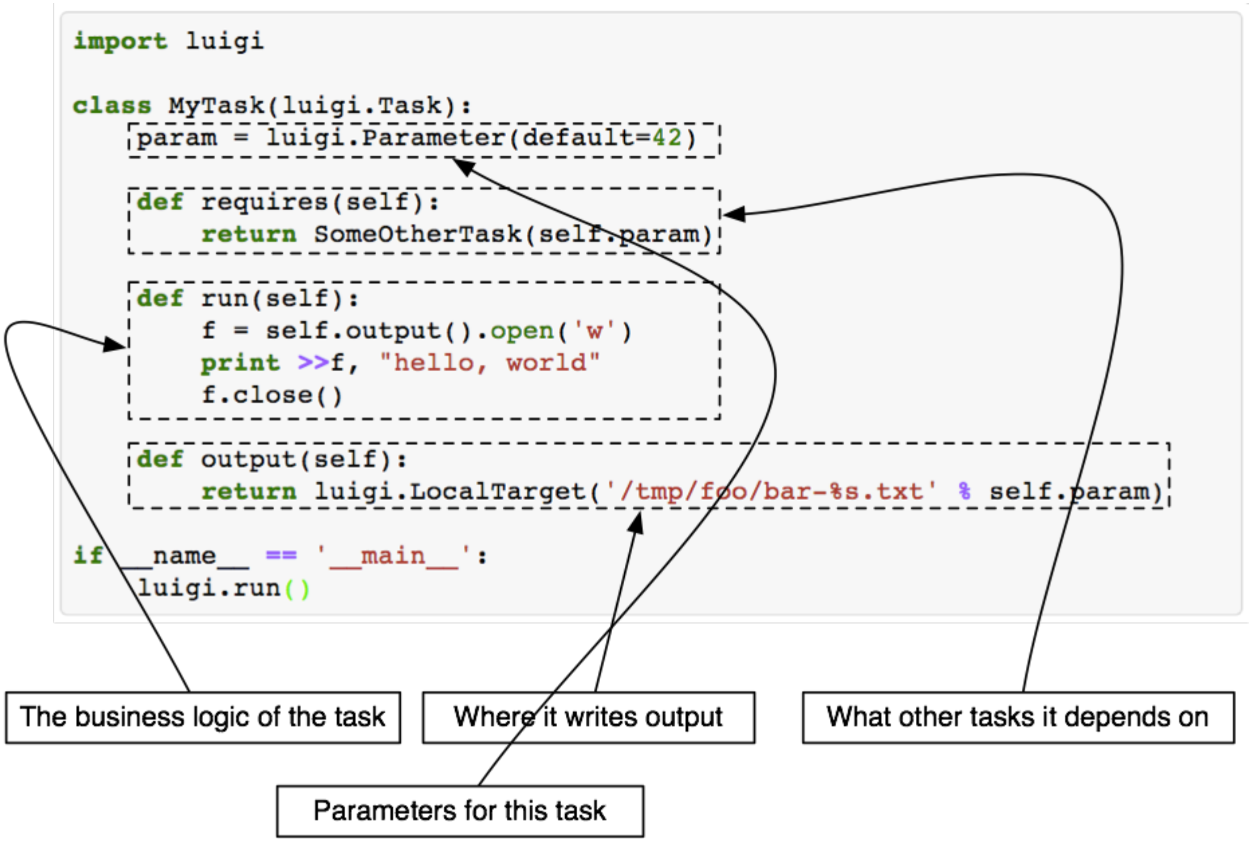
An outline of how a task can look like:

10.3.1 Task.requires

The `requires()` method is used to specify dependencies on other Task object, which might even be of the same class. For instance, an example implementation could be


Recursion (and date algebra)
...
Log(date=2014-04-29)
BloomFilter(date=2014-04-30)
Log(date=2014-04-30)
BloomFilter(date=2014-05-01)


Enum types
IdMap(type=artist)
IdMap(type=track)
IdToIdMap(from_type=artist, to_type=track)



```
import luigi

class MyTask(luigi.Task):
    param = luigi.Parameter(default=42)

    def requires(self):
        return SomeOtherTask(self.param)

    def run(self):
        f = self.output().open('w')
        print >>f, "hello, world"
        f.close()

    def output(self):
        return luigi.LocalTarget('/tmp/foo/bar-%s.txt' % self.param)

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

The business logic of the task
Where it writes output
What other tasks it depends on
Parameters for this task

```
def requires(self):
    return OtherTask(self.date), DailyReport(self.date - datetime.timedelta(1))
```

In this case, the DailyReport task depends on two inputs created earlier, one of which is the same class. requires can return other Tasks in any way wrapped up within dicts/lists/tuples/etc.

10.3.2 Requiring another Task

Note that `requires()` can *not* return a `Target` object. If you have a simple Target object that is created externally you can wrap it in a Task class like this:

```
class LogFiles(luigi.Task):
    def output(self):
        return luigi.contrib.hdfs.HdfsTarget('/log')
```

This also makes it easier to add parameters:

```
class LogFiles(luigi.Task):
    date = luigi.DateParameter()
    def output(self):
        return luigi.contrib.hdfs.HdfsTarget(self.date.strftime('/log/%Y-%m-%d'))
```

10.3.3 Task.output

The `output()` method returns one or more `Target` objects. Similarly to requires, can return wrap them up in any way that's convenient for you. However we recommend that any `Task` only return one single `Target` in output. If multiple outputs are returned, atomicity will be lost unless the `Task` itself can ensure that each `Target` is atomically created. (If atomicity is not of concern, then it is safe to return multiple `Target` objects.)

```
class DailyReport(luigi.Task):
    date = luigi.DateParameter()
    def output(self):
        return luigi.contrib.hdfs.HdfsTarget(self.date.strftime('/reports/%Y-%m-%d'))
    # ...
```

10.3.4 Task.run

The `run()` method now contains the actual code that is run. When you are using *Task.requires* and *Task.run* Luigi breaks down everything into two stages. First it figures out all dependencies between tasks, then it runs everything. The `input()` method is an internal helper method that just replaces all Task objects in requires with their corresponding output. An example:

```
class TaskA(luigi.Task):
    def output(self):
        return luigi.LocalTarget('xyz')

class FlipLinesBackwards(luigi.Task):
    def requires(self):
        return TaskA()

    def output(self):
        return luigi.LocalTarget('abc')

    def run(self):
        f = self.input().open('r') # this will return a file stream that reads from "xyz"
        g = self.output().open('w')
        for line in f:
            g.write('%s\n', ''.join(reversed(line.strip().split()))
        g.close() # needed because files are atomic
```

10.3.5 Task.input

As seen in the example above, `Task` is a wrapper around *Task.requires* that returns the corresponding Target objects instead of Task objects. Anything returned by *Task.requires* will be transformed, including lists, nested dicts, etc. This can be useful if you have many dependencies:

```
class TaskWithManyInputs(luigi.Task):
    def requires(self):
        return {'a': TaskA(), 'b': [TaskB(i) for i in xrange(100)]}

    def run(self):
        f = self.input()['a'].open('r')
        g = [y.open('r') for y in self.input()['b']]
```

10.3.6 Dynamic dependencies

Sometimes you might not know exactly what other tasks to depend on until runtime. In that case, Luigi provides a mechanism to specify dynamic dependencies. If you yield another `Task` in the *Task.run* method, the current task will be suspended and the other task will be run. You can also return a list of tasks.

```
class MyTask(luigi.Task):
    def run(self):
        other_target = yield OtherTask()

        # dynamic dependencies resolve into targets
        f = other_target.open('r')
```

This mechanism is an alternative to *Task.requires* in case you are not able to build up the full dependency graph before running the task. It does come with some constraints: the *Task.run* method will resume from scratch each time a new task is yielded. In other words, you should make sure your *Task.run* method is idempotent. (This is good practice for all Tasks in Luigi, but especially so for tasks with dynamic dependencies).

For an example of a workflow using dynamic dependencies, see examples/dynamic_requirements.py.

10.3.7 Events and callbacks

Luigi has a built-in event system that allows you to register callbacks to events and trigger them from your own tasks. You can both hook into some pre-defined events and create your own. Each event handle is tied to a Task class and will be triggered only from that class or a subclass of it. This allows you to effortlessly subscribe to events only from a specific class (e.g. for hadoop jobs).

```
@luigi.Task.event_handler(luigi.Event.SUCCESS)
def celebrate_success(task):
    """Will be called directly after a successful execution
       of `run` on any Task subclass (i.e. all luigi Tasks)
    """
    ...

@luigi.contrib.hadoop.JobTask.event_handler(luigi.Event.FAILURE)
def mourn_failure(task, exception):
    """Will be called directly after a failed execution
       of `run` on any JobTask subclass
    """
    ...

luigi.run()
```

10.3.8 But I just want to run a Hadoop job?

The Hadoop code is integrated in the rest of the Luigi code because we really believe almost all Hadoop jobs benefit from being part of some sort of workflow. However, in theory, nothing stops you from using the `JobTask` class (and also `HdfsTarget`) without using the rest of Luigi. You can simply run it manually using

```
MyJobTask('abc', 123).run()
```

You can use the hdfs.HdfsTarget class anywhere by just instantiating it:

```
t = luigi.contrib.hdfs.HdfsTarget('/tmp/test.gz', format=format.Gzip)
f = t.open('w')
# ...
f.close() # needed
```

10.3.9 Task priority

The scheduler decides which task to run next from the set of all task that have all their dependencies met. By default, this choice is pretty arbitrary, which is fine for most workflows and situations.

If you want to have some control on the order of execution of available tasks, you can set the `priority` property of a task, for example as follows:

```
# A static priority value as a class constant:
class MyTask(luigi.Task):
    priority = 100
    # ...

# A dynamic priority value with a "@property" decorated method:
class OtherTask(luigi.Task):
    @property
    def priority(self):
        if self.date > some_threshold:
            return 80
        else:
            return 40
    # ...
```

Tasks with a higher priority value will be picked before tasks with a lower priority value. There is no predefined range of priorities, you can choose whatever (int or float) values you want to use. The default value is 0.

Warning: task execution order in Luigi is influenced by both dependencies and priorities, but in Luigi dependencies come first. For example: if there is a task A with priority 1000 but still with unmet dependencies and a task B with priority 1 without any pending dependencies, task B will be picked first.

10.3.10 Instance caching

In addition to the stuff mentioned above, Luigi also does some metaclass logic so that if e.g. `DailyReport(datetime.date(2012, 5, 10))` is instantiated twice in the code, it will in fact result in the same object. See *Instance caching* for more info

10.4 Parameters

Parameters is the Luigi equivalent of creating a constructor for each Task. Luigi requires you to declare these parameters instantiating *`Parameter`* objects on the class scope:

```
class DailyReport(luigi.contrib.hadoop.JobTask):
    date = luigi.DateParameter(default=datetime.date.today())
    # ...
```

By doing this, Luigi can do take care of all the boilerplate code that would normally be needed in the constructor. Internally, the DailyReport object can now be constructed by running `DailyReport(datetime.date(2012, 5, 10))` or just `DailyReport()`. Luigi also creates a command line parser that automatically handles the conversion from strings to Python types. This way you can invoke the job on the command line eg. by passing `--date 2012-15-10`.

The parameters are all set to their values on the Task object instance, i.e.

```
d = DailyReport(datetime.date(2012, 5, 10))
print d.date
```

will return the same date that the object was constructed with. Same goes if you invoke Luigi on the command line.

10.4.1 Instance caching

Tasks are uniquely identified by their class name and values of their parameters. In fact, within the same worker, two tasks of the same class with parameters of the same values are not just equal, but the same instance:

```
>>> import luigi
>>> import datetime
>>> class DateTask(luigi.Task):
...   date = luigi.DateParameter()
...
>>> a = datetime.date(2014, 1, 21)
>>> b = datetime.date(2014, 1, 21)
>>> a is b
False
>>> c = DateTask(date=a)
>>> d = DateTask(date=b)
>>> c
DateTask(date=2014-01-21)
>>> d
DateTask(date=2014-01-21)
>>> c is d
True
```

10.4.2 Insignificant parameters

If a parameter is created with `significant=False`, it is ignored as far as the Task signature is concerned. Tasks created with only insignificant parameters differing have the same signature but are not the same instance:

```
>>> class DateTask2(DateTask):
...   other = luigi.Parameter(significant=False)
...
>>> c = DateTask2(date=a, other="foo")
>>> d = DateTask2(date=b, other="bar")
>>> c
DateTask2(date=2014-01-21)
>>> d
DateTask2(date=2014-01-21)
>>> c.other
'foo'
>>> d.other
'bar'
>>> c is d
False
>>> hash(c) == hash(d)
True
```

10.4.3 Parameter types

In the examples above, the *type* of the parameter is determined by using different subclasses of *`Parameter`*. There are a few of them, like *`DateParameter`*, *`DateIntervalParameter`*, *`IntParameter`*, *`FloatParameter`*, etc.

Python is not a strongly typed language and you don't have to specify the types of any of your parameters. You can simply use the base class `Parameter` if you don't care.

The reason you would use a subclass like `DateParameter` is that Luigi needs to know its type for the command line interaction. That's how it knows how to convert a string provided on the command line to the corresponding type (i.e. datetime.date instead of a string).

10.4.4 Setting parameter value for other classes

All parameters are also exposed on a class level on the command line interface. For instance, say you have classes TaskA and TaskB:

```
class TaskA(luigi.Task):
    x = luigi.Parameter()

class TaskB(luigi.Task):
    y = luigi.Parameter()
```

You can run `TaskB` on the command line: `python script.py TaskB --y 42`. But you can also set the class value of `TaskA` by running `python script.py TaskB --y 42 --TaskA-x 43`. This sets the value of `TaskA.x` to 43 on a *class* level. It is still possible to override it inside Python if you instantiate `TaskA(x=44)`.

All parameters can also be set from the configuration file. For instance, you can put this in the config:

```
[TaskA]
x: 45
```

Just as in the previous case, this will set the value of `TaskA.x` to 45 on the *class* level. And likewise, it is still possible to override it inside Python if you instantiate `TaskA(x=44)`.

10.4.5 Parameter resolution order

Parameters are resolved in the following order of decreasing priority:

1. Any value passed to the constructor, or task level value set on the command line (applies on an instance level)
2. Any value set on the command line (applies on a class level)
3. Any configuration option (applies on a class level)
4. Any default value provided to the parameter (applies on a class level)

See the `Parameter` class for more information.

10.5 Running from the Command Line

Any task can be instantiated and run from the command line:

```
import luigi

class MyTask(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=45)

    def run(self):
        print self.x + self.y
```

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

You can run this task from the command line like this:

```
$ python my_task.py MyTask --local-scheduler --x 123 --y 456
```

You can also pass `main_task_cls=MyTask` and `local_scheduler=True` to `luigi.run()` and that way you can invoke it simply using

```
$ python my_task.py --x 123 --y 456
```

The other way to run a Luigi task is to use the builtin *luigi* script. This will be default on your path and can be run by providing a module name. The module will imported dynamically:

```
$ luigi --module my_module MyTask --x 123 --y 456
```

10.6 Programmatic Execution

As seen above, command line integration is achieved by simply adding

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

This will read the args from the command line (using argparse) and invoke everything.

In case you just want to run a Luigi chain from a Python script, you can do that internally without the command line integration. The code will look something like

```
task = MyTask(123, 'xyz')
interface.setup_interface_logging()
sch = scheduler.CentralPlannerScheduler()
w = worker.Worker(scheduler=sch)
w.add(task)
w.run()
```

10.7 Using the Central Scheduler

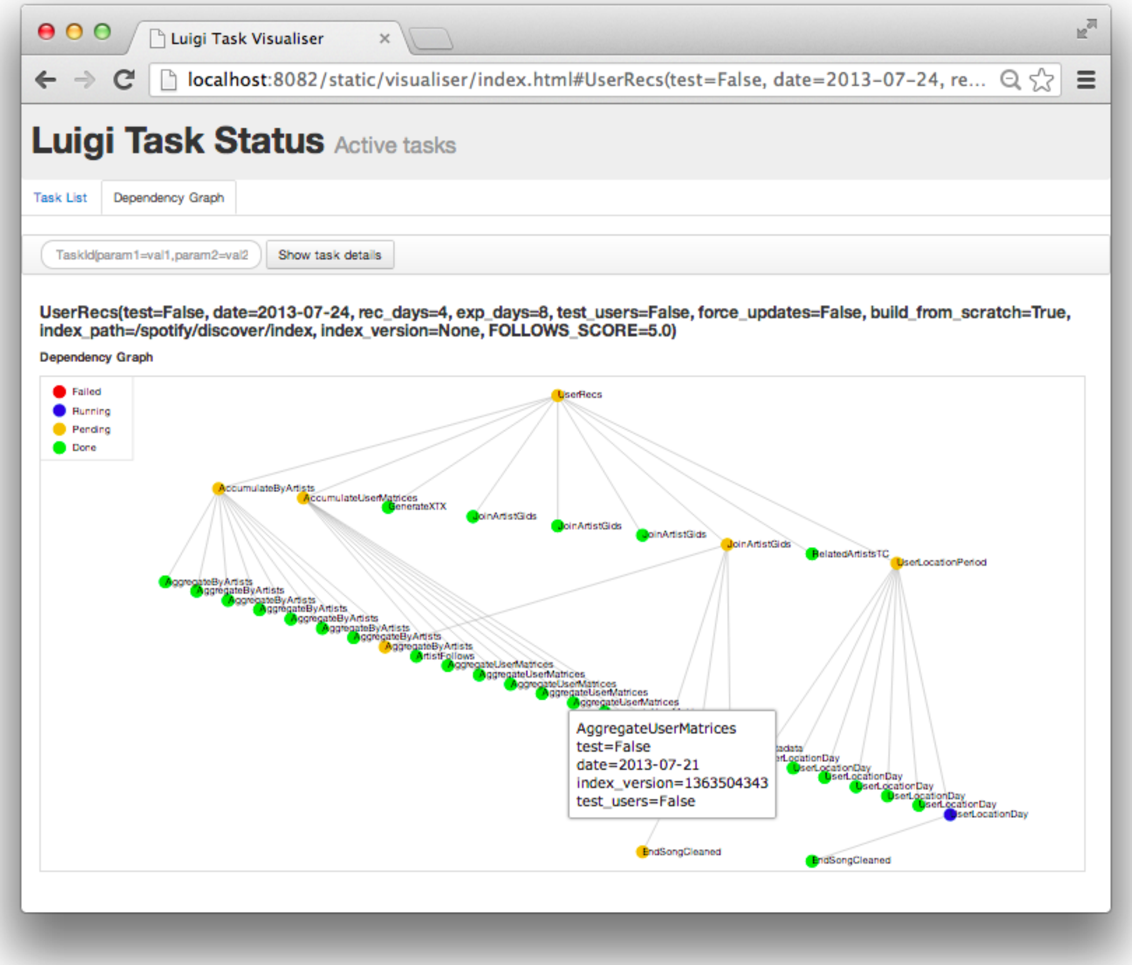
While the `--local-scheduler` flag is useful for development purposes, it's not recommended for production usage. The centralized scheduler services two purposes:

- Make sure two instances of the same task are not running simultaneously
- Provide visualization of everything that's going on.

Note that the central scheduler does not execute anything for you or help you with job parallelization. For running tasks periodically, the easiest thing to do is to trigger a Python script from cron or from a continuously running process. There is no central process that automatically triggers job. This model may seem limited, but we believe that it makes things far more intuitive and easy to understand.

10.7.1 The luigid server

To run the server as a daemon run:



Luigi Task Visualiser
localhost:8082/static/visualiser/index.html#UserRecs(test=False, date=2013-07-24, re...
Task List
Dependency Graph
TaskId|param1=val1,param2=val2
Show task details

Luigi Task Status Active tasks

UserRecs(test=False, date=2013-07-24, rec_days=4, exp_days=8, test_users=False, force_updates=False, build_from_scratch=True, index_path=/spotify/discover/index, index_version=None, FOLLOWS_SCORE=5.0)

Dependency Graph

● Failed
● Running
● Pending
● Done

AggregateUserMatrices
test=False
date=2013-07-21
index_version=1363504343
test_users=False

```
luigid --background --pidfile <PATH_TO_PIDFILE> --logdir <PATH_TO_LOGDIR> --state-path <PATH_TO_STAT
```

Note that this requires python-daemon. By default, the server starts on port 8082 (which can be changed with the --port flag) and listens on all IPs.

For a full list of configuration options and defaults, see the *scheduler configuration section*. Note that luigid uses the same configuration files as the luigi client (i.e. client.cfg or /etc/luigi/client.cfg by default).

10.7.2 Enabling Task History

Task History is an experimental feature in which additional information about tasks that have been executed are recorded in a relational database for historical analysis. This information is exposed via the Central Scheduler at /history.

To enable the task history, specify record_task_history = True in the [scheduler] section of client.cfg and specify db_connection under [task_history]. The db_connection string is to used to configure the SQLAlchemy engine. When starting up, luigid will create all the necessary tables using create_all.

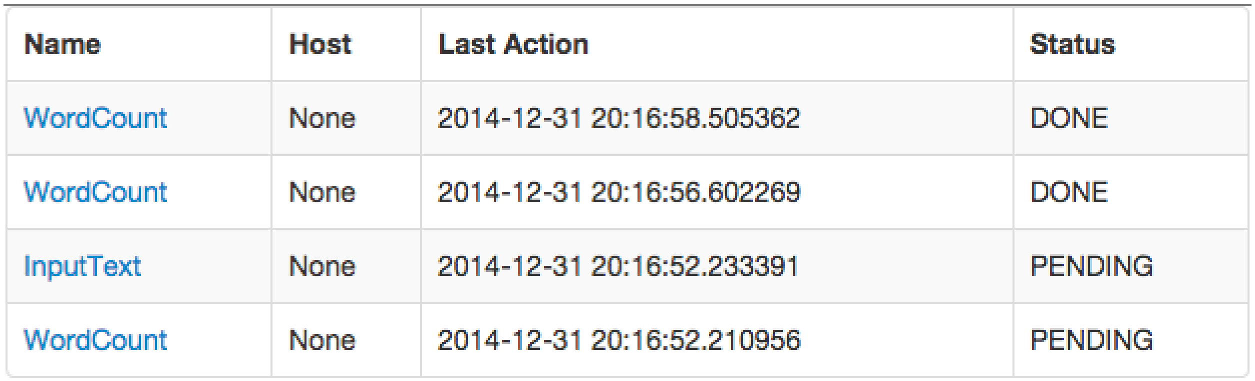
Example configuration:

```
[scheduler]
record_task_history = True
state-path = /usr/local/var/luigi-state.pickle

[task_history]
db_connection = sqlite:////usr/local/var/luigi-task-hist.db
```

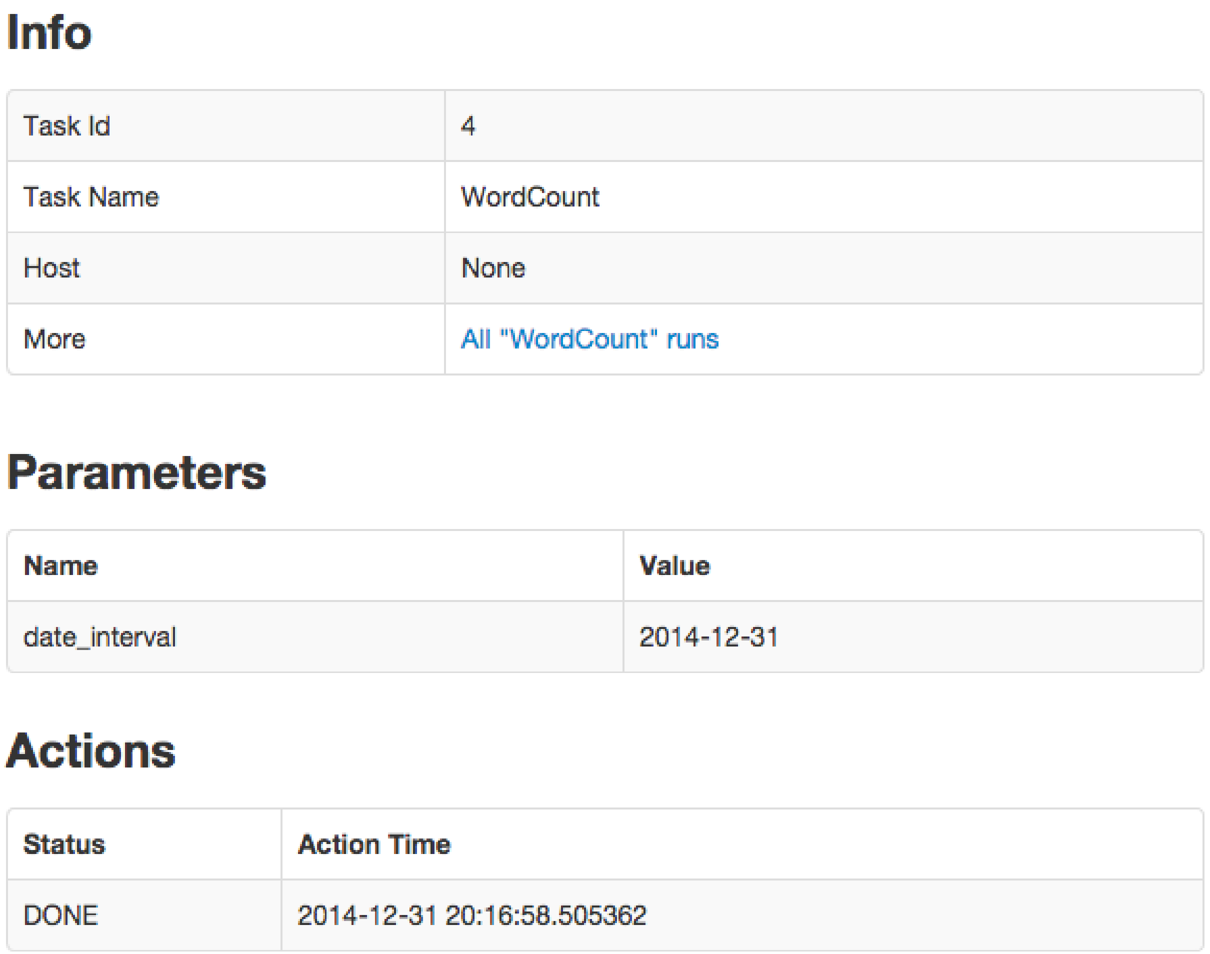
The task history has the following pages:

- /history a reverse-cronological listing of runs from the past 24 hours. Example screenshot:



Name	Host	Last Action	Status
WordCount	None	2014-12-31 20:16:58.505362	DONE
WordCount	None	2014-12-31 20:16:56.602269	DONE
InputText	None	2014-12-31 20:16:52.233391	PENDING
WordCount	None	2014-12-31 20:16:52.210956	PENDING

- /history/by_id/:id detailed information about a run, including: parameter values, the host on which it ran, and timing information. Example screenshot:
- /history/by_name/:name a listing of all runs of a task with the given task name. Example screenshot:
- /history/by_params/:name?data=params a listing of all runs of a given task restricted to runs with param values matching the given data. The data is a json blob describing the parameters, e.g. {"foo": "bar"} looks for a task with foo=bar.



Info

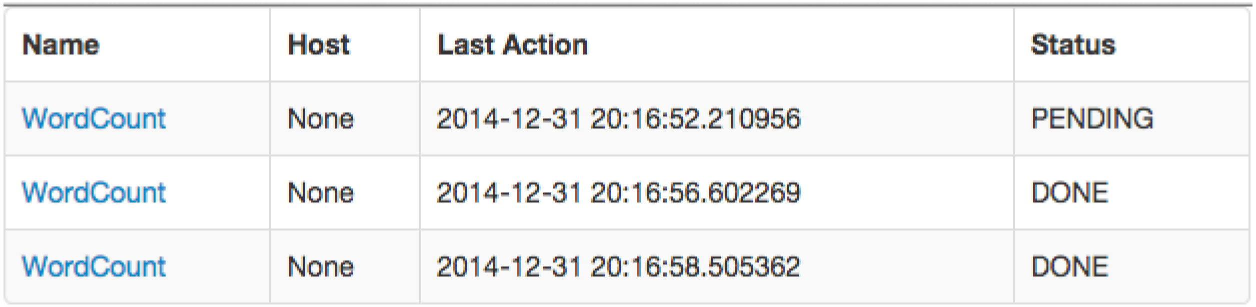
Task Id	4
Task Name	WordCount
Host	None
More	All "WordCount" runs

Parameters

Name	Value
date_interval	2014-12-31

Actions

Status	Action Time
DONE	2014-12-31 20:16:58.505362



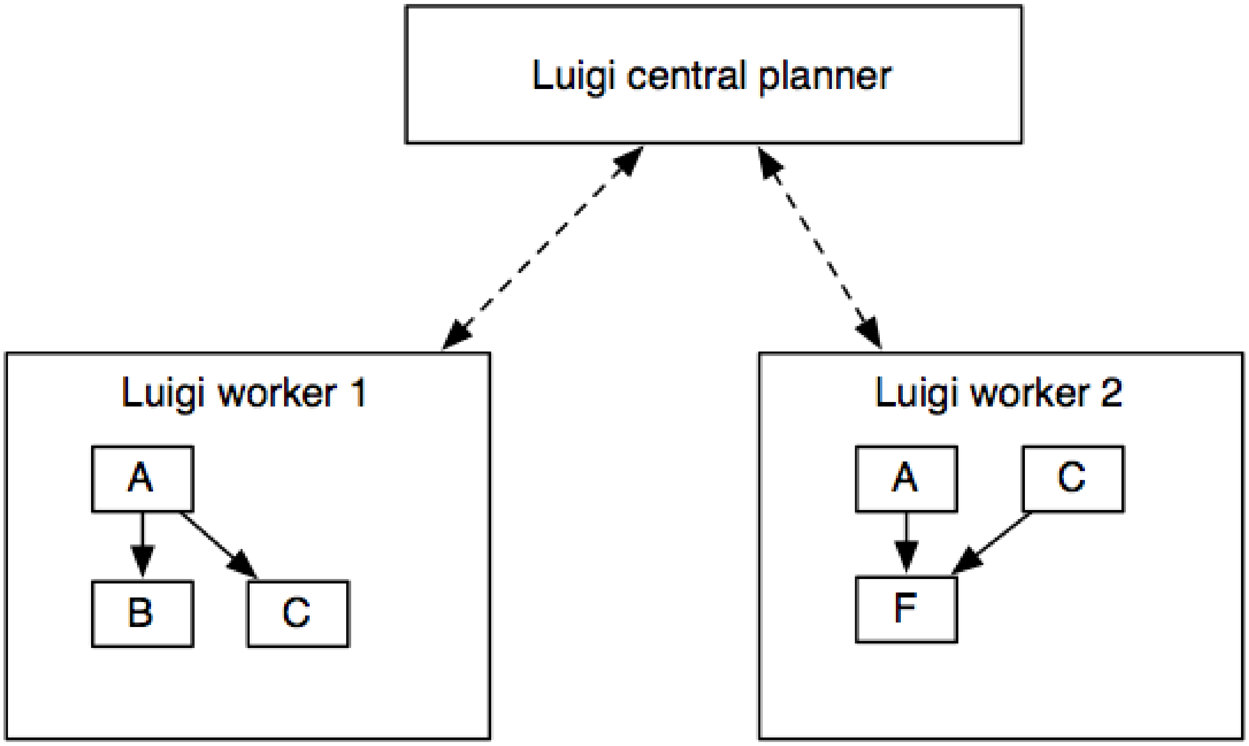
Name	Host	Last Action	Status
WordCount	None	2014-12-31 20:16:52.210956	PENDING
WordCount	None	2014-12-31 20:16:56.602269	DONE
WordCount	None	2014-12-31 20:16:58.505362	DONE

10.8 Execution Model

Luigi has a quite simple model for execution and triggering.

10.8.1 Workers and task execution

The most important aspect is that *no execution is transferred*. When you run a Luigi workflow, the worker schedules all tasks, and also executes the tasks within the process.



The benefit of this scheme is that it's super easy to debug since all execution takes place in the process. It also makes deployment a non-event. During development, you typically run the Luigi workflow from the command line, whereas when you deploy it, you can trigger it using crontab or any other scheduler.

The downside is that Luigi doesn't give you scalability for free. In practice this is not a problem until you start running thousands of tasks.

Isn't the point of Luigi to automate and schedule these workflows? To some extent. Luigi helps you *encode the dependencies* of tasks and build up chains. Furthermore, Luigi's scheduler makes sure that there's centralized view of the dependency graph and that the same job will not be executed by multiple workers simultaneously.

10.8.2 Triggering tasks

Luigi does not include its own triggering, so you have to rely on an external scheduler such as crontab to actually trigger the workflows.

In practice it's not a big hurdle because Luigi avoids all the mess typically caused by it. Scheduling a complex workflow is fairly trivial using eg. crontab.

In the future, Luigi might implement its own triggering. The dependency on crontab (or any external triggering mechanism) is a bit awkward and it would be nice to avoid.

Trigger example

For instance, if you have an external data dump that arrives every day and that your workflow depends on it, you write a workflow that depends on this data dump. Crontab can then trigger this workflow *every minute* to check if the data has arrived. If it has, it will run the full dependency graph.

```
class DataDump(luigi.ExternalTask):
    date = luigi.DateParameter()
    def output(self): return luigi.contrib.hdfs.HdfsTarget(self.date.strftime('/var/log/dump/%Y-%m-%

class AggregationTask(luigi.Task):
    date = luigi.DateParameter()
    window = luigi.IntParameter()
    def requires(self): return [DataDump(self.date - datetime.timedelta(i)) for i in xrange(self.wind
    def run(self): run_some_cool_stuff(self.input())
    def output(self): return luigi.contrib.hdfs.HdfsTarget('/aggregated-%s-%d' % (self.date, self.wi

class RunAll(luigi.Task):
    ''' Dummy task that triggers execution of a other tasks'''
    def requires(self):
        for window in [3, 7, 14]:
            for d in xrange(10): # guarantee that aggregations were run for the past 10 days
               yield AggregationTask(datetime.date.today() - datetime.timedelta(d), window)

if __name__ == '__main__':
    luigi.run(main_task_cls=RunAll)
```

You can trigger this as much as you want from crontab, and even across multiple machines, because the central scheduler will make sure at most one of each `AggregationTask` task is run simultaneously. Note that this might actually mean multiple tasks can be run because there are instances with different parameters, and this can gives you some form of parallelization (eg. `AggregationTask(2013-01-09)` might run in parallel with `AggregationTask(2013-01-08)`).

Of course, some Task types (eg. `HadoopJobTask`) can transfer execution to other places, but this is up to each Task to define.

10.9 Luigi Patterns

10.9.1 Code Reuse

One nice thing about Luigi is that it's super easy to depend on tasks defined in other repos. It's also trivial to have "forks" in the execution path, where the output of one task may become the input of many other tasks.

Currently no semantics for "intermediate" output is supported, meaning that all output will be persisted indefinitely. The upside of that is that if you try to run X -> Y, and Y crashes, you can resume with the previously built X. The downside is that you will have a lot of intermediate results on your file system. A useful pattern is to put these files in a special directory and have some kind of periodical garbage collection clean it up.

10.9.2 Triggering Many Tasks

A convenient pattern is to have a dummy Task at the end of several dependency chains, so you can trigger a multitude of pipelines by specifying just one task in command line, similarly to how e.g. make works.

```
class AllReports(luigi.Task):
    date = luigi.DateParameter(default=datetime.date.today())
    def requires(self):
        yield SomeReport(self.date)
        yield SomeOtherReport(self.date)
        yield CropReport(self.date)
        yield TPSReport(self.date)
        yield FooBarBazReport(self.date)
```

This simple task will not do anything itself, but will invoke a bunch of other tasks. Per each invocation Luigi will perform as many of the pending jobs as possible (those which have all their dependencies present).

10.9.3 Triggering recurring tasks

A common requirement is to have a daily report (or something else) produced every night. Sometimes for various reasons tasks will keep crashing or lacking their required dependencies for more than a day though, which would lead to a missing deliverable for some date. Oops.

To ensure that the above AllReports task is eventually completed for every day (value of date parameter), one could e.g. add a loop in requires method to yield dependencies on the past few days preceding self.date. Then, so long as Luigi keeps being invoked, the backlog of jobs would catch up nicely after fixing intermittent problems.

Luigi actually comes with a reusable tool for achieving this, called RangeDailyBase (resp. RangeHourlyBase). Simply putting

```
luigi --module all_reports RangeDailyBase --of AllReports --start 2015-01-01
```

in your crontab will easily keep gaps from occurring from 2015-01-01 onwards. NB - it will not always loop over everything from 2015-01-01 till current time though, but rather a maximum of 3 months ago by default - see RangeDailyBase documentation for this and more knobs for tweaking behavior. See also Monitoring below.

10.9.4 Efficiently triggering recurring tasks

RangeDailyBase, described above, is named like that because a more efficient subclass exists, RangeDaily (resp. RangeHourly), tailored for hundreds of task classes scheduled concurrently with contiguousness requirements spanning years (which would incur redundant completeness checks and scheduler overload using the naive looping approach.) Usage:

```
luigi --module all_reports RangeDaily --of AllReports --start 2015-01-01
```

It has the same knobs as RangeDailyBase, with some added requirements. Namely the task must implement an efficient bulk_complete method, or must be writing output to file system Target with date parameter value consistently represented in the file path.

10.9.5 Backfilling tasks

Also a common use case, sometimes you have tweaked existing recurring task code and you want to schedule recomputation of it over an interval of dates for that or another reason. Most conveniently it is achieved with the above described range tools, just with both start (inclusive) and stop (exclusive) parameters specified:

```
luigi --module all_reports RangeDaily --of AllReportsV2 --start 2014-10-31 --stop 2014-12-25
```

10.9.6 Monitoring task pipelines

Set error-email in Configuration to receive notifications whenever tasks crash. (This can get noisy with growing numbers of tasks and intermittent failures.)

The above mentioned range tools for recurring tasks not only implement reliable scheduling for you, but also emit events which you can use to set up delay monitoring. That way you can implement alerts for when jobs are stuck for prolonged periods lacking input data or otherwise requiring attention.

10.10 Configuration

All configuration can be done by adding a configuration files. They are looked for in:

- `/etc/luigi/client.cfg`
- `client.cfg` in your current working directory
- `LUIGI_CONFIG_PATH` environment variable

in increasing order of preference. The order only matters in case of key conflicts (see docs for ConfigParser)

The config file is broken into sections, each controlling a different part of the config. Example configuration file:

```
[hadoop]
version: cdh4
streaming-jar: /usr/lib/hadoop-xyz/hadoop-streaming-xyz-123.jar

[core]
default-scheduler-host: luigi-host.mycompany.foo
error-email: foo@bar.baz
```

By default, all parameters will be overridden by matching values in the configuration file. For instance if you have a Task definition:

```
class DailyReport(luigi.contrib.hadoop.JobTask):
    date = luigi.DateParameter(default=datetime.date.today())
    # ...
```

Then you can override the default value for date by providing it in the configuration:

```
[DailyReport]
date: 2012-01-01
```

You can also use `config_path` as an argument to the `Parameter` if you want to use a specific section in the config.

10.10.1 Configurable options

Luigi comes with a lot of configurable options. Below, we describe each section and the parameters available within it.

10.10.2 [core]

These parameters control core luigi behavior, such as error e-mails and interactions between the worker and scheduler.

default-scheduler-host Hostname of the machine running the scheduler. Defaults to localhost.

default-scheduler-port Port of the remote scheduler api process. Defaults to 8082.

email-prefix Optional prefix to add to the subject line of all e-mails. For example, setting this to "[LUIGI]" would change the subject line of an e-mail from "Luigi: Framework error" to "[LUIGI] Luigi: Framework error"

email-sender User name in from field of error e-mails. Default value: luigi-client@<server_name>

email-type Type of e-mail to send. Valid values are "plain" and "html". When set to html, tracebacks are wrapped in <pre> tags to get fixed-width font. Default value is plain.

error-email Recipient of all error e-mails. If this is not set, no error e-mails are sent when luigi crashes. If luigi is run from the command line, no e-mails will be sent unless output is redirected to a file.

hdfs-tmp-dir Base directory in which to store temporary files on hdfs. Defaults to tempfile.gettempdir()

history-filename If set, specifies a filename for Luigi to write stuff (currently just job id) to in mapreduce job's output directory. Useful in a configuration where no history is stored in the output directory by Hadoop.

logging_conf_file Location of the logging configuration file.

max-reschedules The maximum number of times that a job can be automatically rescheduled by a worker before it will stop trying. Workers will reschedule a job if it is found to not be done when attempting to run a dependent job. This defaults to 1.

max-shown-tasks New in version 1.0.20.

The maximum number of tasks returned in a task_list api call. This will restrict the number of tasks shown in any section in the visualiser. Small values can alleviate frozen browsers when there are too many done tasks. This defaults to 100000 (one hundred thousand).

no_configure_logging If true, logging is not configured. Defaults to false.

parallel-scheduling If true, the scheduler will compute complete functions of tasks in parallel using multiprocessing. This can significantly speed up scheduling, but requires that all tasks can be pickled.

retry-external-tasks If true, incomplete external tasks (i.e. tasks where the *run()* method is NotImplemented) will be retested for completion while Luigi is running. This means that if external dependencies are satisfied after a workflow has started, any tasks dependent on that resource will be eligible for running. Note: Every time the task remains incomplete, it will count as FAILED, so normal retry logic applies (see: *disable-num-failures* and *retry-delay*). This setting works best with *worker-keep-alive: true*. If false, external tasks will only be evaluated when Luigi is first invoked. In this case, Luigi will not check whether external dependencies are satisfied while a workflow is in progress, so dependent tasks will remain PENDING until the workflow is reinvoked. Defaults to false for backwards compatibility.

rpc-connect-timeout Number of seconds to wait before timing out when making an API call. Defaults to 10.0

smtp_host Hostname for sending mail throug smtp. Defaults to localhost.

smtp_local_hostname If specified, overrides the FQDN of localhost in the HELO/EHLO command.

smtp_login Username to log in to your smtp server, if necessary.

smtp_password Password to log in to your smtp server. Must be specified for smtp_login to have an effect.

smtp_port Port number for smtp on smtp_host. Defaults to 0.

smtp_ssl If true, connects to smtp through SSL. Defaults to false.

smtp_timeout Optionally sets the number of seconds after which smtp attempts should time out.

tmp-dir DEPRECATED - use hdfs-tmp-dir instead

worker-count-uniques If true, workers will only count unique pending jobs when deciding whether to stay alive. So if a worker can't get a job to run and other workers are waiting on all of its pending jobs, the worker will die. worker-keep-alive must be true for this to have any effect. Defaults to false.

worker-keep-alive If true, workers will stay alive when they run out of jobs to run, as long as they have some pending job waiting to be run. Defaults to false.

worker-ping-interval Number of seconds to wait between pinging scheduler to let it know that the worker is still alive. Defaults to 1.0.

worker-task-limit New in version 1.0.25.

Maximum number of tasks to schedule per invocation. Upon exceeding it, the worker will issue a warning and proceed with the workflow obtained thus far. Prevents incidents due to spamming of the scheduler, usually accidental. Default: no limit.

worker-timeout New in version 1.0.20.

Number of seconds after which to kill a task which has been running for too long. This provides a default value for all tasks, which can be overridden by setting the worker-timeout property in any task. This only works when using multiple workers, as the timeout is implemented by killing worker subprocesses. Default value is 0, meaning no timeout.

worker-wait-interval Number of seconds for the worker to wait before asking the scheduler for another job after the scheduler has said that it does not have any available jobs.

10.10.3 [elasticsearch]

These parameters control use of elasticsearch

marker-index Defaults to "update_log".

marker-doc-type Defaults to "entry".

10.10.4 [email]

These parameters control sending error e-mails through Amazon SES.

AWS_ACCESS_KEY Your AWS access key

AWS_SECRET_KEY Your AWS secret key

region Your AWS region. Defaults to us-east-1.

type If set to "ses", error e-mails will be send through Amazon SES. Otherwise, e-mails are sent via smtp.

10.10.5 [hadoop]

Parameters controlling basic hadoop tasks

command Name of command for running hadoop from the command line. Defaults to "hadoop"

python-executable Name of command for running python from the command line. Defaults to "python"

scheduler Type of scheduler to use when scheduling hadoop jobs. Can be "fair" or "capacity". Defaults to "fair".

streaming-jar Path to your streaming jar. Must be specified to run streaming jobs.

version Version of hadoop used in your cluster. Can be "cdh3", "chd4", or "apache1". Defaults to "cdh4".

10.10.6 [hdfs]

Parameters controlling the use of snakebite to speed up hdfs queries.

client Client to use for most hadoop commands. Options are “snakebite”, “snakebite_with_hadoopcli_fallback”, and “hadoopcli”. Snakebite is much faster, so use of it is encouraged. Using snakebite requires it to be installed separately on the machine. Defaults to “hadoopcli”.

client_version Optionally specifies hadoop client version for snakebite.

effective_user Optionally specifies the effective user for snakebite.

namenode_host The hostname of the namenode. Needed for snakebite if snakebite_autoconfig is not set.

namenode_port The port used by snakebite on the namenode. Needed for snakebite if snakebite_autoconfig is not set.

snakebite_autoconfig If true, attempts to automatically detect the host and port of the namenode for snakebite queries. Defaults to false.

10.10.7 [hive]

Parameters controlling hive tasks

command Name of the command used to run hive on the command line. Defaults to “hive”.

hiverc-location Optional path to hive rc file.

metastore_host Hostname for metastore.

metastore_port Port for hive to connect to metastore host.

release If set to “apache”, uses a hive client that better handles apache hive output. All other values use the standard client Defaults to “cdh4”.

10.10.8 [mysql]

Parameters controlling use of MySQL targets

marker-table Table in which to store status of table updates. This table will be created if it doesn’t already exist. Defaults to “table_updates”.

10.10.9 [postgres]

Parameters controlling the use of Postgres targets

local-tmp-dir Directory in which to temporarily store data before writing to postgres. Uses system default if not specified.

marker-table Table in which to store status of table updates. This table will be created if it doesn’t already exist. Defaults to “table_updates”.

10.10.10 [redshift]

Parameters controlling the use of Redshift targets

marker-table Table in which to store status of table updates. This table will be created if it doesn’t already exist. Defaults to “table_updates”.

10.10.11 [resources]

This section can contain arbitrary keys. Each of these specifies the amount of a global resource that the scheduler can allow workers to use. The scheduler will prevent running jobs with resources specified from exceeding the counts in this section. Unspecified resources are assumed to have limit 1. Example resources section for a configuration with 2 hive resources and 1 mysql resource:

```
[resources]
hive: 2
mysql: 1
```

Note that it was not necessary to specify the 1 for mysql here, but it is good practice to do so when you have a fixed set of resources.

10.10.12 [scalding]

Parameters controlling running of scalding jobs

scala-home Home directory for scala on your machine. Defaults to either SCALA_HOME or /usr/share/scala if SCALA_HOME is unset.

scalding-home Home directory for scalding on your machine. Defaults to either SCALDING_HOME or /usr/share/scalding if SCALDING_HOME is unset.

scalding-provided Provided directory for scalding on your machine. Defaults to either SCALDING_HOME/provided or /usr/share/scalding/provided

scalding-libjars Libjars directory for scalding on your machine. Defaults to either SCALDING_HOME/libjars or /usr/share/scalding/libjars

10.10.13 [scheduler]

Parameters controlling scheduler behavior

disable-hard-timeout Hard time limit after which tasks will be disabled by the server if they fail again, in seconds. It will disable the task if it fails **again** after this amount of time. E.g. if this was set to 600 (i.e. 10 minutes), and the task first failed at 10:00am, the task would be disabled if it failed again any time after 10:10am. Note: This setting does not consider the values of the *disable-num-failures* or *disable-window-seconds* settings.

disable-num-failures Number of times a task can fail within disable-window-seconds before the scheduler will automatically disable it. If not set, the scheduler will not automatically disable jobs.

disable-persist-seconds Number of seconds for which an automatic scheduler disable lasts. Defaults to 86400 (1 day).

disable-window-seconds Number of seconds during which disable-num-failures failures must occur in order for an automatic disable by the scheduler. The scheduler forgets about disables that have occurred longer ago than this amount of time. Defaults to 3600 (1 hour).

record_task_history If true, stores task history in a database. Defaults to false.

remove-delay Number of seconds to wait before removing a task that has no stakeholders. Defaults to 600 (10 minutes).

retry-delay Number of seconds to wait after a task failure to mark it pending again. Defaults to 900 (15 minutes).

state-path Path in which to store the luigi scheduler's state. When the scheduler is shut down, its state is stored in this path. The scheduler must be shut down cleanly for this to work, usually with a kill command. If the kill command includes the -9 flag, the scheduler will not be able to save its state. When the scheduler is started, it will load the state from this path if it exists. This will restore all scheduled jobs and other state from when the scheduler last shut down.

Sometimes this path must be deleted when restarting the scheduler after upgrading luigi, as old state files can become incompatible with the new scheduler. When this happens, all workers should be restarted after the scheduler both to become compatible with the updated code and to reschedule the jobs that the scheduler has now forgotten about.

This defaults to /var/lib/luigi-server/state.pickle

worker-disconnect-delay Number of seconds to wait after a worker has stopped pinging the scheduler before removing it and marking all of its running tasks as failed. Defaults to 60.

10.10.14 [spark]

Parameters controlling the default execution of `SparkSubmitTask` and `PySparkTask`:

Deprecated since version 1.1.1: `SparkJob`, `Spark1xJob` and `PySpark1xJob` are deprecated. Please use `SparkSubmitTask` or `PySparkTask`.

spark-submit Command to run in order to submit spark jobs. Default: spark-submit

master Master url to use for spark-submit. Example: local[*], spark://masterhost:7077. Default: Spark default (Prior to 1.1.1: yarn-client)

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

jars Comma-separated list of local jars to include on the driver and executor classpaths. Default: Spark default

py-files Comma-separated list of .zip, .egg, or .py files to place on the PYTHONPATH for Python apps. Default: Spark default

files Comma-separated list of files to be placed in the working directory of each executor. Default: Spark default

conf: Arbitrary Spark configuration property in the form Prop=Value|Prop2=Value2. Default: Spark default

properties-file Path to a file from which to load extra properties. Default: Spark default

driver-memory Memory for driver (e.g. 1000M, 2G). Default: Spark default

driver-java-options Extra Java options to pass to the driver. Default: Spark default

driver-library-path Extra library path entries to pass to the driver. Default: Spark default

driver-class-path Extra class path entries to pass to the driver. Default: Spark default

executor-memory Memory per executor (e.g. 1000M, 2G). Default: Spark default

Configuration for Spark submit jobs on Spark standalone with cluster deploy mode only:

driver-cores Cores for driver. Default: Spark default

supervise If given, restarts the driver on failure. Default: Spark default

Configuration for Spark submit jobs on Spark standalone and Mesos only:

total-executor-cores Total cores for all executors. Default: Spark default

Configuration for Spark submit jobs on YARN only:

executor-cores Number of cores per executor. Default: Spark default

queue The YARN queue to submit to. Default: Spark default

num-executors Number of executors to launch. Default: Spark default

archives Comma separated list of archives to be extracted into the working directory of each executor. Default: Spark default

hadoop-conf-dir Location of the hadoop conf dir. Sets HADOOP_CONF_DIR environment variable when running spark. Example: /etc/hadoop/conf

Extra configuration for PySparkTask jobs:

py-packages Comma-separated list of local packages (in your python path) to be distributed to the cluster.

Parameters controlling the execution of SparkJob jobs (deprecated):

spark-jar Location of the spark jar. Sets SPARK_JAR environment variable when running spark. Example: /usr/share/spark/jars/spark-assembly-0.8.1-incubating-hadoop2.2.0.jar

spark-class Location of script to invoke. Example: /usr/share/spark/spark-class

10.10.15 [task_history]

Parameters controlling storage of task history in a database

db_connection Connection string for connecting to the task history db using sqlalchemy.

10.11 More Info

Luigi is the successor to a couple of attempts that we weren't fully happy with. We learned a lot from our mistakes and some design decisions include:

- Straightforward command line integration.
- As little boilerplate as possible.
- Focus on job scheduling and dependency resolution, not a particular platform. In particular this means no limitation to Hadoop. Though Hadoop/HDFS support is built-in and is easy to use, this is just one of many types of things you can run.
- A file system abstraction where code doesn't have to care about where files are located.
- Atomic file system operations through this abstraction. If a task crashes it won't lead to a broken state.
- The dependencies are decentralized. No big config file in XML. Each task just specifies which inputs it needs and cross-module dependencies are trivial.
- A web server that renders the dependency graph and does locking etc for free.
- Trivial to extend with new file systems, file formats and job types. You can easily write jobs that inserts a Tokyo Cabinet into Cassandra. Adding support for new systems is generally not very hard. (Feel free to send us a patch when you're done!)
- Date algebra included.
- Lots of unit tests of the most basic stuff

It wouldn't be fair not to mention some limitations with the current design:

- Its focus is on batch processing so it's probably less useful for near real-time pipelines or continuously running processes.
- The assumption is that a each task is a sizable chunk of work. While you can probably schedule a few thousand jobs, it's not meant to scale beyond tens of thousands.

- Luigi does not support distribution of execution. When you have workers running thousands of jobs daily, this starts to matter, because the worker nodes get overloaded. There are some ways to mitigate this (trigger from many nodes, use resources), but none of them is ideal
- Luigi does not come with built-in triggering, and you still need to rely on something like crontab to trigger workflows periodically.

Also it should be mentioned that Luigi is named after the world's second most famous plumber.

10.11.1 Want to Contribute?

Awesome! Let us know if you have any ideas. Feel free to contact x@y.com where x = luigi and y = spotify.

10.11.2 Running Unit Tests

You can see in `.travis.yml` how Travis CI runs the tests. Essentially, what you do is first `pip install tox`, then you can run any of these examples and change them to your needs.

```
# Run all nonhdfs tests
export TOX_ENV=nonhdfs; export PYTHONPATH=''; tox -e $TOX_ENV test

# Run specific nonhdfs tests
export TOX_ENV=nonhdfs; export PYTHONPATH=''; tox -e $TOX_ENV test/test_ssh.py

# Run specific hdp tests with hdp hadoop distrubtion
export TOX_ENV=hdp; export PYTHONPATH=''; JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64 tox -e $TOX
```

10.11.3 Future Ideas

- S3/EC2 - We have some old ugly code based on Boto that could be integrated in a day or two.
- Built in support for Pig/Hive.
- Better visualization tool - the layout gets pretty messy as the number of tasks grows.
- Integration with existing Hadoop frameworks like mrjob would be cool and probably pretty easy.
- Better support (without much boilerplate) for unittesting specific Tasks

CHAPTER 11

API Reference

luigi	Package containing core luigi functionality.
luigi.contrib	Package containing optional and-on functionality.
luigi.tools	Sort of a standard library for doing stuff with Tasks at a somewhat abstract level.

11.1 luigi package

11.1.1 Subpackages

luigi.contrib package

Subpackages

luigi.contrib.hdfs package

Submodules

luigi.contrib.hdfs.abstract_client module Module containing abstract class about hdfs clients.

class `luigi.contrib.hdfs.abstract_client.`**`HdfsFileSystem`**

Bases: *luigi.target.FileSystem*

This client uses Apache 2.x syntax for file system commands, which also matched CDH4.

`rename`(*path*, *dest*)

Rename or move a file

`rename_dont_move`(*path*, *dest*)

Override this method with an implementation that uses rename2, which is a rename operation that never moves.

For instance, *rename2 a b* never moves *a* into *b* folder.

Currently, the hadoop cli does not support this operation.

We keep the interface simple by just aliasing this to normal rename and let individual implementations redefine the method.

rename2 - https://github.com/apache/hadoop/blob/ae91b13/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/protocol/ClientProtocol.java (lines 483-523)

remove (*path*, *recursive=True*, *skip_trash=False*)

chmod (*path*, *permissions*, *recursive=False*)

chown (*path*, *owner*, *group*, *recursive=False*)

count (*path*)
Count contents in a directory

copy (*path*, *destination*)

put (*local_path*, *destination*)

get (*path*, *local_destination*)

mkdir (*path*, *parents=True*, *raise_if_exists=False*)

listdir (*path*, *ignore_directories=False*, *ignore_files=False*, *include_size=False*, *include_type=False*, *include_time=False*, *recursive=False*)

touchz (*path*)

luigi.contrib.hdfs.clients module The implementations of the hdfs clients. The hadoop cli client and the snakebite client.

luigi.contrib.hdfs.clients.**get_autoconfig_client** (*show_warnings=True*)
Creates the client as specified in the *client.cfg* configuration.

luigi.contrib.hdfs.config module You can configure what client by setting the "client" config under the "hdfs" section in the configuration, or using the `--hdfs-client` command line option. "hadoopcli" is the slowest, but should work out of the box. "snakebite" is the fastest, but requires Snakebite to be installed.

class luigi.contrib.hdfs.config.**hdfs** (**args*, ***kwargs*)
Bases: *luigi.task.Config*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

client_version = <luigi.parameter.IntParameter object>

effective_user = <luigi.parameter.Parameter object>

snakebite_autoconfig = <luigi.parameter.BoolParameter object>

namenode_host = <luigi.parameter.Parameter object>

namenode_port = <luigi.parameter.IntParameter object>

client = <luigi.parameter.Parameter object>

tmp_dir = <luigi.parameter.Parameter object>

task_namespace = None

class luigi.contrib.hdfs.config.**hadoopcli** (**args*, ***kwargs*)
Bases: *luigi.task.Config*

Constructor to resolve values for all Parameters.

For example, the Task:

```
    class MyTask(luigi.Task):
        count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`command` = **<luigi.parameter.Parameter object>**

`version` = **<luigi.parameter.Parameter object>**

`task_namespace` = **None**

`luigi.contrib.hdfs.config.`**`load_hadoop_cmd`**`()`

`luigi.contrib.hdfs.config.`**`get_configured_hadoop_version`**`()`

CDH4 (hadoop 2+) has a slightly different syntax for interacting with hdfs via the command line.

The default version is CDH4, but one can override this setting with “cdh3” or “apache1” in the hadoop section of the config in order to use the old syntax.

`luigi.contrib.hdfs.config.`**`get_configured_hdfs_client`**`(`*show_warnings=True*`)`

This is a helper that fetches the configuration value for ‘client’ in the [hdfs] section. It will return the client that retains backwards compatibility when ‘client’ isn’t configured.

`luigi.contrib.hdfs.config.`**`tmppath`**`(`*path=None*, *include_unix_username=True*`)`

@param path: target path for which it is needed to generate temporary location @type path: str @type include_unix_username: bool @rtype: str

Note that include_unix_username might work on windows too.

luigi.contrib.hdfs.error module The implementations of the hdfs clients. The hadoop cli client and the snakebite client.

exception `luigi.contrib.hdfs.error.`**`HDFSCliError`**`(`*command*, *returncode*, *stdout*, *stderr*`)`

Bases: `exceptions.Exception`

luigi.contrib.hdfs.format module

class `luigi.contrib.hdfs.format.`**`HdfsReadPipe`**`(`*path*`)`

Bases: *`luigi.format.InputPipeProcessWrapper`*

class `luigi.contrib.hdfs.format.`**`HdfsAtomicWritePipe`**`(`*path*`)`

Bases: *`luigi.format.OutputPipeProcessWrapper`*

File like object for writing to HDFS

The referenced file is first written to a temporary location and then renamed to final location on close(). If close() isn’t called the temporary file will be cleaned up when this object is garbage collected

TODO: if this is buggy, change it so it first writes to a local temporary file and then uploads it on completion

`abort``()`

`close``()`

class `luigi.contrib.hdfs.format.`**`HdfsAtomicWriteDirPipe`**`(`*path*, *data_extension=’‘*`)`

Bases: *`luigi.format.OutputPipeProcessWrapper`*

Writes a data<data_extension> file to a directory at <path>.

`abort``()`

`close``()`

class `luigi.contrib.hdfs.format.`**`PlainFormat`**

Bases: *`luigi.format.Format`*

input = 'bytes'

output = 'hdfs'

hdfs_writer (*path*)

hdfs_reader (*path*)

pipe_reader (*path*)

pipe_writer (*output_pipe*)

class luigi.contrib.hdfs.format.**PlainDirFormat**

Bases: *luigi.format.Format*

input = 'bytes'

output = 'hdfs'

hdfs_writer (*path*)

hdfs_reader (*path*)

pipe_reader (*path*)

pipe_writer (*path*)

class luigi.contrib.hdfs.format.**CompatibleHdfsFormat** (*writer*, *reader*, *input=None*)

Bases: *luigi.format.Format*

output = 'hdfs'

pipe_writer (*output*)

pipe_reader (*input*)

hdfs_writer (*output*)

hdfs_reader (*input*)

luigi.contrib.hdfs.hadoopcli_clients module The implementations of the hdfs clients. The hadoop cli client and the snakebite client.

luigi.contrib.hdfs.hadoopcli_clients.**create_hadoopcli_client** ()

Given that we want one of the hadoop cli clients (unlike snakebite), this one will return the right one.

class luigi.contrib.hdfs.hadoopcli_clients.**HdfsClient**

Bases: *luigi.contrib.hdfs.abstract_client.HdfsFileSystem*

This client uses Apache 2.x syntax for file system commands, which also matched CDH4.

recursive_listdir_cmd = ['-ls', '-R']

static call_check (*command*)

exists (*path*)

Use `hadoop fs -stat` to check file existence.

rename (*path*, *dest*)

remove (*path*, *recursive=True*, *skip_trash=False*)

chmod (*path*, *permissions*, *recursive=False*)

chown (*path*, *owner*, *group*, *recursive=False*)

count (*path*)

copy (*path*, *destination*)

put (*local_path*, *destination*)

get (*path*, *local_destination*)

getmerge (*path*, *local_destination*, *new_line=False*)

mkdir (*path*, *parents=True*, *raise_if_exists=False*)

listdir (*path*, *ignore_directories=False*, *ignore_files=False*, *include_size=False*, *include_type=False*, *include_time=False*, *recursive=False*)

touchz (*path*)

class `luigi.contrib.hdfs.hadoopcli_clients.`**HdfsClientCdh3**

Bases: *luigi.contrib.hdfs.hadoopcli_clients.HdfsClient*

This client uses CDH3 syntax for file system commands.

mkdir (*path*)

No -p switch, so this will fail creating ancestors.

remove (*path*, *recursive=True*, *skip_trash=False*)

class `luigi.contrib.hdfs.hadoopcli_clients.`**HdfsClientApache1**

Bases: *luigi.contrib.hdfs.hadoopcli_clients.HdfsClientCdh3*

This client uses Apache 1.x syntax for file system commands, which are similar to CDH3 except for the file existence check.

recursive_listdir_cmd = ['-lsr']

exists (*path*)

luigi.contrib.hdfs.snakebite_client module A luigi file system client that wraps around snakebite

Originally written by Alan Brenner <alan@magnetic.com> github.com/alanbbr

class `luigi.contrib.hdfs.snakebite_client.`**SnakebiteHdfsClient**

Bases: *luigi.contrib.hdfs.abstract_client.HdfsFileSystem*

A hdfs client using snakebite. Since Snakebite has a python API, it'll be about 100 times faster than the hadoop cli client, which does shell out to a java program on each file system operation.

static list_path (*path*)

get_bite ()

If Luigi has forked, we have a different PID, and need to reconnect.

exists (*path*)

Use snakebite.test to check file existence.

Parameters **path** (*string*) – path to test

Returns boolean, True if path exists in HDFS

rename (*path*, *dest*)

Use snakebite.rename, if available.

Parameters

- **path** (*either a string or sequence of strings*) – source file(s)
- **dest** (*string*) – destination file (single input) or directory (multiple)

Returns list of renamed items

rename_dont_move (*path*, *dest*)

Use snakebite.rename_dont_move, if available.

Parameters

- **path** (*string*) – source path (single input)
- **dest** (*string*) – destination path

Returns True if succeeded

Raises snakebite.errors.FileAlreadyExistsException

remove (*path*, *recursive=True*, *skip_trash=False*)

Use snakebite.delete, if available.

Parameters

- **path** (*either a string or a sequence of strings*) – delete-able file(s) or directory(ies)
- **recursive** (*boolean, default is True*) – delete directories trees like *nix: rm -r
- **skip_trash** (*boolean, default is False (use trash)*) – do or don't move deleted items into the trash first

Returns list of deleted items

chmod (*path*, *permissions*, *recursive=False*)

Use snakebite.chmod, if available.

Parameters

- **path** (*either a string or sequence of strings*) – update-able file(s)
- **permissions** (*octal*) – *nix style permission number
- **recursive** (*boolean, default is False*) – change just listed entry(ies) or all in directories

Returns list of all changed items

chown (*path*, *owner*, *group*, *recursive=False*)

Use snakebite.chown/chgrp, if available.

One of owner or group must be set. Just setting group calls chgrp.

Parameters

- **path** (*either a string or sequence of strings*) – update-able file(s)
- **owner** (*string*) – new owner, can be blank
- **group** (*string*) – new group, can be blank
- **recursive** (*boolean, default is False*) – change just listed entry(ies) or all in directories

Returns list of all changed items

count (*path*)

Use snakebite.count, if available.

Parameters **path** (*string*) – directory to count the contents of

Returns dictionary with content_size, dir_count and file_count keys

copy (*path*, *destination*)

Raise a NotImplementedError exception.

put (*local_path*, *destination*)

Raise a NotImplementedError exception.

get (*path*, *local_destination*)
Use snakebite.copyToLocal, if available.

Parameters

- **path** (*string*) – HDFS file
- **local_destination** (*string*) – path on the system running Luigi

mkdir (*path*, *parents=True*, *mode=493*, *raise_if_exists=False*)
Use snakebite.mkdir, if available.

Snakebite's mkdir method allows control over full path creation, so by default, tell it to build a full path to work like `hadoop fs -mkdir`.

Parameters

- **path** (*string*) – HDFS path to create
- **parents** (*boolean, default is True*) – create any missing parent directories
- **mode** (*octal, default 0755*) – *nix style owner/group/other permissions

listdir (*path*, *ignore_directories=False*, *ignore_files=False*, *include_size=False*, *include_type=False*, *include_time=False*, *recursive=False*)
Use snakebite.ls to get the list of items in a directory.

Parameters

- **path** (*string*) – the directory to list
- **ignore_directories** (*boolean, default is False*) – if True, do not yield directory entries
- **ignore_files** (*boolean, default is False*) – if True, do not yield file entries
- **include_size** (*boolean, default is False (do not include)*) – include the size in bytes of the current item
- **include_type** (*boolean, default is False (do not include)*) – include the type (d or f) of the current item
- **include_time** (*boolean, default is False (do not include)*) – include the last modification time of the current item
- **recursive** (*boolean, default is False (do not recurse)*) – list subdirectory contents

Returns yield with a string, or if any of the include_* settings are true, a tuple starting with the path, and include_* items in order

touchz (*path*)
Raise a NotImplementedError exception.

luigi.contrib.hdfs.target module Provides access to HDFS using the *HdfsTarget*, a subclass of *Target*.

class luigi.contrib.hdfs.target.**HdfsTarget** (*path=None*, *format=None*, *is_tmp=False*, *fs=None*)
Bases: *luigi.target.FileSystemTarget*

fs

glob_exists (*expected_files*)

open (*mode='r'*)

remove (*skip_trash=False*)

rename (*path*, *raise_if_exists=False*)
Rename does not change self.path, so be careful with assumptions.

Not recommendeed for directories. Use move_dir. spotify/luigi#522

move (*path*, *raise_if_exists=False*)
Move does not change self.path, so be careful with assumptions.

Not recommendeed for directories. Use move_dir. spotify/luigi#522

move_dir (*path*)
Rename a directory.

The implementation uses *rename_dont_move*, which on some clients is just a normal *mv* operation, which can cause nested directories.

One could argue that the implementation should use the mkdir+raise_if_exists approach, but we at Spotify have had more trouble with that over just using plain mv. See spotify/luigi#557

is_writable ()
Currently only works with hadoopcli

Module contents Provides access to HDFS using the `HdfsTarget`, a subclass of *`Target`*. You can configure what client by setting the "client" config under the "hdfs" section in the configuration, or using the `--hdfs-client` command line option. "hadoopcli" is the slowest, but should work out of the box. "snakebite" is the fastest, but requires Snakebite to be installed.

Currently (4th May) the *`luigi.contrib.hdfs`* module is under reorganization. We recommend importing the reexports from *`luigi.contrib.hdfs`* instead of the sub-modules, as we're not yet sure how the final structure of the sub-modules will be. Eventually this module will be empty and you'll have to import directly from the sub modules like *`luigi.contrib.hdfs.config`*.

Submodules

luigi.contrib.bigquery module

class `luigi.contrib.bigquery.`**CreateDisposition**
Bases: `object`

CREATE_IF_NEEDED = 'CREATE_IF_NEEDED'

CREATE_NEVER = 'CREATE_NEVER'

class `luigi.contrib.bigquery.`**WriteDisposition**
Bases: `object`

WRITE_TRUNCATE = 'WRITE_TRUNCATE'

WRITE_APPEND = 'WRITE_APPEND'

WRITE_EMPTY = 'WRITE_EMPTY'

class `luigi.contrib.bigquery.`**QueryMode**
Bases: `object`

INTERACTIVE = 'INTERACTIVE'

BATCH = 'BATCH'

class `luigi.contrib.bigquery.`**SourceFormat**
Bases: `object`

CSV = 'CSV'

DATASTORE_BACKUP = 'DATASTORE_BACKUP'

NEWLINE_DELIMITED_JSON = 'NEWLINE_DELIMITED_JSON'

class `luigi.contrib.bigquery.`**`BQDataset`**(*project_id*, *dataset_id*)

Bases: `tuple`

`dataset_id`

Alias for field number 1

`project_id`

Alias for field number 0

class `luigi.contrib.bigquery.`**`BQTable`**

Bases: *`luigi.contrib.bigquery.BQTable`*

`dataset`

class `luigi.contrib.bigquery.`**`BigqueryClient`**(*oauth_credentials=None*, *descriptor=''*, *http_=None*)

Bases: `object`

A client for Google BigQuery.

For details of how authentication and the descriptor work, see the documentation for the GCS client. The descriptor URL for BigQuery is https://www.googleapis.com/discovery/v1/apis/bigquery/v2/rest

`dataset_exists`(*dataset*)

Returns whether the given dataset exists.

Parameters **`dataset`** (BQDataset) –

`table_exists`(*table*)

Returns whether the given table exists.

Parameters **`table`** (BQTable) –

`make_dataset`(*dataset*, *raise_if_exists=False*, *body={}*)

Creates a new dataset with the default permissions.

Parameters

- **`dataset`** (BQDataset) –
- **`raise_if_exists`** – whether to raise an exception if the dataset already exists.

Raises ***luigi.target.FileAlreadyExists*** if raise_if_exists=True and the dataset exists

`delete_dataset`(*dataset*, *delete_nonempty=True*)

Deletes a dataset (and optionally any tables in it), if it exists.

Parameters

- **`dataset`** (BQDataset) –
- **`delete_nonempty`** – if true, will delete any tables before deleting the dataset

`delete_table`(*table*)

Deletes a table, if it exists.

Parameters **`table`** (BQTable) –

`list_datasets`(*project_id*)

Returns the list of datasets in a given project.

Parameters **`project_id`** (*str*) –

list_tables (*dataset*)

Returns the list of tables in a given dataset.

Parameters **dataset** (BQDataset) –

run_job (*project_id*, *body*, *dataset=None*)

Runs a bigquery "job". See the documentation for the format of body.

Note: You probably don't need to use this directly. Use the tasks defined below.

Parameters **dataset** (BQDataset) –

copy (*source_table*, *dest_table*, *create_disposition='CREATE_IF_NEEDED'*, *write_disposition='WRITE_TRUNCATE'*)

Copies (or appends) a table to another table.

Parameters

- **source_table** (BQTable) –
- **dest_table** (BQTable) –
- **create_disposition** (CreateDisposition) – whether to create the table if needed
- **write_disposition** (WriteDisposition) – whether to append/truncate/fail if the table exists

class luigi.contrib.bigquery.**BigqueryTarget** (*project_id*, *dataset_id*, *table_id*, *client=None*)

Bases: *luigi.target.Target*

classmethod from_bqtable (*table*, *client=None*)

A constructor that takes a *BQTable*.

Parameters **table** (BQTable) –

exists ()

class luigi.contrib.bigquery.**BigqueryLoadTask** (**args*, ***kwargs*)

Bases: *luigi.task.Task*

Load data into bigquery from GCS.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

source_format

The source format to use (see *SourceFormat*).

write_disposition

What to do if the table already exists. By default this will fail the job.

See *WriteDisposition*

schema

Schema in the format defined at https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load.schema.

If the value is falsy, it is omitted and inferred by bigquery, which only works for CSV inputs.

max_bad_records

source_uris
Source data which should be in GCS.

run()

task_namespace = None

class luigi.contrib.bigquery.**BigqueryRunQueryTask**(**args*, ***kwargs*)
Bases: *luigi.task.Task*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

write_disposition
What to do if the table already exists. By default this will fail the job.

See *WriteDisposition*

create_disposition
Whether to create the table or not. See *CreateDisposition*

query
The query, in text form.

query_mode
The query mode. See *QueryMode*.

run()

task_namespace = None

luigi.contrib.esindex module Support for Elasticsearch (1.0.0 or newer).

Provides an *ElasticsearchTarget* and a *CopyToIndex* template task.

Modeled after *luigi.contrib.rdbms.CopyToTable*.

A minimal example (assuming elasticsearch is running on localhost:9200):

```
class ExampleIndex(CopyToIndex):
    index = 'example'

    def docs(self):
        return [{'_id': 1, 'title': 'An example document.'}]

if __name__ == '__main__':
    task = ExampleIndex()
    luigi.build([task], local_scheduler=True)
```

All options:

```
class ExampleIndex(CopyToIndex):
    host = 'localhost'
    port = 9200
    index = 'example'
    doc_type = 'default'
    purge_existing_index = True
    marker_index_hist_size = 1

    def docs(self):
        return [{'_id': 1, 'title': 'An example document.'}]

if __name__ == '__main__':
    task = ExampleIndex()
    luigi.build([task], local_scheduler=True)
```

Host, *port*, *index*, *doc_type* parameters are standard elasticsearch.

purge_existing_index will delete the index, whenever an update is required. This is useful, when one deals with "dumps" that represent the whole data, not just updates.

marker_index_hist_size sets the maximum number of entries in the 'marker' index:

- 0 (default) keeps all updates,
- 1 to only remember the most recent update to the index.

This can be useful, if an index needs to recreated, even though the corresponding indexing task has been run sometime in the past - but a later indexing task might have altered the index in the meantime.

There are a two luigi *client.cfg* configuration options:

```
[elasticsearch]

marker-index = update_log
marker-doc-type = entry
```

class `luigi.contrib.esindex.`**`ElasticsearchTarget`**(*host*, *port*, *index*, *doc_type*, *update_id*, *marker_index_hist_size=0*, *http_auth=None*, *timeout=10*, *extra_elasticsearch_args={}*)

Bases: *`luigi.target.Target`*

Target for a resource in Elasticsearch.

Parameters

- **`host`** (*str*) – Elasticsearch server host
- **`port`** (*int*) – Elasticsearch server port
- **`index`** (*str*) – index name
- **`doc_type`** (*str*) – doctype name
- **`update_id`** (*str*) – an identifier for this data set
- **`marker_index_hist_size`** (*int*) – list of changes to the index to remember
- **`timeout`** (*int*) – Elasticsearch connection timeout
- **`extra_elasticsearch_args`** – extra args for Elasticsearch

`marker_index` **= 'update_log'**

`marker_doc_type` **= 'entry'**

`marker_index_document_id`()

Generate an id for the indicator document.

`touch`()

Mark this update as complete.

The document id would be sufficent but, for documentation, we index the parameters *update_id*, *target_index*, *target_doc_type* and *date* as well.

exists()

Test, if this task has been run.

create_marker_index()

Create the index that will keep track of the tasks if necessary.

ensure_hist_size()

Shrink the history of updates for a *index/doc_type* combination down to *self.marker_index_hist_size*.

class luigi.contrib.esindex.**CopyToIndex**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Template task for inserting a data set into Elasticsearch.

Usage:

1. Subclass and override the required *index* attribute.
2. Implement a custom *docs* method, that returns an iterable over the documents. A document can be a JSON string, e.g. from a newline-delimited JSON (ldj) file (default implementation) or some dictionary.

Optional attributes:

- doc_type (default),
- host (localhost),
- port (9200),
- settings ({'settings': {}})
- mapping (None),
- chunk_size (2000),
- raise_on_error (True),
- purge_existing_index (False),
- marker_index_hist_size (0)

If settings are defined, they are only applied at index creation time.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

host

ES hostname.

port

ES port.

http_auth

ES optional http auth information as either ':' separated string or a tuple, e.g. *('user', 'pass')* or *"user:pass"*.

index
The target index.

May exist or not.

doc_type
The target doc_type.

mapping
Dictionary with custom mapping or *None*.

settings
Settings to be used at index creation time.

chunk_size
Single API call for this number of docs.

raise_on_error
Whether to fail fast.

purge_existing_index
Whether to delete the *index* completely before any indexing.

marker_index_hist_size
Number of event log entries in the marker index. 0: unlimited.

timeout
Timeout.

extra_elasticsearch_args
Extra arguments to pass to the Elasticsearch constructor

docs()
Return the documents to be indexed.

Beside the user defined fields, the document may contain an *_index*, *_type* and *_id*.

create_index()
Override to provide code for creating the target index.

By default it will be created without any special settings or mappings.

delete_index()
Delete the index, if it exists.

update_id()
This id will be a unique identifier for this indexing task.

output()
Returns a ElasticsearchTarget representing the inserted dataset.

Normally you don't override this.

run()
Run task, namely:

- purge existing index, if requested (*purge_existing_index*),
- create the index, if missing,
- apply mappings, if given,
- set refresh interval to -1 (disable) for performance reasons,
- bulk index in batches of size *chunk_size* (2000),

- set refresh interval to 1s,
- refresh Elasticsearch,
- create entry in marker index.

task_namespace = None

luigi.contrib.ftp module This library is a wrapper of ftplib. It is convenient to move data from/to FTP.

There is an example on how to use it (example/ftp_experiment_outputs.py)

You can also find unittest for each class.

Be aware that normal ftp do not provide secure communication.

class `luigi.contrib.ftp.`**`RemoteFileSystem`**(*host*, *username=None*, *password=None*, *port=21*, *tls=False*)

Bases: *`luigi.target.FileSystem`*

`exists`(*path*, *mtime=None*)

Return *True* if file or directory at *path* exist, False otherwise.

Additional check on modified time when mtime is passed in.

Return False if the file's modified time is older mtime.

`remove`(*path*, *recursive=True*)

Remove file or directory at location `path`.

Parameters

- **`path`** (*str*) – a path within the FileSystem to remove.
- **`recursive`** (*bool*) – if the path is a directory, recursively remove the directory and all of its descendants. Defaults to `True`.

`put`(*local_path*, *path*)

`get`(*path*, *local_path*)

class `luigi.contrib.ftp.`**`AtomicFtpFile`**(*fs*, *path*)

Bases: *`luigi.target.AtomicLocalFile`*

Simple class that writes to a temp file and upload to ftp on close().

Also cleans up the temp file if close is not invoked.

Initializes an AtomicFtpfile instance. :param fs: :param path: :type path: str

`move_to_final_destination`()

`fs`

class `luigi.contrib.ftp.`**`RemoteTarget`**(*path*, *host*, *format=None*, *username=None*, *password=None*, *port=21*, *mtime=None*, *tls=False*)

Bases: *`luigi.target.FileSystemTarget`*

Target used for reading from remote files.

The target is implemented using ssh commands streaming data over the network.

`fs`

`open`(*mode*)

Open the FileSystem target.

This method returns a file-like object which can either be read from or written to depending on the specified mode.

Parameters **`mode`** (*str*) – the mode *r* opens the FileSystemTarget in read-only mode, whereas *w* will open the FileSystemTarget in write mode. Subclasses can implement additional options.

`exists`()

`put`(*local_path*)

`get`(*local_path*)

luigi.contrib.gcs module luigi bindings for Google Cloud Storage

exception `luigi.contrib.gcs.`**`InvalidDeleteException`**

Bases: *`luigi.target.FileSystemException`*

class `luigi.contrib.gcs.`**`GCSClient`**(*oauth_credentials=None*, *descriptor=’‘*, *http_=None*)

Bases: *`luigi.target.FileSystem`*

An implementation of a FileSystem over Google Cloud Storage.

There are several ways to use this class. By default it will use the app default credentials, as described at https://developers.google.com/identity/protocols/application-default-credentials . Alternatively, you may pass an oauth2client credentials object. e.g. to use a service account:

```
credentials = oauth2client.client.SignedJwtAssertionCredentials(
    '012345678912-ThisIsARandomServiceAccountEmail@developer.gserviceaccount.com',
    'These are the contents of the p12 file that came with the service account',
    scope='https://www.googleapis.com/auth/devstorage.read_write')
client = GCSClient(oauth_credentials=credentails)
```

Warning: By default this class will use “automated service discovery” which will require a connection to the web. The google api client downloads a JSON file to “create” the library interface on the fly. If you want a more hermetic build, you can pass the contents of this file (currently found at https://www.googleapis.com/discovery/v1/apis/storage/v1/rest) as the `descriptor` argument.

`exists`(*path*)

`isdir`(*path*)

`remove`(*path*, *recursive=True*)

`put`(*filename*, *dest_path*, *mimetype=None*)

`put_string`(*contents*, *dest_path*, *mimetype=None*)

`mkdir`(*path*, *parents=True*, *raise_if_exists=False*)

`copy`(*source_path*, *destination_path*)

`rename`(*source_path*, *destination_path*)

Rename/move an object from one S3 location to another.

`listdir`(*path*)

Get an iterable with S3 folder contents. Iterable contains paths relative to queried path.

`download`(*path*)

class `luigi.contrib.gcs.`**`AtomicGCSFile`**(*path*, *gcs_client*)

Bases: *`luigi.target.AtomicLocalFile`*

A GCS file that writes to a temp file and put to GCS on close.

move_to_final_destination()

class luigi.contrib.gcs.**GCSTarget**(*path*, *format=None*, *client=None*)

Bases: *luigi.target.FileSystemTarget*

fs = None

open(*mode='r'*)

class luigi.contrib.gcs.**GCSFlagTarget**(*path*, *format=None*, *client=None*, *flag='_SUCCESS'*)

Bases: *luigi.contrib.gcs.GCSTarget*

Defines a target directory with a flag-file (defaults to *_SUCCESS*) used to signify job success.

This checks for two things:

- the path exists (just like the GCSTarget)
- the _SUCCESS file exists within the directory.

Because Hadoop outputs into a directory and not a single file, the path is assumed to be a directory.

This is meant to be a handy alternative to AtomicGCSFile.

The AtomicFile approach can be burdensome for GCS since there are no directories, per se.

If we have 1,000,000 output files, then we have to rename 1,000,000 objects.

Initializes a S3FlagTarget.

Parameters

- **path** (*str*) – the directory where the files are stored.
- **client** –
- **flag** (*str*) –

fs = None

exists()

luigi.contrib.hadoop module Run Hadoop Mapreduce jobs using Hadoop Streaming. To run a job, you need to subclass *luigi.contrib.hadoop.JobTask* and implement a `mapper` and `reducer` methods. See Example – Top Artists for an example of how to run a Hadoop job.

class luigi.contrib.hadoop.**BaseHadoopJobTask**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

batch_counter_default = 1

deps()

final_combiner = NotImplemented

final_mapper = NotImplemented

final_reducer = NotImplemented

init_hadoop()

init_local()

Implement any work to setup any internal datastructure etc here.

You can add extra input using the requires_local/input_local methods.

Anything you set on the object will be pickled and available on the Hadoop nodes.

input_hadoop()

input_local()

job_runner()

jobconfs()

mr_priority = NotImplemented

on_failure(*exception*)

pool = <luigi.parameter.Parameter object>

requires_hadoop()

requires_local()

Default impl - override this method if you need any local input to be accessible in init().

run()

task_id = None

task_namespace = None

class luigi.contrib.hadoop.**DefaultHadoopJobRunner**

Bases: *luigi.contrib.hadoop.HadoopJobRunner*

The default job runner just reads from config and sets stuff.

exception luigi.contrib.hadoop.**HadoopJobError**(*message*, *out=None*, *err=None*)

Bases: exceptions.RuntimeError

class luigi.contrib.hadoop.**HadoopJobRunner**(*streaming_jar*, *modules=None*, *streaming_args=None*, *libjars=None*, *libjars_in_hdfs=None*, *jobconfs=None*, *input_format=None*, *output_format=None*, *end_job_with_atomic_move_dir=True*)

Bases: *luigi.contrib.hadoop.JobRunner*

Takes care of uploading & executing a Hadoop job using Hadoop streaming.

TODO: add code to support Elastic Mapreduce (using boto) and local execution.

finish()

run_job(*job*)

class luigi.contrib.hadoop.**HadoopRunContext**

Bases: object

kill_job(*captured_signal=None*, *stack_frame=None*)

class luigi.contrib.hadoop.**JobRunner**

Bases: object

run_job = NotImplemented

class luigi.contrib.hadoop.**JobTask**(**args*, ***kwargs*)

Bases: *luigi.contrib.hadoop.BaseHadoopJobTask*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

add_link(*src*, *dst*)

combiner = NotImplemented

data_interchange_format = ‘python’

deserialize

dump(*directory=’‘*)

Dump instance to file.

extra_files()

Can be overriden in subclass.

Each element is either a string, or a pair of two strings (src, dst).

- *src* can be a directory (in which case everything will be copied recursively).
- *dst* can include subdirectories (foo/bar/baz.txt etc)

Uses Hadoop’s -files option so that the same file is reused across tasks.

extra_modules()

incr_counter(**args*, ***kwargs*)

Increments a Hadoop counter.

Since counters can be a bit slow to update, this batches the updates.

init_combiner()

init_mapper()

init_reducer()

internal_reader(*input_stream*)

Reader which uses python eval on each part of a tab separated string. Yields a tuple of python objects.

internal_serialize

internal_writer(*outputs*, *stdout*)

Writer which outputs the python repr for each item.

job_runner()

Get the MapReduce runner for this job.

If all outputs are HdfsTargets, the DefaultHadoopJobRunner will be used. Otherwise, the LocalJobRunner which streams all data through the local machine will be used (great for testing).

jobconfs()

mapper(*item*)

Re-define to process an input item (usually a line of input data).

Defaults to identity mapper that sends all lines to the same reducer.

n_reduce_tasks = 25

reader (*input_stream*)

Reader is a method which iterates over input lines and outputs records.

The default implementation yields one argument containing the line for each line in the input.

reducer = NotImplemented

run_combiner (*stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>*)

run_mapper (*stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>*)

Run the mapper on the hadoop node.

run_reducer (*stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>*)

Run the reducer on the hadoop node.

serialize

task_namespace = None

writer (*outputs, stdout, stderr=<open file '<stderr>', mode 'w'>*)

Writer format is a method which iterates over the output records from the reducer and formats them for output.

The default implementation outputs tab separated items.

class luigi.contrib.hadoop.**LocalJobRunner** (*samplelines=None*)

Bases: *luigi.contrib.hadoop.JobRunner*

Will run the job locally.

This is useful for debugging and also unit testing. Tries to mimic Hadoop Streaming.

TODO: integrate with JobTask

group (*input_stream*)

run_job (*job*)

sample (*input_stream, n, output*)

luigi.contrib.hadoop.**attach** (**packages*)

Attach a python package to hadoop map reduce tarballs to make those packages available on the hadoop cluster.

luigi.contrib.hadoop.**create_packages_archive** (*packages, filename*)

Create a tar archive which will contain the files for the packages listed in packages.

luigi.contrib.hadoop.**dereference** (*f*)

luigi.contrib.hadoop.**fetch_task_failures** (*tracking_url*)

Uses mechanize to fetch the actual task logs from the task tracker.

This is highly opportunistic, and we might not succeed. So we set a low timeout and hope it works. If it does not, it's not the end of the world.

TODO: Yarn has a REST API that we should probably use instead: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html

luigi.contrib.hadoop.**flatten** (*sequence*)

A simple generator which flattens a sequence.

Only one level is flattened.

```
(1, (2, 3), 4) -> (1, 2, 3, 4)
```

luigi.contrib.hadoop.**get_extra_files** (*extra_files*)

class `luigi.contrib.hadoop.`**`hadoop`**`(`**args*, ***kwargs*`)`

Bases: *`luigi.task.Config`*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`pool` **= <luigi.parameter.Parameter object>**

`task_namespace` **= None**

`luigi.contrib.hadoop.`**`run_and_track_hadoop_job`**`(`*arglist*, *tracking_url_callback=None*, *env=None*`)`

Runs the job by invoking the command from the given arglist. Finds tracking urls from the output and attempts to fetch errors using those urls if the job fails. Throws HadoopJobError with information about the error (including stdout and stderr from the process) on failure and returns normally otherwise.

Parameters

- **`arglist`** –
- **`tracking_url_callback`** –
- **`env`** –

Returns

luigi.contrib.hadoop_jar module Provides functionality to run a Hadoop job using a Jar

`luigi.contrib.hadoop_jar.`**`fix_paths`**`(`*job*`)`

Coerce input arguments to use temporary files when used for output.

Return a list of temporary file pairs (tmpfile, destination path) and a list of arguments.

Converts each HdfsTarget to a string for the path.

exception `luigi.contrib.hadoop_jar.`**`HadoopJarJobError`**

Bases: `exceptions.Exception`

class `luigi.contrib.hadoop_jar.`**`HadoopJarJobRunner`**

Bases: *`luigi.contrib.hadoop.JobRunner`*

JobRunner for *hadoop jar* commands. Used to run a HadoopJarJobTask.

`run_job``(`*job*`)`

class `luigi.contrib.hadoop_jar.`**`HadoopJarJobTask`**`(`**args*, ***kwargs*`)`

Bases: *`luigi.contrib.hadoop.BaseHadoopJobTask`*

A job task for *hadoop jar* commands that define a jar and (optional) main method.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

jar()
Path to the jar for this Hadoop Job.

main()
optional main method for this Hadoop Job.

job_runner()

atomic_output()
If True, then rewrite output arguments to be temp locations and atomically move them into place after the job finishes.

ssh()
Set this to run hadoop command remotely via ssh. It needs to be a dict that looks like {"host": "myhost", "key_file": None, "username": None, ["no_host_key_check": False]}

args()
Returns an array of args to pass to the job (after hadoop jar <jar> <main>).

task_namespace = None

luigi.contrib.hive module

exception luigi.contrib.hive.**HiveCommandError**(*message*, *out=None*, *err=None*)
Bases: exceptions.RuntimeError

luigi.contrib.hive.**load_hive_cmd**()

luigi.contrib.hive.**get_hive_syntax**()

luigi.contrib.hive.**run_hive**(*args*, *check_return_code=True*)
Runs the *hive* from the command line, passing in the given args, and returning stdout.

With the apache release of Hive, so of the table existence checks (which are done using DESCRIBE do not exit with a return code of 0 so we need an option to ignore the return code and just return stdout for parsing

luigi.contrib.hive.**run_hive_cmd**(*hivecmd*, *check_return_code=True*)
Runs the given hive query and returns stdout.

luigi.contrib.hive.**run_hive_script**(*script*)
Runs the contents of the given script in hive and returns stdout.

class luigi.contrib.hive.**HiveClient**
Bases: object

table_location(*table*, *database='default'*, *partition=None*)
Returns location of db.table (or db.table.partition). partition is a dict of partition key to value.

table_schema(*table*, *database='default'*)
Returns list of [(name, type)] for each column in database.table.

table_exists(*table*, *database='default'*, *partition=None*)
Returns true if db.table (or db.table.partition) exists. partition is a dict of partition key to value.

partition_spec(*partition*)
Turn a dict into a string partition specification

class luigi.contrib.hive.**HiveCommandClient**
Bases: *luigi.contrib.hive.HiveClient*

Uses *hive* invocations to find information.

table_location(*table*, *database='default'*, *partition=None*)

table_exists(*table*, *database='default'*, *partition=None*)

table_schema (*table*, *database='default'*)

partition_spec (*partition*)

Turns a dict into the a Hive partition specification string.

class luigi.contrib.hive.**ApacheHiveCommandClient**

Bases: *luigi.contrib.hive.HiveCommandClient*

A subclass for the HiveCommandClient to (in some cases) ignore the return code from the hive command so that we can just parse the output.

table_schema (*table*, *database='default'*)

class luigi.contrib.hive.**MetastoreClient**

Bases: *luigi.contrib.hive.HiveClient*

table_location (*table*, *database='default'*, *partition=None*)

table_exists (*table*, *database='default'*, *partition=None*)

table_schema (*table*, *database='default'*)

partition_spec (*partition*)

class luigi.contrib.hive.**HiveThriftContext**

Bases: object

Context manager for hive metastore client.

luigi.contrib.hive.**get_default_client** ()

class luigi.contrib.hive.**HiveQueryTask** (**args*, ***kwargs*)

Bases: *luigi.contrib.hadoop.BaseHadoopJobTask*

Task to run a hive query.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

n_reduce_tasks = None

bytes_per_reducer = None

reducers_max = None

query ()

Text of query to run in hive

hiverc ()

Location of an rc file to run before the query if hiverc-location key is specified in client.cfg, will default to the value there otherwise returns None.

Returning a list of rc files will load all of them in order.

hiveconfs ()

Returns an dict of key=value settings to be passed along to the hive command line via –hiveconf. By default, sets mapred.job.name to task_id and if not None, sets:

- mapred.reduce.tasks (n_reduce_tasks)
- mapred.fairscheduler.pool (pool) or mapred.job.queue.name (pool)

- hive.exec.reducers.bytes.per.reducer (bytes_per_reducer)
- hive.exec.reducers.max (reducers_max)

job_runner()

task_namespace = None

class luigi.contrib.hive.**HiveQueryRunner**

Bases: *luigi.contrib.hadoop.JobRunner*

Runs a HiveQueryTask by shelling out to hive.

prepare_outputs(*job*)

Called before job is started.

If output is a *FileSystemTarget*, create parent directories so the hive command won't fail

run_job(*job*)

class luigi.contrib.hive.**HiveTableTarget**(*table*, *database='default'*, *client=None*)

Bases: *luigi.target.Target*

exists returns true if the table exists.

exists()

path

Returns the path to this table in HDFS.

open(*mode*)

class luigi.contrib.hive.**HivePartitionTarget**(*table*, *partition*, *database='default'*, *fail_missing_table=True*, *client=None*)

Bases: *luigi.target.Target*

exists returns true if the table's partition exists.

exists()

path

Returns the path for this HiveTablePartitionTarget's data.

open(*mode*)

class luigi.contrib.hive.**ExternalHiveTask**(**args*, ***kwargs*)

Bases: *luigi.task.ExternalTask*

External task that depends on a Hive table/partition.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

database = <luigi.parameter.Parameter object>

table = <luigi.parameter.Parameter object>

partition = <luigi.parameter.Parameter object>

task_namespace = None

output()

luigi.contrib.mysqldb module

class `luigi.contrib.mysqldb.`**`MySqlTarget`**(*host*, *database*, *user*, *password*, *table*, *update_id*)

Bases: *`luigi.target.Target`*

Target for a resource in MySql.

Initializes a MySqlTarget instance.

Parameters

- **`host`** (*str*) – MySql server address. Possibly a host:port string.
- **`database`** (*str*) – database name.
- **`user`** (*str*) – database user
- **`password`** (*str*) – password for specified user.
- **`update_id`** (*str*) – an identifier for this data set.

`marker_table` **= 'table_updates'**

`touch`(*connection=None*)

Mark this update as complete.

IMPORTANT, If the marker table doesn't exist, the connection transaction will be aborted and the connection reset. Then the marker table will be created.

`exists`(*connection=None*)

`connect`(*autocommit=False*)

`create_marker_table`()

Create marker table if it doesn't exist.

Using a separate connection since the transaction might have to be reset.

luigi.contrib.pig module Apache Pig support. Example configuration section in client.cfg:

```
[pig]
# pig home directory
home: /usr/share/pig
```

class `luigi.contrib.pig.`**`PigJobTask`**(**args*, ***kwargs*)

Bases: *`luigi.task.Task`*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`pig_home`()

`pig_command_path`()

`pig_env_vars`()

Dictionary of environment variables that should be set when running Pig.

Ex:: return { 'PIG_CLASSPATH': '/your/path' }

pig_properties()

Dictionary of properties that should be set when running Pig.

Example:

```
    return { 'pig.additional.jars':'/path/to/your/jar' }
```

pig_parameters()

Dictionary of parameters that should be set for the Pig job.

Example:

```
    return { 'YOUR_PARAM_NAME':'Your param value' }
```

pig_options()

List of options that will be appended to the Pig command.

Example:

```
    return ['-x', 'local']
```

output()

pig_script_path()

Return the path to the Pig script to be run.

run()

track_and_progress(*cmd*)

task_namespace = None

class luigi.contrib.pig.**PigRunContext**

Bases: object

kill_job(*captured_signal=None*, *stack_frame=None*)

exception luigi.contrib.pig.**PigJobError**(*message*, *out=None*, *err=None*)

Bases: exceptions.RuntimeError

luigi.contrib.pyspark_runner module The pyspark program.

This module will be run by spark-submit for PySparkTask jobs.

The first argument is a path to the pickled instance of the PySparkTask, other arguments are the ones returned by PySparkTask.app_options()

class luigi.contrib.pyspark_runner.**PySparkRunner**(*job*, **args*)

Bases: object

run()

luigi.contrib.rdbms module A common module for posgres like databases, such as postgres or redshift

class luigi.contrib.rdbms.**CopyToTable**(**args*, ***kwargs*)

Bases: *luigi.task.MixinNaiveBulkComplete*, *luigi.task.Task*

An abstract task for inserting a data set into RDBMS.

Usage:

Subclass and override the following attributes:

- *host*,

- *database*,
- *user*,
- *password*,
- *table*
- *columns*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`host`

`database`

`user`

`password`

`table`

`columns` = []

`null_values` = (None,)

`column_separator` = '\t'

`create_table`(*connection*)

Override to provide code for creating the target table.

By default it will be created using types (optionally) specified in columns.

If overridden, use the provided connection object for setting up the table in order to create the table and insert data using the same transaction.

`update_id`()

This update id will be a unique identifier for this insert on this table.

`output`()

`init_copy`(*connection*)

Override to perform custom queries.

Any code here will be formed in the same transaction as the main copy, just prior to copying data. Example use cases include truncating the table or removing all data older than X in the database to keep a rolling window of data available in the table.

`copy`(*cursor*, *file*)

`task_namespace` = None

luigi.contrib.redis_store module

class `luigi.contrib.redis_store.`**`RedisTarget`**(*host*, *port*, *db*, *update_id*, *password=None*, *socket_timeout=None*, *expire=None*)

Bases: *`luigi.target.Target`*

Target for a resource in Redis.

Parameters

- **host** (*str*) – Redis server host
- **port** (*int*) – Redis server port
- **db** (*int*) – database index
- **update_id** (*str*) – an identifier for this data hash
- **password** (*str*) – a password to connect to the redis server
- **socket_timeout** (*int*) – client socket timeout
- **expire** (*int*) – timeout before the target is deleted

marker_prefix = <luigi.parameter.Parameter object>

marker_key()

Generate a key for the indicator hash.

touch()

Mark this update as complete.

We index the parameters *update_id* and *date*.

exists()

Test, if this task has been run.

luigi.contrib.redshift module

class luigi.contrib.redshift.**RedshiftTarget**(*host*, *database*, *user*, *password*, *table*, *update_id*)

Bases: *luigi.postgres.PostgresTarget*

Target for a resource in Redshift.

Redshift is similar to postgres with a few adjustments required by redshift.

Args: host (str): Postgres server address. Possibly a host:port string. database (str): Database name user (str): Database user password (str): Password for specified user update_id (str): An identifier for this data set

marker_table = 'table_updates'

use_db_timestamps = False

class luigi.contrib.redshift.**S3CopyToTable**(**args*, ***kwargs*)

Bases: *luigi.contrib.rdbms.CopyToTable*

Template task for inserting a data set into Redshift from s3.

Usage:

- Subclass and override the required attributes: * *host*, * *database*, * *user*, * *password*, * *table*, * *columns*, * *aws_access_key_id*, * *aws_secret_access_key*, * *s3_load_path*.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

s3_load_path

Override to return the load path.

aws_access_key_id

Override to return the key id.

aws_secret_access_key
Override to return the secret access key.

copy_options
Add extra copy options, for example:

- TIMEFORMAT 'auto'
- IGNOREHEADER 1
- TRUNCATECOLUMNS
- IGNOREBLANKLINES

table_attributes()
Add extra table attributes, for example: DISTSTYLE KEY DISTKEY (MY_FIELD) SORTKEY (MY_FIELD_2, MY_FIELD_3)

do_truncate_table()
Return True if table should be truncated before copying new data in.

truncate_table(*connection*)

create_table(*connection*)
Override to provide code for creating the target table.

By default it will be created using types (optionally) specified in columns.

If overridden, use the provided connection object for setting up the table in order to create the table and insert data using the same transaction.

run()
If the target table doesn't exist, self.create_table will be called to attempt to create the table.

copy(*cursor*, *f*)
Defines copying from s3 into redshift.

output()
Returns a RedshiftTarget representing the inserted dataset.

Normally you don't override this.

does_table_exist(*connection*)
Determine whether the table already exists.

task_namespace = None

class luigi.contrib.redshift.**S3CopyJSONToTable**(**args*, ***kwargs*)
Bases: *luigi.contrib.redshift.S3CopyToTable*

Template task for inserting a JSON data set into Redshift from s3.

Usage:

- Subclass and override the required attributes:
  - *host*,
  - *database*,
  - *user*,
  - *password*,
  - *table*,
  - *columns*,

–*aws_access_key_id*,

–*aws_secret_access_key*,

–*s3_load_path*,

–*jsonpath*,

–*copy_json_options*.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`jsonpath`
Override the jsonpath schema location for the table.

`copy_json_options`
Add extra copy options, for example:

- GZIP
- LZOP

`copy`(*cursor*, *f*)
Defines copying JSON from s3 into redshift.

`task_namespace` = None

class `luigi.contrib.redshift.`**`RedshiftManifestTask`**(**args*, ***kwargs*)
Bases: *`luigi.s3.S3PathTask`*

Generic task to generate a manifest file that can be used in S3CopyToTable in order to copy multiple files from your s3 folder into a redshift table at once.

For full description on how to use the manifest file see http://docs.aws.amazon.com/redshift/latest/dg/loading-data-files-using-manifest.html

Usage:

- **requires parameters**
  - **path - s3 path to the generated manifest file, including the** name of the generated file to be copied into a redshift table
  - folder_paths - s3 paths to the folders containing files you wish to be copied

Output:

- generated manifest file

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`folder_paths` = <luigi.parameter.Parameter object>

`text_target` = True

run()

task_namespace = None

class luigi.contrib.redshift.**KillOpenRedshiftSessions**(**args*, ***kwargs*)
Bases: *luigi.task.Task*

An task for killing any open Redshift sessions in a given database. This is necessary to prevent open user sessions with transactions against the table from blocking drop or truncate table commands.

Usage:

Subclass and override the required *host*, *database*, *user*, and *password* attributes.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

connection_reset_wait_seconds = <luigi.parameter.IntParameter object>

host

database

user

password

update_id()
This update id will be a unique identifier for this insert on this table.

output()
Returns a RedshiftTarget representing the inserted dataset.

Normally you don't override this.

run()
Kill any open Redshift sessions for the given database.

task_namespace = None

luigi.contrib.scalding module

luigi.contrib.scalding.**logger = <logging.Logger object>**
Scalding support for Luigi.

Example configuration section in client.cfg:

```
[scalding]
# scala home directory, which should include a lib subdir with scala jars.
scala-home: /usr/share/scala

# scalding home directory, which should include a lib subdir with
# scalding-*-assembly-* jars as built from the official Twitter build script.
scalding-home: /usr/share/scalding

# provided dependencies, e.g. jars required for compiling but not executing
# scalding jobs. Currently requred jars:
# org.apache.hadoop/hadoop-core/0.20.2
# org.slf4j/slf4j-log4j12/1.6.6
# log4j/log4j/1.2.15
# commons-httpclient/commons-httpclient/3.1
# commons-cli/commons-cli/1.2
# org.apache.zookeeper/zookeeper/3.3.4
scalding-provided: /usr/share/scalding/provided

# additional jars required.
scalding-libjars: /usr/share/scalding/libjars
```

class luigi.contrib.scalding.**ScaldingJobRunner**

Bases: *luigi.contrib.hadoop.JobRunner*

JobRunner for *pyscald* commands. Used to run a ScaldingJobTask.

get_scala_jars(*include_compiler=False*)

get_scalding_jars()

get_scalding_core()

get_provided_jars()

get_libjars()

get_tmp_job_jar(*source*)

get_build_dir(*source*)

get_job_class(*source*)

build_job_jar(*job*)

run_job(*job*)

class luigi.contrib.scalding.**ScaldingJobTask**(**args*, ***kwargs*)

Bases: *luigi.contrib.hadoop.BaseHadoopJobTask*

A job task for Scalding that define a scala source and (optional) main method.

requires() should return a dictionary where the keys are Scalding argument names and values are sub tasks or lists of subtasks.

For example:

```
{'input1': A, 'input2': C} => --input1 <Aoutput> --input2 <Coutput>
{'input1': [A, B], 'input2': [C]} => --input1 <Aoutput> <Boutput> --input2 <Coutput>
```

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

relpath(*current_file*, *rel_path*)

Compute path given current file and relative path.

source()

Path to the scala source for this Scalding Job

Either one of source() or jar() must be specified.

jar()

Path to the jar file for this Scalding Job

Either one of source() or jar() must be specified.

extra_jars()

Extra jars for building and running this Scalding Job.

job_class()

optional main job class for this Scalding Job.

job_runner()

atomic_output()

If True, then rewrite output arguments to be temp locations and atomically move them into place after the job finishes.

requires()

job_args()

Extra arguments to pass to the Scalding job.

task_namespace = None

args()

Returns an array of args to pass to the job.

luigi.contrib.spark module

class luigi.contrib.spark.**SparkRunContext**(*proc*)

Bases: object

kill_job(*captured_signal=None*, *stack_frame=None*)

exception luigi.contrib.spark.**SparkJobError**(*message*, *out=None*, *err=None*)

Bases: exceptions.RuntimeError

class luigi.contrib.spark.**SparkSubmitTask**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Template task for running a Spark job

Supports running jobs on Spark local, standalone, Mesos or Yarn

See http://spark.apache.org/docs/latest/submitting-applications.html for more information

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

name = None

entry_class = None

app = None

app_options()

Subclass this method to map your task parameters to the app’s arguments

spark_submit

master

deploy_mode

jars

py_files

files

conf

properties_file

driver_memory

driver_java_options

driver_library_path

driver_class_path

executor_memory

driver_cores

supervise

total_executor_cores

executor_cores

queue

num_executors

archives

hadoop_conf_dir

get_environment()

spark_command()

app_command()

run()

task_namespace = None

class luigi.contrib.spark.**PySparkTask**(**args*, ***kwargs*)

Bases: *luigi.contrib.spark.SparkSubmitTask*

Template task for running an inline PySpark job

Simply implement the `main` method in your subclass

You can optionally define package names to be distributed to the cluster with `py_packages` (uses luigi's global py-packages configuration by default)

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

app = '/home/docs/checkouts/readthedocs.org/user_builds/luigi/checkouts/latest/luigi/contrib/pyspark_runner.py'

deploy_mode = 'client'

name

py_packages

setup (*conf*)

Called by the pyspark_runner with a SparkConf instance that will be used to instantiate the SparkContext

Parameters **conf** – SparkConf

setup_remote (*sc*)

main (*sc*, **args*)

Called by the pyspark_runner with a SparkContext and any arguments returned by `app_options()`

Parameters

- **sc** – SparkContext
- **args** – arguments list

app_command ()

run ()

task_namespace = None

class luigi.contrib.spark.**SparkJob** (**args*, ***kwargs*)

Bases: *luigi.task.Task*

Deprecated since version 1.1.1: Use `SparkSubmitTask` or `PySparkTask` instead.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

spark_workers = None

spark_master_memory = None

spark_worker_memory = None

queue = <luigi.parameter.Parameter object>

temp_hadoop_output_file = None

requires_local ()

Default impl - override this method if you need any local input to be accessible in init().

requires_hadoop ()

input_local ()

input ()

deps ()

jar ()

job_class ()

job_args ()

output ()

run ()

track_progress (*proc*)

task_namespace = None

class luigi.contrib.spark.**Spark1xBackwardCompat** (**args*, ***kwargs*)

Bases: *luigi.contrib.spark.SparkSubmitTask*

Adapts SparkSubmitTask interface to (Py)Spark1xJob interface

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

master

output ()

spark_options ()

dependency_jars ()

job_args ()

jars

app_options ()

spark_command ()

task_namespace = None

class luigi.contrib.spark.**Spark1xJob** (**args*, ***kwargs*)

Bases: *luigi.contrib.spark.Spark1xBackwardCompat*

Deprecated since version 1.1.1: Use `SparkSubmitTask` or `PySparkTask` instead.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

job_class ()

jar ()

entry_class

app

run ()

task_namespace = None

class luigi.contrib.spark.**PySpark1xJob** (**args*, ***kwargs*)

Bases: *luigi.contrib.spark.Spark1xBackwardCompat*

Deprecated since version 1.1.1: Use `SparkSubmitTask` or `PySparkTask` instead.

Constructor to resolve values for all Parameters.

For example, the Task:

```
    class MyTask(luigi.Task):
        count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

program()

app

run()

task_namespace = None

luigi.contrib.sparkey module

class luigi.contrib.sparkey.**SparkeyExportTask**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

A luigi task that writes to a local sparkey log file.

Subclasses should implement the requires and output methods. The output must be a luigi.LocalTarget.

The resulting sparkey log file will contain one entry for every line in the input, mapping from the first value to a tab-separated list of the rest of the line.

To generate a simple key-value index, yield “key”, “value” pairs from the input(s) to this task.

separator = ‘\t’

run()

task_namespace = None

luigi.contrib.sqla module Support for SQLAlchmey. Provides SQLAlchemyTarget for storing in databases supported by SQLAlchemy. The user would be responsible for installing the required database driver to connect using SQLAlchemy.

Minimal example of a job to copy data to database using SQLAlchemy is as shown below:

```
from sqlalchemy import String
import luigi
from luigi.contrib import sqla

class SQLATask(sqla.CopyToTable):
    # columns defines the table schema, with each element corresponding
    # to a column in the format (args, kwargs) which will be sent to
    # the sqlalchemy.Column(*args, **kwargs)
    columns = [
        (["item", String(64)], {"primary_key": True}),
        (["property", String(64)], {})
    ]
    connection_string = "sqlite://"  # in memory SQLite database
    table = "item_property"  # name of the table to store data

    def rows(self):
        for row in [("item1" "property1"), ("item2", "property2")]:
            yield row

if __name__ == '__main__':
    task = SQLATask()
    luigi.build([task], local_scheduler=True)
```

If the target table where the data needs to be copied already exists, then the column schema definition can be skipped and instead the reflect flag can be set as True. Here is a modified version of the above example:

```
from sqlalchemy import String
import luigi
from luigi.contrib import sqla

class SQLATask(sqla.CopyToTable):
    # If database table is already created, then the schema can be loaded
    # by setting the reflect flag to True
    reflect = True
    connection_string = "sqlite://"  # in memory SQLite database
    table = "item_property"  # name of the table to store data

    def rows(self):
        for row in [("item1" "property1"), ("item2", "property2")]:
            yield row

if __name__ == '__main__':
    task = SQLATask()
    luigi.build([task], local_scheduler=True)
```

In the above examples, the data that needs to be copied was directly provided by overriding the rows method. Alternately, if the data comes from another task, the modified example would look as shown below:

```
from sqlalchemy import String
import luigi
from luigi.contrib import sqla
from luigi.mock import MockFile

class BaseTask(luigi.Task):
    def output(self):
        return MockFile("BaseTask")

    def run(self):
        out = self.output().open("w")
        TASK_LIST = ["item%d\tproperty%d\n" % (i, i) for i in range(10)]
        for task in TASK_LIST:
            out.write(task)
        out.close()

class SQLATask(sqla.CopyToTable):
    # columns defines the table schema, with each element corresponding
    # to a column in the format (args, kwargs) which will be sent to
    # the sqlalchemy.Column(*args, **kwargs)
    columns = [
        (["item", String(64)], {"primary_key": True}),
        (["property", String(64)], {})
    ]
    connection_string = "sqlite://"  # in memory SQLite database
    table = "item_property"  # name of the table to store data

    def requires(self):
        return BaseTask()

if __name__ == '__main__':
    task1, task2 = SQLATask(), BaseTask()
    luigi.build([task1, task2], local_scheduler=True)
```

In the above example, the output from *BaseTask* is copied into the database. Here we did not have to implement the *rows* method because by default *rows* implementation assumes every line is a row with column values separated by a tab. One can define *column_separator* option for the task if the values are say comma separated instead of tab separated.

You can pass in database specific connection arguments by setting the connect_args dictionary. The options will be passed directly to the DBAPI's connect method as keyword arguments.

The other option to *sqla.CopyToTable* that can be of help with performance aspect is the *chunk_size*. The default is 5000. This is the number of rows that will be inserted in a transaction at a time. Depending on the size of the inserts, this value can be tuned for performance.

See here for a tutorial on building task pipelines using luigi and using SQLAlchemy in workflow pipelines.

Author: Gouthaman Balaraman Date: 01/02/2015

class `luigi.contrib.sqla.`**`SQLAlchemyTarget`**(*connection_string*, *target_table*, *update_id*, *echo=False*, *connect_args=None*)

Bases: *`luigi.target.Target`*

Database target using SQLAlchemy.

This will rarely have to be directly instantiated by the user.

Typical usage would be to override *luigi.contrib.sqla.CopyToTable* class to create a task to write to the database.

Constructor for the SQLAlchemyTarget.

Parameters

- **`connection_string`** (*str*) – SQLAlchemy connection string
- **`target_table`** (*str*) – The table name for the data
- **`update_id`** (*str*) – An identifier for this data set
- **`echo`** (*bool*) – Flag to setup SQLAlchemy logging
- **`connect_args`** (*dict*) – A dictionary of connection arguments

Returns

`marker_table` **= None**

class **`Connection`**(*engine*, *pid*)

Bases: `tuple`

`engine`

Alias for field number 0

`pid`

Alias for field number 1

`SQLAlchemyTarget.`**`connect_args`** **= {}**

`SQLAlchemyTarget.`**`engine`**

Return an engine instance, creating it if it doesn't exist.

Recreate the engine connection if it wasn't originally created by the current process.

`SQLAlchemyTarget.`**`touch`**()

Mark this update as complete.

`SQLAlchemyTarget.`**`exists`**()

SQLAlchemyTarget.**create_marker_table**()

Create marker table if it doesn't exist.

Using a separate connection since the transaction might have to be reset.

SQLAlchemyTarget.**open**(*mode*)

class luigi.contrib.sqla.**CopyToTable**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

An abstract task for inserting a data set into SQLAlchemy RDBMS

Usage:

- subclass and override the required *connection_string*, *table* and *columns* attributes.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

echo = False

connect_args = {}

connection_string()

table

columns = []

column_separator = '\t'

chunk_size = 5000

reflect = False

create_table(*engine*)

Override to provide code for creating the target table.

By default it will be created using types specified in columns. If the table exists, then it binds to the existing table.

If overridden, use the provided connection object for setting up the table in order to create the table and insert data using the same transaction. :param engine: The sqlalchemy engine instance :type engine: object

update_id()

This update id will be a unique identifier for this insert on this table.

output()

rows()

Return/yield tuples or lists corresponding to each row to be inserted.

This method can be overridden for custom file types or formats.

run()

copy(*conn*, *ins_rows*, *table_bound*)

This method does the actual insertion of the rows of data given by ins_rows into the database. A task that needs row updates instead of insertions should overload this method. :param conn: The sqlalchemy connection object :param ins_rows: The dictionary of rows with the keys in the format _<column_name>. For example if you have a table with a column name "property", then the key in the dictionary would be "_property". This format is consistent with the bindparam usage in sqlalchemy. :param table_bound: The object referring to the table :return:

task_namespace = None

luigi.contrib.ssh module Light-weight remote execution library and utilities.

There are some examples in the unittest, but I added another more luigi-specific in the examples directory (examples/ssh_remote_execution.py

`RemoteContext` is meant to provide functionality similar to that of the standard library subprocess module, but where the commands executed are run on a remote machine instead, without the user having to think about prefixing everything with "ssh" and credentials etc.

Using this mini library (which is just a convenience wrapper for subprocess), `RemoteTarget` is created to let you stream data from a remotely stored file using the luigi `FileSystemTarget` semantics.

As a bonus, `RemoteContext` also provides a really cool feature that let's you set up ssh tunnels super easily using a python context manager (there is an example in the integration part of unittests).

This can be super convenient when you want secure communication using a non-secure protocol or circumvent firewalls (as long as they are open for ssh traffic).

class `luigi.contrib.ssh.`**`RemoteContext`**(*host*, ***kwargs*)

Bases: `object`

`Popen`(*cmd*, ***kwargs*)

Remote Popen.

`check_output`(*cmd*)

Execute a shell command remotely and return the output.

Simplified version of Popen when you only want the output as a string and detect any errors.

`tunnel`(**args*, ***kwds*)

Open a tunnel between localhost:local_port and remote_host:remote_port via the host specified by this context.

Remember to close() the returned "tunnel" object in order to clean up after yourself when you are done with the tunnel.

class `luigi.contrib.ssh.`**`RemoteFileSystem`**(*host*, ***kwargs*)

Bases: `luigi.target.FileSystem`

`exists`(*path*)

Return *True* if file or directory at *path* exist, False otherwise.

`listdir`(*path*)

`isdir`(*path*)

Return *True* if directory at *path* exist, False otherwise.

`remove`(*path*, *recursive=True*)

Remove file or directory at location *path*.

`mkdir`(*path*, *parents=True*, *raise_if_exists=False*)

`put`(*local_path*, *path*)

`get`(*path*, *local_path*)

class `luigi.contrib.ssh.`**`AtomicRemoteFileWriter`**(*fs*, *path*)

Bases: `luigi.format.OutputPipeProcessWrapper`

close()

tmp_path

fs

class luigi.contrib.ssh.**RemoteTarget**(*path*, *host*, *format=None*, ***kwargs*)

Bases: *luigi.target.FileSystemTarget*

Target used for reading from remote files.

The target is implemented using ssh commands streaming data over the network.

fs

open(*mode='r'*)

put(*local_path*)

get(*local_path*)

luigi.contrib.target module

class luigi.contrib.target.**CascadingClient**(*clients*, *method_names=None*)

Bases: object

A FilesystemClient that will cascade failing function calls through a list of clients.

Which clients are used are specified at time of construction.

ALL_METHOD_NAMES = [’exists’, ‘rename’, ‘remove’, ‘chmod’, ‘chown’, ‘count’, ‘copy’, ‘get’, ‘put’, ‘mkdir’, ‘list’, ‘listdi

luigi.contrib.webhdfs module

Provides a *WebHdfsTarget* and *WebHdfsClient* using the Python hdfs

class luigi.contrib.webhdfs.**WebHdfsTarget**(*path*, *client=None*, *format=None*)

Bases: *luigi.target.FileSystemTarget*

fs = None

open(*mode='r'*)

class luigi.contrib.webhdfs.**ReadableWebHdfsFile**(*path*, *client*)

Bases: object

read()

readlines(*char='\n'*)

close()

class luigi.contrib.webhdfs.**AtomicWebHdfsFile**(*path*, *client*)

Bases: *luigi.target.AtomicLocalFile*

An Hdfs file that writes to a temp file and put to WebHdfs on close.

move_to_final_destination()

class luigi.contrib.webhdfs.**WebHdfsClient**(*host=None*, *port=None*, *user=None*)

Bases: object

get_config(*key*)

walk(*path*, *depth=1*)

exists(*path*)

Returns true if the path exists and false otherwise.

upload(*hdfs_path*, *local_path*, *overwrite=False*)

download(*hdfs_path*, *local_path*, *overwrite=False*, *n_threads=-1*)

remove(*hdfs_path*, *recursive=False*)

read(*hdfs_path*, *offset=0*, *length=None*, *buffer_size=None*, *chunk_size=1024*, *buffer_char=None*)

Module contents

Package containing optional and-on functionality.

luigi.tools package

Submodules

luigi.tools.deps module

luigi.tools.deps.**get_task_requires**(*task*)

luigi.tools.deps.**dfs_paths**(*start_task*, *goal_task_family*, *path=None*)

class luigi.tools.deps.**upstream**(**args*, ***kwargs*)

Bases: *luigi.task.Config*

Used to provide the parameter upstream-task-family

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

family = <luigi.parameter.Parameter object>

task_namespace = None

luigi.tools.deps.**find_deps**(*task*, *upstream_task_family*)

Finds all dependencies that start with the given task and have a path to upstream_task_family

Returns all deps on all paths between task and upstream

luigi.tools.deps.**find_deps_cli**()

Finds all tasks on all paths from provided CLI task

luigi.tools.deps.**main**()

luigi.tools.luigi_grep module

class luigi.tools.luigi_grep.**LuigiGrep**(*host*, *port*)

Bases: object

graph_url

prefix_search(*job_name_prefix*)

searches for jobs matching the given job_name_prefix.

status_search(*status*)

searches for jobs matching the given status

luigi.tools.luigi_grep.**main**()

luigi.tools.parse_task module

luigi.tools.parse_task.**id_to_name_and_params**(*task_id*)

Turn a task_id into a (task_family, {params}) tuple.

E.g. calling with `Foo(bar=bar, baz=baz)` returns `('Foo', {'bar': 'bar', 'baz': 'baz'})`.

luigi.tools.range module

Produces contiguous completed ranges of recurring tasks.

See RangeDaily and RangeHourly for basic usage.

Caveat - if gaps accumulate, their causes (e.g. missing dependencies) going unmonitored/unmitigated, then this will eventually keep retrying the same gaps over and over and make no progress to more recent times. (See 'task_limit' and 'reverse' parameters.) TODO foolproof against that kind of misuse?

class luigi.tools.range.**RangeEvent**

Bases: *luigi.event.Event*

Events communicating useful metrics.

COMPLETE_COUNT would normally be nondecreasing, and its derivative would describe performance (how many instances complete invocation-over-invocation).

COMPLETE_FRACTION reaching 1 would be a telling event in case of a backfill with defined start and stop. Would not be strikingly useful for a typical recurring task without stop defined, fluctuating close to 1.

DELAY is measured from the first found missing datehour till (current time + hours_forward), or till stop if it is defined. In hours for Hourly. TBD different units for other frequencies? TODO any different for reverse mode? From first missing till last missing? From last gap till stop?

COMPLETE_COUNT = 'event.tools.range.complete.count'

COMPLETE_FRACTION = 'event.tools.range.complete.fraction'

DELAY = 'event.tools.range.delay'

class luigi.tools.range.**RangeBase**(**args*, ***kwargs*)

Bases: *luigi.task.WrapperTask*

Produces a contiguous completed range of a recurring task.

Made for the common use case where a task is parameterized by e.g. DateParameter, and assurance is needed that any gaps arising from downtime are eventually filled.

Emits events that one can use to monitor gaps and delays.

At least one of start and stop needs to be specified.

(This is quite an abstract base class for subclasses with different datetime parameter class, e.g. DateParameter, DateHourParameter, ..., and different parameter naming, e.g. days_back/forward, hours_back/forward, ..., as well as different documentation wording, for good user experience.)

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

of = <luigi.parameter.Parameter object>

start = <luigi.parameter.Parameter object>

stop = <luigi.parameter.Parameter object>

reverse = <luigi.parameter.BoolParameter object>

task_limit = <luigi.parameter.IntParameter object>

now = <luigi.parameter.IntParameter object>

datetime_to_parameter(*dt*)

parameter_to_datetime(*p*)

moving_start(*now*)

Returns a datetime from which to ensure contiguousness in the case when start is None or unfeasibly far back.

moving_stop(*now*)

Returns a datetime till which to ensure contiguousness in the case when stop is None or unfeasibly far forward.

finite_datetimes(*finite_start*, *finite_stop*)

Returns the individual datetimes in interval [finite_start, finite_stop) for which task completeness should be required, as a sorted list.

requires()

missing_datetimes(*task_cls*, *finite_datetimes*)

Override in subclasses to do bulk checks.

Returns a sorted list.

This is a conservative base implementation that brutally checks completeness, instance by instance.

Inadvisable as it may be slow.

task_namespace = None

class luigi.tools.range.**RangeDailyBase**(**args*, ***kwargs*)

Bases: *luigi.tools.range.RangeBase*

Produces a contiguous completed range of a daily recurring task.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

start = <luigi.parameter.DateParameter object>

stop = <luigi.parameter.DateParameter object>

days_back = <luigi.parameter.IntParameter object>

days_forward = <luigi.parameter.IntParameter object>

datetime_to_parameter(*dt*)

parameter_to_datetime(*p*)

moving_start(*now*)

moving_stop(*now*)

finite_datetimes (*finite_start*, *finite_stop*)

Simply returns the points in time that correspond to turn of day.

task_namespace = None

class luigi.tools.range.**RangeHourlyBase** (**args*, ***kwargs*)

Bases: *luigi.tools.range.RangeBase*

Produces a contiguous completed range of an hourly recurring task.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

start = **<luigi.parameter.DateHourParameter object>**

stop = **<luigi.parameter.DateHourParameter object>**

hours_back = **<luigi.parameter.IntParameter object>**

hours_forward = **<luigi.parameter.IntParameter object>**

datetime_to_parameter (*dt*)

parameter_to_datetime (*p*)

moving_start (*now*)

moving_stop (*now*)

finite_datetimes (*finite_start*, *finite_stop*)

Simply returns the points in time that correspond to whole hours.

task_namespace = None

luigi.tools.range.**most_common** (*items*)

Wanted functionality from Counters (new in Python 2.7).

luigi.tools.range.**infer_bulk_complete_from_fs** (*datetimes*, *datetime_to_task*, *datetime_to_re*)

Efficiently determines missing datetimes by filesystem listing.

The current implementation works for the common case of a task writing output to a FileSystemTarget whose path is built using strftime with format like '...%Y...%m...%d...%H...', without custom complete() or exists().

(Eventually Luigi could have ranges of completion as first-class citizens. Then this listing business could be factored away/be provided for explicitly in target API or some kind of a history server.)

class luigi.tools.range.**RangeDaily** (**args*, ***kwargs*)

Bases: *luigi.tools.range.RangeDailyBase*

Efficiently produces a contiguous completed range of a daily recurring task that takes a single DateParameter.

Falls back to infer it from output filesystem listing to facilitate the common case usage.

Convenient to use even from command line, like:

```
luigi --module your.module RangeDaily --of YourActualTask --start 2014-01-01
```

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`missing_datetimes`(*task_cls*, *finite_datetimes*)

`task_namespace` = **None**

class `luigi.tools.range.`**`RangeHourly`**(**args*, ***kwargs*)

Bases: *`luigi.tools.range.RangeHourlyBase`*

Efficiently produces a contiguous completed range of an hourly recurring task that takes a single DateHourParameter.

Benefits from bulk_complete information to efficiently cover gaps.

Falls back to infer it from output filesystem listing to facilitate the common case usage.

Convenient to use even from command line, like:

```
luigi --module your.module RangeHourly --of YourActualTask --start 2014-01-01T00
```

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`missing_datetimes`(*task_cls*, *finite_datetimes*)

`task_namespace` = **None**

Module contents

Sort of a standard library for doing stuff with Tasks at a somewhat abstract level.

Submodule introduced to stop growing util.py unstructured.

11.1.2 Submodules

luigi.cmdline module

`luigi.cmdline.`**`luigi_run`**(*argv=[’-b’, ‘latex’, ‘-D’, ‘language=en’, ‘-d’, ‘_build/doctrees’, ‘.’, ‘_build/latex’]*)

`luigi.cmdline.`**`luigid`**(*argv=[’-b’, ‘latex’, ‘-D’, ‘language=en’, ‘-d’, ‘_build/doctrees’, ‘.’, ‘_build/latex’]*)

luigi.configuration module

luigi.configuration provides some convenience wrappers around Python’s ConfigParser to get configuration options from config files.

The default location for configuration files is client.cfg in the current working directory, then /etc/luigi/client.cfg.

Configuration has largely been superseded by parameters since they can do essentially everything configuration can do, plus a tighter integration with the rest of Luigi.

See Configuration for more info.

class `luigi.configuration.`**`LuigiConfigParser`**(*defaults=None*, *dict_type=<class 'collections.OrderedDict'>*, *allow_no_value=False*)

Bases: `ConfigParser.ConfigParser`

`NO_DEFAULT` **= <object object>**

classmethod **`add_config_path`**(*path*)

classmethod **`instance`**(**args*, ***kwargs*)

Singleton getter

classmethod **`reload`**()

`get`(*section*, *option*, *default=<object object>*, ***kwargs*)

`getboolean`(*section*, *option*, *default=<object object>*)

`getint`(*section*, *option*, *default=<object object>*)

`getfloat`(*section*, *option*, *default=<object object>*)

`getintdict`(*section*)

`set`(*section*, *option*, *value=None*)

`luigi.configuration.`**`get_config`**()

Convenience method (for backwards compatibility) for accessing config singleton.

luigi.date_interval module

`luigi.date_interval` provides convenient classes for date algebra. Everything uses ISO 8601 notation, i.e. YYYY-MM-DD for dates, etc. There is a corresponding *`luigi.parameter.DateIntervalParameter`* that you can use to parse date intervals.

Example:

```
class MyTask(luigi.Task):
    date_interval = luigi.DateIntervalParameter()
```

Now, you can launch this from the command line using `--date-interval 2014-05-10` or `--date-interval 2014-W26` (using week notation) or `--date-interval 2014` (for a year) and some other notations.

class `luigi.date_interval.`**`DateInterval`**(*date_a*, *date_b*)

Bases: `object`

The *`DateInterval`* is the base class with subclasses *`Date`*, *`Week`*, *`Month`*, *`Year`*, and *`Custom`*. Note that the *`DateInterval`* is abstract and should not be used directly: use *`Custom`* for arbitrary date intervals. The base class features a couple of convenience methods, such as `next()` which returns the next consecutive date interval.

Example:

```
x = luigi.date_interval.Week(2013, 52)
print x.prev()
```

This will print `2014-W01`.

All instances of *`DateInterval`* have attributes `date_a` and `date_b` set. This represents the half open range of the date interval. For instance, a May 2014 is represented as `date_a = 2014-05-01`, `date_b = 2014-06-01`.

`dates`()
: Returns a list of dates in this date interval.

`hours`()
: Same as dates() but returns 24 times more info: one for each hour.

`prev`()
: Returns the preceding corresponding date interval (eg. May -> April).

`next`()
: Returns the subsequent corresponding date interval (eg. 2014 -> 2015).

`to_string`()

classmethod `from_date`(*d*)
: Abstract class method.

For instance, `Month.from_date(datetime.date(2012, 6, 6))` returns a `Month(2012, 6)`.

classmethod `parse`(*s*)
: Abstract class method.

For instance, `Year.parse("2014")` returns a `Year(2014)`.

class `luigi.date_interval.`**`Date`**(*y*, *m*, *d*)

Bases: *`luigi.date_interval.DateInterval`*

Most simple *`DateInterval`* where `date_b == date_a + datetime.timedelta(1)`.

`to_string`()

classmethod `from_date`(*d*)

classmethod `parse`(*s*)

class `luigi.date_interval.`**`Week`**(*y*, *w*)

Bases: *`luigi.date_interval.DateInterval`*

ISO 8601 week. Note that it has some counterintuitive behavior around new year. For instance Monday 29 December 2008 is week 2009-W01, and Sunday 3 January 2010 is week 2009-W53 This example was taken from from http://en.wikipedia.org/wiki/ISO_8601#Week_dates

Python datetime does not have a method to convert from ISO weeks, so the constructor uses some stupid brute force

`to_string`()

classmethod `from_date`(*d*)

classmethod `parse`(*s*)

class `luigi.date_interval.`**`Month`**(*y*, *m*)

Bases: *`luigi.date_interval.DateInterval`*

`to_string`()

classmethod `from_date`(*d*)

classmethod `parse`(*s*)

class `luigi.date_interval.`**`Year`**(*y*)

Bases: *`luigi.date_interval.DateInterval`*

`to_string`()

classmethod **`from_date`**(*d*)

classmethod **`parse`**(*s*)

class `luigi.date_interval.`**`Custom`**(*date_a*, *date_b*)

Bases: *`luigi.date_interval.DateInterval`*

Custom date interval (does not implement prev and next methods)

Actually the ISO 8601 specifies <start>/<end> as the time interval format Not sure if this goes for date intervals as well. In any case slashes will most likely cause problems with paths etc.

`to_string`()

classmethod **`parse`**(*s*)

luigi.db_task_history module

Provides a database backend to the central scheduler. This lets you see historical runs. See *Enabling Task History* for information about how to turn out the task history feature.

class `luigi.db_task_history.`**`DbTaskHistory`**

Bases: *`luigi.task_history.TaskHistory`*

Task History that writes to a database using sqlalchemy. Also has methods for useful db queries.

`task_scheduled`(*task_id*)

`task_finished`(*task_id*, *successful*)

`task_started`(*task_id*, *worker_host*)

`find_all_by_parameters`(*task_name*, *session=None*, ***task_params*)

Find tasks with the given task_name and the same parameters as the kwargs.

`find_all_by_name`(*task_name*, *session=None*)

Find all tasks with the given task_name.

`find_latest_runs`(*session=None*)

Return tasks that have been updated in the past 24 hours.

`find_all_runs`(*session=None*)

Return all tasks that have been updated.

`find_all_events`(*session=None*)

Return all running/failed/done events.

`find_task_by_id`(*id*, *session=None*)

Find task with the given record ID.

class `luigi.db_task_history.`**`TaskParameter`**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Table to track luigi.Parameter()s of a Task.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

task_id

name

value

class `luigi.db_task_history.`**TaskEvent**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Table to track when a task is scheduled, starts, finishes, and fails.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id

task_id

event_name

ts

class `luigi.db_task_history.`**TaskRecord**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`

Base table to track information about a luigi.Task.

References to other tables are available through task.events, task.parameters, etc.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in `kwargs`.

Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

id

name

host

parameters

events

luigi.deprecate_kwarg module

`luigi.deprecate_kwarg.`**deprecate_kwarg**(*old_name*, *new_name*, *kw_value*)

Rename keyword arguments, but keep backwards compatibility.

Usage:

luigi.event module

Definitions needed for events. See *Events and callbacks* for info on how to use it.

class `luigi.event.`**`Event`**

Bases: `object`

`DEPENDENCY_DISCOVERED` = **'event.core.dependency.discovered'**

`DEPENDENCY_MISSING` = **'event.core.dependency.missing'**

`DEPENDENCY_PRESENT` = **'event.core.dependency.present'**

`BROKEN_TASK` = **'event.core.task.broken'**

`START` = **'event.core.start'**

`FAILURE` = **'event.core.failure'**

`SUCCESS` = **'event.core.success'**

`PROCESSING_TIME` = **'event.core.processing_time'**

luigi.file module

`LocalTarget` provides a concrete implementation of a *`Target`* class that uses files on the local file system

class `luigi.file.`**`atomic_file`**(*path*)

Bases: *`luigi.target.AtomicLocalFile`*

Simple class that writes to a temp file and moves it on close() Also cleans up the temp file if close is not invoked

`move_to_final_destination`()

`generate_tmp_path`(*path*)

class `luigi.file.`**`LocalFileSystem`**

Bases: *`luigi.target.FileSystem`*

Wrapper for access to file system operations.

Work in progress - add things as needed.

`exists`(*path*)

`mkdir`(*path*, *parents=True*, *raise_if_exists=False*)

`isdir`(*path*)

`listdir`(*path*)

`remove`(*path*, *recursive=True*)

class `luigi.file.`**`LocalTarget`**(*path=None*, *format=None*, *is_tmp=False*)

Bases: *`luigi.target.FileSystemTarget`*

`fs` = **<luigi.file.LocalFileSystem object>**

`makedirs`()

Create all parent folders if they do not exist.

`open`(*mode='r'*)

`move`(*new_path*, *raise_if_exists=False*)

`move_dir`(*new_path*)

remove()

copy(*new_path*, *raise_if_exists=False*)

fn

class luigi.file.**File**(**args*, ***kwargs*)
Bases: *luigi.file.LocalTarget*

luigi.format module

class luigi.format.**FileWrapper**(*file_object*)
Bases: object

Wrap *file* in a "real" so stuff can be added to it after creation.

class luigi.format.**InputPipeProcessWrapper**(*command*, *input_pipe=None*)
Bases: object

Initializes a InputPipeProcessWrapper instance.

Parameters command – a subprocess.Popen instance with stdin=input_pipe and stdout=subprocess.PIPE. Alternatively, just its args argument as a convenience.

create_subprocess(*command*)
http://www.chiark.greenend.org.uk/ucgi/~cjwatson/blosxom/2009-07-02-python-sigpipe.html

close()

readable()

writable()

seekable()

class luigi.format.**OutputPipeProcessWrapper**(*command*, *output_pipe=None*)
Bases: object

WRITES_BEFORE_FLUSH = 10000

write(**args*, ***kwargs*)

writeLine(*line*)

close()

abort()

readable()

writable()

seekable()

class luigi.format.**BaseWrapper**(*stream*, **args*, ***kwargs*)
Bases: object

class luigi.format.**NewlineWrapper**(*stream*, *newline=None*)
Bases: *luigi.format.BaseWrapper*

read(*n=-1*)

writelines(*lines*)

write(*b*)

class `luigi.format.`**`MixedUnicodeBytesWrapper`**(*stream*, *encoding=None*)

Bases: *`luigi.format.BaseWrapper`*

`write`(*b*)

`writelines`(*lines*)

class `luigi.format.`**`Format`**

Bases: `object`

Interface for format specifications.

classmethod **`pipe_reader`**(*input_pipe*)

classmethod **`pipe_writer`**(*output_pipe*)

class `luigi.format.`**`ChainFormat`**(**args*, ***kwargs*)

Bases: *`luigi.format.Format`*

`pipe_reader`(*input_pipe*)

`pipe_writer`(*output_pipe*)

class `luigi.format.`**`TextWrapper`**(*stream*, **args*, ***kwargs*)

Bases: `_io.TextIOWrapper`

class `luigi.format.`**`NopFormat`**

Bases: *`luigi.format.Format`*

`pipe_reader`(*input_pipe*)

`pipe_writer`(*output_pipe*)

class `luigi.format.`**`WrappedFormat`**(**args*, ***kwargs*)

Bases: *`luigi.format.Format`*

`pipe_reader`(*input_pipe*)

`pipe_writer`(*output_pipe*)

class `luigi.format.`**`TextFormat`**(**args*, ***kwargs*)

Bases: *`luigi.format.WrappedFormat`*

`input` **= 'unicode'**

`output` **= 'bytes'**

`wrapper_cls`

alias of *`TextWrapper`*

class `luigi.format.`**`MixedUnicodeBytesFormat`**(**args*, ***kwargs*)

Bases: *`luigi.format.WrappedFormat`*

`output` **= 'bytes'**

`wrapper_cls`

alias of *`MixedUnicodeBytesWrapper`*

class `luigi.format.`**`NewlineFormat`**(**args*, ***kwargs*)

Bases: *`luigi.format.WrappedFormat`*

`input` **= 'bytes'**

`output` **= 'bytes'**

`wrapper_cls`

alias of *`NewlineWrapper`*

class `luigi.format.`**`GzipFormat`**`(`*compression_level=None*`)`

Bases: `luigi.format.Format`

`input` **= 'bytes'**

`output` **= 'bytes'**

`pipe_reader``(`*input_pipe*`)`

`pipe_writer``(`*output_pipe*`)`

class `luigi.format.`**`Bzip2Format`**

Bases: `luigi.format.Format`

`input` **= 'bytes'**

`output` **= 'bytes'**

`pipe_reader``(`*input_pipe*`)`

`pipe_writer``(`*output_pipe*`)`

`luigi.format.`**`get_default_format`**`()`

luigi.hadoop module

luigi.hadoop has moved to `luigi.contrib.hadoop`

luigi.hadoop_jar module

luigi.hadoop_jar has moved to `luigi.contrib.hadoop_jar`

luigi.hdfs module

luigi.hdfs has moved to `luigi.contrib.hdfs`

luigi.hive module

The hive module has been moved to `luigi.contrib.hive`

luigi.interface module

This module contains the bindings for command line integration and dynamic loading of tasks

`luigi.interface.`**`setup_interface_logging`**`(`*conf_file=None*`)`

class `luigi.interface.`**`core`**`(`**args*, ***kwargs*`)`

Bases: `luigi.task.Config`

Keeps track of a bunch of environment params.

Uses the internal luigi parameter mechanism. The nice thing is that we can instantiate this class and get an object with all the environment variables set. This is arguably a bit of a hack.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

use_cmdline_section = False

local_scheduler = <luigi.parameter.BoolParameter object>

scheduler_host = <luigi.parameter.Parameter object>

scheduler_port = <luigi.parameter.IntParameter object>

lock_size = <luigi.parameter.IntParameter object>

no_lock = <luigi.parameter.BoolParameter object>

lock_pid_dir = <luigi.parameter.Parameter object>

workers = <luigi.parameter.IntParameter object>

logging_conf_file = <luigi.parameter.Parameter object>

module = <luigi.parameter.Parameter object>

parallel_scheduling = <luigi.parameter.BoolParameter object>

assistant = <luigi.parameter.BoolParameter object>

task_namespace = None

class luigi.interface.**WorkerSchedulerFactory**

Bases: object

create_local_scheduler()

create_remote_scheduler(*host*, *port*)

create_worker(*scheduler*, *worker_processes*, *assistant=False*)

class luigi.interface.**Interface**

Bases: object

parse()

static run(*tasks*, *worker_scheduler_factory=None*, *override_defaults=None*)

Parameters

- **tasks** –
- **worker_scheduler_factory** –
- **override_defaults** –

Returns True if all tasks and their dependencies were successfully run (or already completed); False if any error occurred.

luigi.interface.**error_task_names**(*task_name*, *task_names*)

luigi.interface.**add_task_parameters**(*parser*, *task_cls*, *optparse=False*)

luigi.interface.**get_global_parameters**()

luigi.interface.**add_global_parameters**(*parser*, *optparse=False*)

luigi.interface.**get_task_parameters**(*task_cls*, *args*)

luigi.interface.**set_global_parameters**(*args*)

class `luigi.interface.`**`ArgParseInterface`**

Bases: *`luigi.interface.Interface`*

Takes the task as the command, with parameters specific to it.

`parse_task`(*cmdline_args=None*, *main_task_cls=None*)

`parse`(*cmdline_args=None*, *main_task_cls=None*)

class `luigi.interface.`**`DynamicArgParseInterface`**

Bases: *`luigi.interface.ArgParseInterface`*

Uses –module as a way to load modules dynamically

Usage:

```
python whatever.py --module foo_module FooTask --blah xyz --x 123
```

This will dynamically import foo_module and then try to create FooTask from this.

`parse`(*cmdline_args=None*, *main_task_cls=None*)

class `luigi.interface.`**`PassThroughOptionParser`**(*usage=None*, *option_list=None*, *option_class=<class optparse.Option>*, *version=None*, *conflict_handler='error'*, *description=None*, *formatter=None*, *add_help_option=True*, *prog=None*, *epilog=None*)

Bases: `optparse.OptionParser`

An unknown option pass-through implementation of OptionParser.

When unknown arguments are encountered, bundle with largs and try again, until rargs is depleted.

sys.exit(status) will still be called if a known argument is passed incorrectly (e.g. missing arguments or bad argument types, etc.)

class `luigi.interface.`**`OptParseInterface`**(*existing_optparse*)

Bases: *`luigi.interface.Interface`*

Supported for legacy reasons where it's necessary to interact with an existing parser.

Takes the task using –task. All parameters to all possible tasks will be defined globally in a big unordered soup.

`parse`(*cmdline_args=None*, *main_task_cls=None*)

`luigi.interface.`**`run`**(*cmdline_args=None*, *existing_optparse=None*, *use_optparse=False*, *main_task_cls=None*, *worker_scheduler_factory=None*, *use_dynamic_argparse=False*, *local_scheduler=False*)

Run from cmdline.

The default parser uses argparse however, for legacy reasons, we support optparse that optionally allows for overriding an existing option parser with new args.

Parameters

- **`cmdline_args`** –
- **`existing_optparse`** –
- **`use_optparse`** –
- **`main_task_cls`** –
- **`worker_scheduler_factory`** –

- **use_dynamic_argparse** –
- **local_scheduler** –

luigi.interface.**build**(*tasks*, *worker_scheduler_factory=None*, ***env_params*)

Run internally, bypassing the cmdline parsing.

Useful if you have some luigi code that you want to run internally. Example:

```
luigi.build([MyTask1(), MyTask2()], local_scheduler=True)
```

One notable difference is that *build* defaults to not using the identical process lock. Otherwise, *build* would only be callable once from each process.

Parameters

- **tasks** –
- **worker_scheduler_factory** –
- **env_params** –

Returns

luigi.lock module

Locking functionality when launching things from the command line. Uses a pidfile. This prevents multiple identical workflows to be launched simultaneously.

luigi.lock.**getpcmd**(*pid*)

Returns command of process.

Parameters pid –

luigi.lock.**get_info**(*pid_dir*, *my_pid=None*)

luigi.lock.**acquire_for**(*pid_dir*, *num_available=1*)

Makes sure the process is only run once at the same time with the same name.

Notice that we since we check the process name, different parameters to the same command can spawn multiple processes at the same time, i.e. running "/usr/bin/my_process" does not prevent anyone from launching "/usr/bin/my_process –foo bar".

luigi.mock module

This moduel provides a class *MockTarget*, an implementation of *Target*. *MockTarget* contains all data in-memory. The main purpose is unit testing workflows without writing to disk.

class luigi.mock.**MockFileSystem**

Bases: *luigi.target.FileSystem*

MockFileSystem inspects/modifies _data to simulate file system operations.

get_all_data()

get_data(*fn*)

exists(*path*)

remove(*path*, *recursive=True*, *skip_trash=True*)

Removes the given mockfile. skip_trash doesn't have any meaning.

listdir(*path*)
listdir does a prefix match of self.get_all_data(), but doesn't yet support globs.

isdir(*path*)

mkdir(*path*, *parents=True*, *raise_if_exists=False*)
mkdir is a noop.

clear()

class luigi.mock.**MockTarget**(*fn*, *is_tmp=None*, *mirror_on_stderr=False*, *format=None*)
Bases: *luigi.target.FileSystemTarget*

fs = <luigi.mock.MockFileSystem object>

exists()

rename(*path*, *raise_if_exists=False*)

path

open(*mode*)

class luigi.mock.**MockFile**(**args*, ***kwargs*)
Bases: *luigi.mock.MockTarget*

luigi.mrrunner module

The hadoop runner.

This module contains the main() method which will be used to run the mapper and reducer on the Hadoop nodes.

class luigi.mrrunner.**Runner**(*job=None*)
Bases: object

Run the mapper or reducer on hadoop nodes.

extract_packages_archive()

run(*kind*, *stdin=<open file '<stdin>', mode 'r'>*, *stdout=<open file '<stdout>', mode 'w'>*)

luigi.mrrunner.**main**(*args=None*, *stdin=<open file '<stdin>', mode 'r'>*, *stdout=<open file '<stdout>', mode 'w'>*, *print_exception=<function print_exception>*)
Run either the mapper or the reducer from the class instance in the file "job-instance.pickle".

Arguments:

kind – is either map or reduce

luigi.mrrunner.**print_exception**(*exc*)

luigi.notifications module

Supports sending emails when tasks fail.

This needs some more documentation. See Configuration for configuration options. In particular using the config *error-email* should set up Luigi so that it will send emails when tasks fail.

```
[core]
error-email: foo@bar.baz
```

luigi.notifications.**email_type**()

luigi.notifications.**generate_email**(*sender*, *subject*, *message*, *recipients*, *image_png*) `luigi.notifications.`**`wrap_traceback`**`(`*traceback*`)`

`luigi.notifications.`**`send_email_smtp`**`(`*config*, *sender*, *subject*, *message*, *recipients*, *image_png*`)`

`luigi.notifications.`**`send_email_ses`**`(`*config*, *sender*, *subject*, *message*, *recipients*, *image_png*`)`

`luigi.notifications.`**`send_email_sendgrid`**`(`*config*, *sender*, *subject*, *message*, *recipients*, *image_png*`)`

`luigi.notifications.`**`send_email`**`(`*subject*, *message*, *sender*, *recipients*, *image_png=None*`)`

`luigi.notifications.`**`send_error_email`**`(`*subject*, *message*`)`

Sends an email to the configured error-email.

If no error-email is configured, then a message is logged.

luigi.parameter module

Parameters are one of the core concepts of Luigi. All Parameters sit on *`Task`* classes. See *Parameter* for more info on how to define parameters.

exception `luigi.parameter.`**`ParameterException`**

Bases: `exceptions.Exception`

Base exception.

exception `luigi.parameter.`**`MissingParameterException`**

Bases: *`luigi.parameter.ParameterException`*

Exception signifying that there was a missing Parameter.

exception `luigi.parameter.`**`UnknownParameterException`**

Bases: *`luigi.parameter.ParameterException`*

Exception signifying that an unknown Parameter was supplied.

exception `luigi.parameter.`**`DuplicateParameterException`**

Bases: *`luigi.parameter.ParameterException`*

Exception signifying that a Parameter was specified multiple times.

exception `luigi.parameter.`**`UnknownConfigException`**

Bases: *`luigi.parameter.ParameterException`*

Exception signifying that the `config_path` for the Parameter could not be found.

class `luigi.parameter.`**`Parameter`**`(`**args*, ***kwargs*`)`

Bases: `object`

An untyped Parameter

Parameters are objects set on the Task class level to make it possible to parameterize tasks. For instance:

class MyTask(luigi.Task): foo = luigi.Parameter()

This makes it possible to instantiate multiple tasks, eg `MyTask(foo='bar')` and `My(foo='baz')`. The task will then have the `foo` attribute set appropriately.

There are subclasses of `Parameter` that define what type the parameter has. This is not enforced within Python, but are used for command line interaction.

The `config_path` argument lets you specify a place where the parameter is read from config in case no value is provided.

When a task is instantiated, it will first use any argument as the value of the parameter, eg. if you instantiate a = TaskA(x=44) then a.x == 44. If this does not exist, it will use the value of the Parameter object, which is defined on a class level. This will be resolved in this order of falling priority:

- Any value provided on the command line on the class level (eg. `--TaskA-param xyz`)
- Any value provided via config (using the `config_path` argument)
- Any default value set using the `default` flag.

`counter` = 67
non-atomically increasing counter used for ordering parameters.

`has_value`
`True` if a default was specified or if config_path references a valid entry in the conf.

Note that "value" refers to the Parameter object itself - it can be either

1. The default value for this parameter
2. A value read from the config
3. A global value

Any Task instance can have its own value set that overrides this.

`value`
The value for this Parameter.

This refers to any value defined by a default, a config option, or a global value.

Raises MissingParameterException if a value is not set.

Returns the parsed value.

`has_task_value`(*task_name*, *param_name*)

`task_value`(*task_name*, *param_name*)

`set_global`(*value*)
Set the global value of this Parameter.

Parameters **`value`** – the new global value.

`reset_global`()

`parse`(*x*)
Parse an individual value from the input.

The default implementation is an identify (it returns `x`), but subclasses should override this method for specialized parsing. This method is called by *`parse_from_input()`* if `x` exists. If this Parameter was specified with `is_list=True`, then `parse` is called once for each item in the list.

Parameters **`x`** (*str*) – the value to parse.

Returns the parsed value.

`serialize`(*x*)
Opposite of *`parse()`*.

Converts the value `x` to a string.

Parameters **`x`** – the value to serialize.

`parse_from_input`(*param_name*, *x*, *task_name=None*)
Parses the parameter value from input `x`, handling defaults and is_list.

Parameters

- **param_name** – the name of the parameter. This is used for the message in `MissingParameterException`.
- **x** – the input value to parse.

Raises MissingParameterException if x is false-y and no default is specified.

serialize_to_input (*x*)

parser_dest (*param_name*, *task_name*, *glob=False*, *is_without_section=False*)

add_to_cmdline_parser (*parser*, *param_name*, *task_name*, *optparse=False*, *glob=False*, *is_without_section=False*)

parse_from_args (*param_name*, *task_name*, *args*, *params*)

set_global_from_args (*param_name*, *task_name*, *args*, *is_without_section=False*)

class `luigi.parameter.`**DateHourParameter** (**args*, ***kwargs*)

Bases: *luigi.parameter.Parameter*

Parameter whose value is a `datetime` specified to the hour.

A DateHourParameter is a ISO 8601 formatted date and time specified to the hour. For example, `2013-07-10T19` specifies July 10, 2013 at 19:00.

date_format = '%Y-%m-%dT%H'

parse (*s*)

Parses a string to a `datetime` using the format string `%Y-%m-%dT%H`.

serialize (*dt*)

Converts the datetime to a string usnig the format string `%Y-%m-%dT%H`.

class `luigi.parameter.`**DateMinuteParameter** (**args*, ***kwargs*)

Bases: *luigi.parameter.DateHourParameter*

Parameter whose value is a `datetime` specified to the minute.

A DateMinuteParameter is a ISO 8601 formatted date and time specified to the minute. For example, `2013-07-10T19H07` specifies July 10, 2013 at 19:07.

date_format = '%Y-%m-%dT%HH%M'

class `luigi.parameter.`**DateParameter** (**args*, ***kwargs*)

Bases: *luigi.parameter.Parameter*

Parameter whose value is a `date`.

A DateParameter is a Date string formatted `YYYY-MM-DD`. For example, `2013-07-10` specifies July 10, 2013.

parse (*s*)

Parses a date string formatted as `YYYY-MM-DD`.

class `luigi.parameter.`**IntParameter** (**args*, ***kwargs*)

Bases: *luigi.parameter.Parameter*

Parameter whose value is an `int`.

parse (*s*)

Parses an `int` from the string using `int()`.

class `luigi.parameter.`**FloatParameter** (**args*, ***kwargs*)

Bases: *luigi.parameter.Parameter*

Parameter whose value is a `float`.

parse(*s*)
Parses a `float` from the string using `float()`.

class `luigi.parameter.`**`BoolParameter`**(**args*, ***kwargs*)
Bases: *`luigi.parameter.Parameter`*

A Parameter whose value is a `bool`.

This constructor passes along args and kwargs to ctor for *`Parameter`* but specifies `is_bool=True`.

parse(*s*)
Parses a `bool` from the string, matching 'true' or 'false' ignoring case.

class `luigi.parameter.`**`BooleanParameter`**(**args*, ***kwargs*)
Bases: *`luigi.parameter.BoolParameter`*

class `luigi.parameter.`**`DateIntervalParameter`**(**args*, ***kwargs*)
Bases: *`luigi.parameter.Parameter`*

A Parameter whose value is a *`DateInterval`*.

Date Intervals are specified using the ISO 8601 Time Interval notation.

parse(*s*)
Parses a *:py:class:'~luigi.date_interval.DateInterval* from the input.

see ***`luigi.date_interval`*** for details on the parsing of DateIntervals.

class `luigi.parameter.`**`TimeDeltaParameter`**(**args*, ***kwargs*)
Bases: *`luigi.parameter.Parameter`*

Class that maps to timedelta using strings in any of the following forms:

- **`n {w[eek[s]]|d[ay[s]]|h[our[s]]|m[inute[s]|s[second[s]]}` (e.g. "1 week 2 days" or "1 h")**
  Note: multiple arguments must be supplied in longest to shortest unit order
- ISO 8601 duration `PnDTnHnMnS` (each field optional, years and months not supported)
- ISO 8601 duration `PnW`

See https://en.wikipedia.org/wiki/ISO_8601#Durations

parse(*input*)
Parses a time delta from the input.

See *`TimeDeltaParameter`* for details on supported formats.

luigi.postgres module

Implements a sublass of *`Target`* that writes data to Postgres. Also provides a helper task to copy data into a Postgres table.

class `luigi.postgres.`**`MultiReplacer`**(*replace_pairs*)
Bases: `object`

Object for one-pass replace of multiple words

Substituted parts will not be matched against other replace patterns, as opposed to when using multipass replace. The order of the items in the replace_pairs input will dictate replacement precedence.

Constructor arguments: replace_pairs – list of 2-tuples which hold strings to be replaced and replace string

Usage:

```
>>> replace_pairs = [("a", "b"), ("b", "c")]
>>> MultiReplacer(replace_pairs)("abcd")
'bccd'
>>> replace_pairs = [("ab", "x"), ("a", "x")]
>>> MultiReplacer(replace_pairs)("ab")
'x'
>>> replace_pairs.reverse()
>>> MultiReplacer(replace_pairs)("ab")
'xb'
```

Initializes a MultiReplacer instance.

Parameters **`replace_pairs`** (*tuple*) – list of 2-tuples which hold strings to be replaced and replace string.

class `luigi.postgres.`**`PostgresTarget`**`(`*host*, *database*, *user*, *password*, *table*, *update_id*`)`

Bases: *`luigi.target.Target`*

Target for a resource in Postgres.

This will rarely have to be directly instantiated by the user.

Args: host (str): Postgres server address. Possibly a host:port string. database (str): Database name user (str): Database user password (str): Password for specified user update_id (str): An identifier for this data set

`marker_table` **= 'table_updates'**

`use_db_timestamps` **= True**

`touch``(`*connection=None*`)`

Mark this update as complete.

Important: If the marker table doesn't exist, the connection transaction will be aborted and the connection reset. Then the marker table will be created.

`exists``(`*connection=None*`)`

`connect``()`

Get a psycopg2 connection object to the database where the table is.

`create_marker_table``()`

Create marker table if it doesn't exist.

Using a separate connection since the transaction might have to be reset.

`open``(`*mode*`)`

class `luigi.postgres.`**`CopyToTable`**`(`**args*, ***kwargs*`)`

Bases: *`luigi.contrib.rdbms.CopyToTable`*

Template task for inserting a data set into Postgres

Usage: Subclass and override the required *host*, *database*, *user*, *password*, *table* and *columns* attributes.

To customize how to access data from an input task, override the *rows* method with a generator that yields each row as a tuple with fields ordered according to *columns*.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

rows()

Return/yield tuples or lists corresponding to each row to be inserted.

map_column(*value*)

Applied to each column of every row returned by *rows*.

Default behaviour is to escape special characters and identify any self.null_values.

output()

Returns a PostgresTarget representing the inserted dataset.

Normally you don't override this.

copy(*cursor*, *file*)

run()

Inserts data generated by rows() into target table.

If the target table doesn't exist, self.create_table will be called to attempt to create the table.

Normally you don't want to override this.

task_namespace = None

luigi.process module

Contains some helper functions to run luigid in daemon mode

luigi.process.**check_pid**(*pidfile*)

luigi.process.**write_pid**(*pidfile*)

luigi.process.**get_log_format**()

luigi.process.**get_spool_handler**(*filename*)

luigi.process.**daemonize**(*cmd*, *pidfile=None*, *logdir=None*, *api_port=8082*, *address=None*)

luigi.rpc module

Implementation of the REST interface between the workers and the server. rpc.py implements the client side of it, server.py implements the server side. See Using the Central Scheduler for more info.

exception luigi.rpc.**RPCError**(*message*, *sub_exception=None*)

Bases: exceptions.Exception

class luigi.rpc.**RemoteScheduler**(*host='localhost'*, *port=8082*, *connect_timeout=None*, *url_prefix=''*)

Bases: *luigi.scheduler.Scheduler*

Scheduler proxy object. Talks to a RemoteSchedulerResponder.

ping(*worker*)

add_task(*worker*, *task_id*, *status='PENDING'*, *runnable=True*, *deps=None*, *new_deps=None*, *expl=None*, *resources=None*, *priority=0*, *family=''*, *module=None*, *params=None*, *assistant=False*)

get_work(*worker*, *host=None*, *assistant=False*)

graph()

dep_graph(*task_id*)

inverse_dep_graph(*task_id*)

task_list(*status*, *upstream_status*, *search=None*)

worker_list()

task_search(*task_str*)

fetch_error(*task_id*)

add_worker(*worker*, *info*)

update_resources(***resources*)

prune()

re_enable_task(*task_id*)

luigi.s3 module

Implementation of Simple Storage Service support. *S3Target* is a subclass of the Target class to support S3 file system operations

exception luigi.s3.**InvalidDeleteException**

Bases: *luigi.target.FileSystemException*

exception luigi.s3.**FileNotFoundException**

Bases: *luigi.target.FileSystemException*

class luigi.s3.**S3Client**(*aws_access_key_id=None*, *aws_secret_access_key=None*, ***kwargs*)

Bases: *luigi.target.FileSystem*

boto-powered S3 client.

exists(*path*)

Does provided path exist on S3?

remove(*path*, *recursive=True*)

Remove a file or directory from S3.

get_key(*path*)

put(*local_path*, *destination_s3_path*)

Put an object stored locally to an S3 path.

put_string(*content*, *destination_s3_path*)

Put a string to an S3 path.

put_multipart(*local_path*, *destination_s3_path*, *part_size=67108864*)

Put an object stored locally to an S3 path using S3 multi-part upload (for files > 5GB).

Parameters

- **local_path** – Path to source local file
- **destination_s3_path** – URL for target S3 location
- **part_size** – Part size in bytes. Default: 67108864 (64MB), must be >= 5MB and <= 5 GB.

copy(*source_path*, *destination_path*)

Copy an object from one S3 location to another.

rename(*source_path*, *destination_path*)

Rename/move an object from one S3 location to another.

listdir (*path*)
Get an iterable with S3 folder contents. Iterable contains paths relative to queried path.

list (*path*)

isdir (*path*)
Is the parameter S3 path a directory?

is_dir (*path*)
Is the parameter S3 path a directory?

mkdir (*path*, *parents=True*, *raise_if_exists=False*)

class luigi.s3.**AtomicS3File** (*path*, *s3_client*)
Bases: *luigi.target.AtomicLocalFile*

An S3 file that writes to a temp file and put to S3 on close.

move_to_final_destination ()

class luigi.s3.**ReadableS3File** (*s3_key*)
Bases: object

read (*size=0*)

close ()

readable ()

writable ()

seekable ()

class luigi.s3.**S3Target** (*path*, *format=None*, *client=None*)
Bases: *luigi.target.FileSystemTarget*

fs = None

open (*mode='r'*)

class luigi.s3.**S3FlagTarget** (*path*, *format=None*, *client=None*, *flag='_SUCCESS'*)
Bases: *luigi.s3.S3Target*

Defines a target directory with a flag-file (defaults to *_SUCCESS*) used to signify job success.

This checks for two things:

- the path exists (just like the S3Target)
- the _SUCCESS file exists within the directory.

Because Hadoop outputs into a directory and not a single file, the path is assumed to be a directory.

This is meant to be a handy alternative to AtomicS3File.

The AtomicFile approach can be burdensome for S3 since there are no directories, per se.

If we have 1,000,000 output files, then we have to rename 1,000,000 objects.

Initializes a S3FlagTarget.

Parameters

- **path** (*str*) – the directory where the files are stored.
- **client** –
- **flag** (*str*) –

fs = None

exists()

class luigi.s3.**S3EmrTarget**(**args*, ***kwargs*)
Bases: *luigi.s3.S3FlagTarget*

Deprecated. Use *S3FlagTarget*

class luigi.s3.**S3PathTask**(**args*, ***kwargs*)
Bases: *luigi.task.ExternalTask*

A external task that to require existence of a path in S3.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

path = <luigi.parameter.Parameter object>

output()

task_namespace = None

class luigi.s3.**S3EmrTask**(**args*, ***kwargs*)
Bases: *luigi.task.ExternalTask*

An external task that requires the existence of EMR output in S3.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

path = <luigi.parameter.Parameter object>

output()

task_namespace = None

class luigi.s3.**S3FlagTask**(**args*, ***kwargs*)
Bases: *luigi.task.ExternalTask*

An external task that requires the existence of EMR output in S3.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

task_namespace = None

path = <luigi.parameter.Parameter object>

flag = <luigi.parameter.Parameter object>

output()

luigi.scalding module

luigi.scalding has moved to luigi.contrib.scalding

luigi.scheduler module

The system for scheduling tasks and executing them in order. Deals with dependencies, priorities, resources, etc. The *Worker* pulls tasks from the scheduler (usually over the REST interface) and executes them. See Using the Central Scheduler for more info.

class luigi.scheduler.**Scheduler**

Bases: object

Abstract base class.

Note that the methods all take string arguments, not Task objects...

add_task = NotImplemented

get_work = NotImplemented

ping = NotImplemented

luigi.scheduler.**UPSTREAM_SEVERITY_KEY**()

T.index(value, [start, [stop]]) -> integer – return first index of value. Raises ValueError if the value is not present.

class luigi.scheduler.**scheduler**(**args*, ***kwargs*)

Bases: *luigi.task.Config*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

retry_delay = <luigi.parameter.FloatParameter object>

remove_delay = <luigi.parameter.FloatParameter object>

worker_disconnect_delay = <luigi.parameter.FloatParameter object>

state_path = <luigi.parameter.Parameter object>

disable_window = <luigi.parameter.IntParameter object>

disable_failures = <luigi.parameter.IntParameter object>

disable_hard_timeout = <luigi.parameter.IntParameter object>

disable_persist = <luigi.parameter.IntParameter object>

max_shown_tasks = <luigi.parameter.IntParameter object>

prune_done_tasks = <luigi.parameter.BoolParameter object>

record_task_history = <luigi.parameter.BoolParameter object>

visualization_graph = <luigi.parameter.Parameter object>

task_namespace = None

luigi.scheduler.**fix_time**(*x*)

class luigi.scheduler.**Failures**(*window*)
Bases: object

This class tracks the number of failures in a given time window.

Failures added are marked with the current timestamp, and this class counts the number of failures in a sliding time window ending at the present.

Initialize with the given window.

Parameters **window** – how long to track failures for, as a float (number of seconds).

add_failure()
Add a failure event with the current timestamp.

num_failures()
Return the number of failures in the window.

clear()
Clear the failure queue.

class luigi.scheduler.**Task**(*task_id*, *status*, *deps*, *resources=None*, *priority=0*, *family='‘*, *module=None*, *params=None*, *disable_failures=None*, *disable_window=None*, *disable_hard_timeout=None*)
Bases: object

add_failure()

has_excessive_failures()

can_disable()

class luigi.scheduler.**Worker**(*worker_id*, *last_active=None*)
Bases: object

Structure for tracking worker activity and keeping their references.

add_info(*info*)

update(*worker_reference*)

prune(*config*)

get_pending_tasks()

is_trivial_worker()
If it's not an assistant having only tasks that are without requirements

assistant

class luigi.scheduler.**SimpleTaskState**(*state_path*)
Bases: object

Keep track of the current state and handle persistance.

The point of this class is to enable other ways to keep state, eg. by using a database These will be implemented by creating an abstract base class that this and other classes inherit from.

dump()

load()

get_active_tasks(*status=None*)

get_running_tasks()

get_pending_tasks()

get_task(*task_id*, *default=None*, *setdefault=None*)

has_task(*task_id*)

re_enable(*task*, *config=None*)

set_status(*task*, *new_status*, *config=None*)

prune(*task*, *config*, *assistants*)

inactivate_tasks(*delete_tasks*)

get_active_workers(*last_active_lt=None*)

get_assistants(*last_active_lt=None*)

get_worker_ids()

get_worker(*worker_id*)

inactivate_workers(*delete_workers*)

get_necessary_tasks()

class luigi.scheduler.**CentralPlannerScheduler**(*config=None*, *resources=None*, *task_history_impl=None*, ***kwargs*)

Bases: *luigi.scheduler.Scheduler*

Async scheduler that can handle multiple workers, etc.

Can be run locally or on a server (using RemoteScheduler + server.Server).

Keyword Arguments: :param config: an object of class "scheduler" or None (in which the global instance will be used) :param resources: a dict of str->int constraints :param task_history_override: ignore config and use this object as the task history

load()

dump()

prune()

update(*worker_id*, *worker_reference=None*)

Keep track of whenever the worker was last active.

add_task(*task_id=None*, *status='PENDING'*, *runnable=True*, *deps=None*, *new_deps=None*, *expl=None*, *resources=None*, *priority=0*, *family=''*, *module=None*, *params=None*, *assistant=False*, ***kwargs*)

- add task identified by task_id if it doesn't exist
- if deps is not None, update dependency list
- update status of task
- add additional workers/stakeholders
- update priority when needed

add_worker(*worker*, *info*, ***kwargs*)

update_resources(***resources*)

get_work(*host=None*, *assistant=False*, ***kwargs*)

ping(***kwargs*)

graph(***kwargs*)

dep_graph(*task_id*, ***kwargs*)

task_list(*status*, *upstream_status*, *limit=True*, *search=None*, ***kwargs*)
Query for a subset of tasks by status.

worker_list(*include_running=True*, ***kwargs*)

inverse_dep_graph(*task_id*, ***kwargs*)

task_search(*task_str*, ***kwargs*)
Query for a subset of tasks by task_id.

Parameters **task_str** –

Returns

re_enable_task(*task_id*)

fetch_error(*task_id*, ***kwargs*)

task_history

luigi.server module

Simple REST server that takes commands in a JSON payload Interface to the `CentralPlannerScheduler` class. See Using the Central Scheduler for more info.

class luigi.server.**RPCHandler**(*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

Handle remote scheduling calls using rpc.RemoteSchedulerResponder.

initialize(*scheduler*)

get(*method*)

post(*method*)

class luigi.server.**BaseTaskHistoryHandler**(*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

initialize(*scheduler*)

get_template_path()

class luigi.server.**AllRunHandler**(*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get()

class luigi.server.**SelectedRunHandler**(*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get(*name*)

luigi.server.**from_utc**(*utcTime*, *fmt=None*)
convert UTC time string to time.struct_time: change datetime.datetime to time, return time.struct_time type

class luigi.server.**RecentRunHandler**(*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get()

class luigi.server.**ByNameHandler**(*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get (*name*)

class luigi.server.**ByIdHandler** (*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get (*id*)

class luigi.server.**ByParamsHandler** (*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get (*name*)

class luigi.server.**StaticFileHandler** (*application*, *request*, ***kwargs*)
Bases: tornado.web.RequestHandler

get (*path*)

class luigi.server.**RootPathHandler** (*application*, *request*, ***kwargs*)
Bases: *luigi.server.BaseTaskHistoryHandler*

get ()

luigi.server.**app** (*scheduler*)

luigi.server.**run** (*api_port=8082*, *address=None*, *scheduler=None*, *responder=None*)
Runs one instance of the API server.

luigi.server.**stop** ()

luigi.six module

luigi.target module

The abstract *Target* class. It is a central concept of Luigi and represents the state of the workflow.

class luigi.target.**Target**
Bases: object

A Target is a resource generated by a *Task*.

For example, a Target might correspond to a file in HDFS or data in a database. The Target interface defines one method that must be overridden: *exists()*, which signifies if the Target has been created or not.

Typically, a *Task* will define one or more Targets as output, and the Task is considered complete if and only if each of its output Targets exist.

exists ()
Returns True if the *Target* exists and False otherwise.

exception luigi.target.**FileSystemException**
Bases: exceptions.Exception

Base class for generic file system exceptions.

exception luigi.target.**FileAlreadyExists**
Bases: *luigi.target.FileSystemException*

Raised when a file system operation can't be performed because a directory exists but is required to not exist.

exception luigi.target.**MissingParentDirectory**
Bases: *luigi.target.FileSystemException*

Raised when a parent directory doesn't exist. (Imagine mkdir without -p)

exception luigi.target.**NotADirectory**

Bases: *luigi.target.FileSystemException*

Raised when a file system operation can't be performed because an expected directory is actually a file.

class luigi.target.**FileSystem**

Bases: object

FileSystem abstraction used in conjunction with *FileSystemTarget*.

Typically, a FileSystem is associated with instances of a *FileSystemTarget*. The instances of the py:class:*FileSystemTarget* will delegate methods such as *FileSystemTarget.exists()* and *FileSystemTarget.remove()* to the FileSystem.

Methods of FileSystem raise *FileSystemException* if there is a problem completing the operation.

exists(*path*)

Return True if file or directory at path exist, False otherwise

Parameters **path** (*str*) – a path within the FileSystem to check for existence.

remove(*path*, *recursive=True*, *skip_trash=True*)

Remove file or directory at location path

Parameters

- **path** (*str*) – a path within the FileSystem to remove.
- **recursive** (*bool*) – if the path is a directory, recursively remove the directory and all of its descendants. Defaults to True.

mkdir(*path*, *parents=True*, *raise_if_exists=False*)

Create directory at location path

Creates the directory at path and implicitly create parent directories if they do not already exist.

Parameters

- **path** (*str*) – a path within the FileSystem to create as a directory.
- **parents** (*bool*) – Create parent directories when necessary. When parents=False and the parent directory doesn't exist, raise luigi.target.MissingParentDirectory
- **raise_if_exists** (*bool*) – raise luigi.target.FileAlreadyExists if the folder already exists.

Note: This method is optional, not all FileSystem subclasses implements it.

***Note*: parents and raise_if_exists were added in August 2014. Some** implementations might not support these flags yet.

isdir(*path*)

Return True if the location at path is a directory. If not, return False.

Parameters **path** (*str*) – a path within the FileSystem to check as a directory.

Note: This method is optional, not all FileSystem subclasses implements it.

listdir(*path*)

Return a list of files rooted in path.

This returns an iterable of the files rooted at path. This is intended to be a recursive listing.

Parameters **path** (*str*) – a path within the FileSystem to list.

Note: This method is optional, not all FileSystem subclasses implements it.

class luigi.target.**FileSystemTarget**(*path*)

Bases: *luigi.target.Target*

Base class for FileSystem Targets like *LocalTarget* and HdfsTarget.

A FileSystemTarget has an associated *FileSystem* to which certain operations can be delegated. By default, *exists()* and *remove()* are delegated to the *FileSystem*, which is determined by the *fs()* property.

Methods of FileSystemTarget raise *FileSystemException* if there is a problem completing the operation.

Initializes a FileSystemTarget instance.

Parameters **path** (*str*) – the path associated with this FileSystemTarget.

fs

The *FileSystem* associated with this FileSystemTarget.

open(*mode*)

Open the FileSystem target.

This method returns a file-like object which can either be read from or written to depending on the specified mode.

Parameters **mode** (*str*) – the mode *r* opens the FileSystemTarget in read-only mode, whereas *w* will open the FileSystemTarget in write mode. Subclasses can implement additional options.

exists()

Returns True if the path for this FileSystemTarget exists; False otherwise.

This method is implemented by using *fs()*.

remove()

Remove the resource at the path specified by this FileSystemTarget.

This method is implemented by using *fs()*.

class luigi.target.**AtomicLocalFile**(*path*)

Bases: _io.BufferedWriter

Abstract class to create Target that create a tempoprary file in the local filesystem before moving it to there final destination

This class is just for the writing part of the Target. See *luigi.file.LocalTarget* for example

close()

generate_tmp_path(*path*)

move_to_final_destination()

tmp_path

luigi.task module

The abstract *Task* class. It is a central concept of Luigi and represents the state of the workflow. See Tasks for an overview.

luigi.task.**namespace**(*namespace=None*)

Call to set namespace of tasks declared after the call.

If called without arguments or with None as the namespace, the namespace is reset, which is recommended to do at the end of any file where the namespace is set to avoid unintentionally setting namespace on tasks outside of the scope of the current file.

The namespace of a Task can also be changed by specifying the property `task_namespace`. This solution has the advantage that the namespace doesn't have to be restored.

```
class Task2(luigi.Task):
    task_namespace = 'namespace2'
```

`luigi.task.`**`id_to_name_and_params`**`(`*task_id*`)`

exception `luigi.task.`**`BulkCompleteNotImplementedError`**

Bases: `exceptions.NotImplementedError`

This is here to trick pylint.

pylint thinks anything raising NotImplementedError needs to be implemented in any subclass. bulk_complete isn't like that. This tricks pylint into thinking that the default implementation is a valid implementation and no an abstract method.

class `luigi.task.`**`Task`**`(`**args*, ***kwargs*`)`

Bases: `object`

This is the base class of all Luigi Tasks, the base unit of work in Luigi.

A Luigi Task describes a unit or work.

The key methods of a Task, which must be implemented in a subclass are:

- *`run()`* - the computation done by this task.
- *`requires()`* - the list of Tasks that this Task depends on.
- *`output()`* - the output `Target` that this Task creates.

Parameters to the Task should be declared as members of the class, e.g.:

```
.. code-block:: python
```

class MyTask(luigi.Task): count = luigi.IntParameter()

Each Task exposes a constructor accepting all `Parameter` (and values) as kwargs. e.g. `MyTask(count=10)` would instantiate *MyTask*.

In addition to any declared properties and methods, there are a few non-declared properties, which are created by the `Register` metaclass:

`Task.task_namespace` optional string which is prepended to the task name for the sake of scheduling. If it isn't overridden in a Task, whatever was last declared using *luigi.namespace* will be used.

`Task._parameters` list of `(parameter_name, parameter)` tuples for this task class

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`priority` **= 0**

Priority of the task: the scheduler should favor available tasks with higher priority values first. See *Task priority*

`disabled` **= False**

resources = {}

Resources used by the task. Should be formatted like {"scp": 1} to indicate that the task requires 1 unit of the scp resource.

worker_timeout = None

Number of seconds after which to time out the run function. No timeout if set to 0. Defaults to 0 or value in client.cfg

use_cmdline_section

Property used by core config such as *–workers* etc. These will be exposed without the class as prefix.

classmethod **event_handler**(*event*)

Decorator for adding event handlers.

trigger_event(*event*, **args*, ***kwargs*)

Trigger that calls all of the specified events associated with this class.

task_module

Returns what Python module to import to get access to this class.

task_family = 'Task'

classmethod **get_params**()

Returns all of the Parameters for this Task.

classmethod **get_param_values**(*params*, *args*, *kwargs*)

Get the values of the parameters from the args and kwargs.

Parameters

- **params** – list of (param_name, Parameter).
- **args** – positional arguments
- **kwargs** – keyword arguments.

Returns list of *(name, value)* tuples, one for each parameter.

initialized()

Returns `True` if the Task is initialized and `False` otherwise.

classmethod **from_str_params**(*params_str=None*)

Creates an instance from a str->str hash.

Parameters **params_str** – dict of param name -> value.

to_str_params()

Convert all parameters to a str->str hash.

clone(*cls=None*, ***kwargs*)

Creates a new instance from an existing instance where some of the args have changed.

There's at least two scenarios where this is useful (see test/clone_test.py):

- remove a lot of boiler plate when you have recursive dependencies and lots of args
- there's task inheritance and some logic is on the base class

Parameters

- **cls** –
- **kwargs** –

Returns

complete()

If the task has any outputs, return `True` if all outputs exists. Otherwise, return `False`.

However, you may freely override this method with custom logic.

classmethod **bulk_complete**(*parameter_tuples*)

Returns those of parameter_tuples for which this Task is complete.

Override (with an efficient implementation) for efficient scheduling with range tools. Keep the logic consistent with that of complete().

output()

The output that this Task produces.

The output of the Task determines if the Task needs to be run–the task is considered finished iff the outputs all exist. Subclasses should override this method to return a single `Target` or a list of `Target` instances.

Implementation note If running multiple workers, the output must be a resource that is accessible by all workers, such as a DFS or database. Otherwise, workers might compute the same output since they don't see the work done by other workers.

See *Task.output*

requires()

The Tasks that this Task depends on.

A Task will only run if all of the Tasks that it requires are completed. If your Task does not require any other Tasks, then you don't need to override this method. Otherwise, a Subclasses can override this method to return a single Task, a list of Task instances, or a dict whose values are Task instances.

See *Task.requires*

process_resources()

Override in "template" tasks which provide common resource functionality but allow subclasses to specify additional resources while preserving the name for consistent end-user experience.

input()

Returns the outputs of the Tasks returned by `requires()`

See *Task.input*

Returns a list of `Target` objects which are specified as outputs of all required Tasks.

deps()

Internal method used by the scheduler.

Returns the flattened list of requires.

run()

The task run method, to be overridden in a subclass.

See *Task.run*

on_failure(*exception*)

Override for custom error handling.

This method gets called if an exception is raised in `run()`. Return value of this method is json encoded and sent to the scheduler as the *expl* argument. Its string representation will be used as the body of the error email sent out if any.

Default behavior is to return a string representation of the stack trace.

on_success()

Override for doing custom completion handling for a larger class of tasks

This method gets called when *run()* completes without raising any exceptions.

The returned value is json encoded and sent to the scheduler as the *expl* argument.

Default behavior is to send an None value

task_namespace = None

class luigi.task.**MixinNaiveBulkComplete**

Bases: object

Enables a Task to be efficiently scheduled with e.g. range tools, by providing a bulk_complete implementation which checks completeness in a loop.

Applicable to tasks whose completeness checking is cheap.

This doesn't exploit output location specific APIs for speed advantage, nevertheless removes redundant scheduler roundtrips.

classmethod bulk_complete(*parameter_tuples*)

luigi.task.**externalize**(*task*)

Returns an externalized version of the Task.

See *ExternalTask*.

class luigi.task.**ExternalTask**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Subclass for references to external dependencies.

An ExternalTask's does not have a *run* implementation, which signifies to the framework that this Task's output() is generated outside of Luigi.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

run = NotImplemented

task_namespace = None

class luigi.task.**WrapperTask**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Use for tasks that only wrap other tasks and that by definition are done if all their requirements exist.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

complete()

task_namespace = None

class luigi.task.**Config**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Used for configuration that's not specific to a certain task

TODO: let's refactor Task & Config so that it inherits from a common ParamContainer base class

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

task_namespace = None

luigi.task.**getpaths**(*struct*)

Maps all Tasks in a structured data object to their .output().

luigi.task.**flatten**(*struct*)

Creates a flat list of all all items in structured output (dicts, lists, items):

```
>>> sorted(flatten({'a': 'foo', 'b': 'bar'}))
['bar', 'foo']
>>> sorted(flatten(['foo', ['bar', 'troll']]))
['bar', 'foo', 'troll']
>>> flatten('foo')
['foo']
>>> flatten(42)
[42]
```

luigi.task.**flatten_output**(*task*)

Lists all output targets by recursively walking output-less (wrapper) tasks.

FIXME order consistently.

luigi.task_history module

Abstract class for task history. Currently the only subclass is *DbTaskHistory*.

class luigi.task_history.**Task**(*task_id*, *status*, *host=None*)

Bases: object

Interface for methods on TaskHistory

class luigi.task_history.**TaskHistory**

Bases: object

Abstract Base Class for updating the run history of a task

task_scheduled(*task_id*)

task_finished(*task_id*, *successful*)

task_started(*task_id*, *worker_host*)

class luigi.task_history.**NopHistory**

Bases: *luigi.task_history.TaskHistory*

task_scheduled(*task_id*)

task_finished(*task_id*, *successful*)

task_started(*task_id*, *worker_host*)

luigi.task_register module

Define the centralized register of all *Task* classes.

exception luigi.task_register.**TaskClassException**
Bases: exceptions.Exception

class luigi.task_register.**Register**
Bases: abc.ABCMeta

The Metaclass of Task.

Acts as a global registry of Tasks with the following properties:

1.Cache instances of objects so that eg. X(1, 2, 3) always returns the same object.

2.Keep track of all subclasses of Task and expose them.

AMBIGUOUS_CLASS = <object object>
If this value is returned by _get_reg() then there is an ambiguous task name (two Task have the same name). This denotes an error.

classmethod clear_instance_cache()
Clear/Reset the instance cache.

classmethod disable_instance_cache()
Disables the instance cache.

task_family
The task family for the given class.

If cls.task_namespace is None then it's the name of the class. Otherwise, <task_namespace>. is prefixed to the class name.

classmethod task_names()
List of task names as strings

classmethod tasks_str()
Human-readable register contents dump.

classmethod get_task_cls(*name*)
Returns an unambiguous class or raises an exception.

classmethod get_all_params()
Compiles and returns all parameters for all Task.

Returns a generator of tuples (TODO: we should make this more elegant)

luigi.task_register.**load_task**(*module*, *task_name*, *params_str*)
Imports task dynamically given a module and a task name.

luigi.task_status module

Possible values for a Task's status in the Scheduler

luigi.util module

luigi.util.**common_params**(*task_instance*, *task_cls*)

Grab all the values in task_instance that are found in task_cls.

luigi.util.**task_wraps**(*P*)

class luigi.util.**inherits**(*task_to_inherit*)

Bases: object

Task inheritance.

Usage:

```
class AnotherTask(luigi.Task):
    n = luigi.IntParameter()
    # ...

@inherits(AnotherTask):
class MyTask(luigi.Task):
    def requires(self):
       return self.clone_parent()

    def run(self):
       print self.n # this will be defined
       # ...
```

class luigi.util.**requires**(*task_to_require*)

Bases: object

Same as @inherits, but also auto-defines the requires method.

class luigi.util.**copies**(*task_to_copy*)

Bases: object

Auto-copies a task.

Usage:

```
@copies(MyTask):
class CopyOfMyTask(luigi.Task):
    def output(self):
       return LocalTarget(self.date.strftime('/var/xyz/report-%Y-%m-%d'))
```

luigi.util.**delegates**(*task_that_delegates*)

Lets a task call methods on subtask(s).

The way this works is that the subtask is run as a part of the task, but the task itself doesn't have to care about the requirements of the subtasks. The subtask doesn't exist from the scheduler's point of view, and its dependencies are instead required by the main task.

Example:

```
class PowersOfN(luigi.Task):
    n = luigi.IntParameter()
    def f(self, x): return x ** self.n

@delegates
class T(luigi.Task):
    def subtasks(self): return PowersOfN(5)
    def run(self): print self.subtasks().f(42)
```

luigi.util.**previous**(*task*)

Return a previous Task of the same family.

By default checks if this task family only has one non-global parameter and if it is a DateParameter, DateHourParameter or DateIntervalParameter in which case it returns with the time decremented by 1 (hour, day or interval)

luigi.util.**get_previous_completed**(*task*, *max_steps=10*)

luigi.webhdfs module

luigi.webhdfs has moved to luigi.contrib.webhdfs

luigi.worker module

The worker communicates with the scheduler and does two things:

1. Sends all tasks that has to be run
2. Gets tasks from the scheduler that should be run

When running in local mode, the worker talks directly to a `CentralPlannerScheduler` instance. When you run a central server, the worker will talk to the scheduler using a `RemoteScheduler` instance.

exception luigi.worker.**TaskException**

Bases: `exceptions.Exception`

class luigi.worker.**TaskProcess**(*task*, *worker_id*, *result_queue*, *random_seed=False*, *worker_timeout=0*)

Bases: `multiprocessing.process.Process`

Wrap all task execution in this class.

Mainly for convenience since this is run in a separate process.

run()

class luigi.worker.**SingleProcessPool**

Bases: `object`

Dummy process pool for using a single processor.

Imitates the api of multiprocessing.Pool using single-processor equivalents.

apply_async(*function*, *args*)

class luigi.worker.**DequeQueue**

Bases: `collections.deque`

deque wrapper implementing the Queue interface.

put()

Add an element to the right side of the deque.

get()

Remove and return the rightmost element.

exception luigi.worker.**AsyncCompletionException**(*trace*)

Bases: `exceptions.Exception`

Exception indicating that something went wrong with checking complete.

class `luigi.worker.`**`TracebackWrapper`**`(`*trace*`)`

Bases: `object`

Class to wrap tracebacks so we can know they're not just strings.

`luigi.worker.`**`check_complete`**`(`*task*, *out_queue*`)`

Checks if task is complete, puts the result to out_queue.

class `luigi.worker.`**`worker`**`(`**args*, ***kwargs*`)`

Bases: *`luigi.task.Config`*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`ping_interval` **= <luigi.parameter.FloatParameter object>**

`keep_alive` **= <luigi.parameter.BoolParameter object>**

`count_uniques` **= <luigi.parameter.BoolParameter object>**

`wait_interval` **= <luigi.parameter.IntParameter object>**

`max_reschedules` **= <luigi.parameter.IntParameter object>**

`timeout` **= <luigi.parameter.IntParameter object>**

`task_limit` **= <luigi.parameter.IntParameter object>**

`retry_external_tasks` **= <luigi.parameter.BoolParameter object>**

`task_namespace` **= None**

class `luigi.worker.`**`KeepAliveThread`**`(`*scheduler*, *worker_id*, *ping_interval*`)`

Bases: `threading.Thread`

Periodically tell the scheduler that the worker still lives.

`stop``()`

`run``()`

class `luigi.worker.`**`Worker`**`(`*scheduler=None*, *worker_id=None*, *worker_processes=1*, *assistant=False*, ***kwargs*`)`

Bases: `object`

Worker object communicates with a scheduler.

Simple class that talks to a scheduler and:

- tells the scheduler what it has to do + its dependencies
- asks for stuff to do (pulls it in a loop and runs it)

`stop``()`

Stop the KeepAliveThread associated with this Worker.

This should be called whenever you are done with a worker instance to clean up.

Warning: this should _only_ be performed if you are sure this worker is not performing any work or will perform any work after this has been called

TODO: also kill all currently running tasks

TODO (maybe): Worker should be/have a context manager to enforce calling this whenever you stop using a Worker instance

add(*task*, *multiprocess=False*)

Add a Task for the worker to check and possibly schedule and run.

Returns True if task and its dependencies were successfully scheduled or completed before.

run()

Returns True if all scheduled tasks were executed successfully.

11.1.3 Module contents

Package containing core luigi functionality.

11.2 luigi.contrib package

11.2.1 Subpackages

luigi.contrib.hdfs package

Submodules

luigi.contrib.hdfs.abstract_client module Module containing abstract class about hdfs clients.

class luigi.contrib.hdfs.abstract_client.**HdfsFileSystem**

Bases: *luigi.target.FileSystem*

This client uses Apache 2.x syntax for file system commands, which also matched CDH4.

rename(*path*, *dest*)

Rename or move a file

rename_dont_move(*path*, *dest*)

Override this method with an implementation that uses rename2, which is a rename operation that never moves.

For instance, *rename2 a b* never moves *a* into *b* folder.

Currently, the hadoop cli does not support this operation.

We keep the interface simple by just aliasing this to normal rename and let individual implementations redefine the method.

rename2 - https://github.com/apache/hadoop/blob/ae91b13/hadoop-hdfs-project/hadoop-hdfs/src/main/java/org/apache/hadoop/hdfs/protocol/ClientProtocol.java (lines 483-523)

remove(*path*, *recursive=True*, *skip_trash=False*)

chmod(*path*, *permissions*, *recursive=False*)

chown(*path*, *owner*, *group*, *recursive=False*)

count(*path*)

Count contents in a directory

copy(*path*, *destination*)

put(*local_path*, *destination*)

get (*path*, *local_destination*)

mkdir (*path*, *parents=True*, *raise_if_exists=False*)

listdir (*path*, *ignore_directories=False*, *ignore_files=False*, *include_size=False*, *include_type=False*, *include_time=False*, *recursive=False*)

touchz (*path*)

luigi.contrib.hdfs.clients module The implementations of the hdfs clients. The hadoop cli client and the snakebite client.

luigi.contrib.hdfs.clients.**get_autoconfig_client** (*show_warnings=True*)
Creates the client as specified in the *client.cfg* configuration.

luigi.contrib.hdfs.config module You can configure what client by setting the "client" config under the "hdfs" section in the configuration, or using the `--hdfs-client` command line option. "hadoopcli" is the slowest, but should work out of the box. "snakebite" is the fastest, but requires Snakebite to be installed.

class luigi.contrib.hdfs.config.**hdfs** (**args*, ***kwargs*)
Bases: *luigi.task.Config*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

client_version = <luigi.parameter.IntParameter object>

effective_user = <luigi.parameter.Parameter object>

snakebite_autoconfig = <luigi.parameter.BoolParameter object>

namenode_host = <luigi.parameter.Parameter object>

namenode_port = <luigi.parameter.IntParameter object>

client = <luigi.parameter.Parameter object>

tmp_dir = <luigi.parameter.Parameter object>

task_namespace = None

class luigi.contrib.hdfs.config.**hadoopcli** (**args*, ***kwargs*)
Bases: *luigi.task.Config*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

command = <luigi.parameter.Parameter object>

version = <luigi.parameter.Parameter object>

task_namespace = None

luigi.contrib.hdfs.config.**load_hadoop_cmd**()

luigi.contrib.hdfs.config.**get_configured_hadoop_version**()

CDH4 (hadoop 2+) has a slightly different syntax for interacting with hdfs via the command line.

The default version is CDH4, but one can override this setting with "cdh3" or "apache1" in the hadoop section of the config in order to use the old syntax.

luigi.contrib.hdfs.config.**get_configured_hdfs_client**(*show_warnings=True*)

This is a helper that fetches the configuration value for 'client' in the [hdfs] section. It will return the client that retains backwards compatibility when 'client' isn't configured.

luigi.contrib.hdfs.config.**tmppath**(*path=None*, *include_unix_username=True*)

@param path: target path for which it is needed to generate temporary location @type path: str @type include_unix_username: bool @rtype: str

Note that include_unix_username might work on windows too.

luigi.contrib.hdfs.error module The implementations of the hdfs clients. The hadoop cli client and the snakebite client.

exception luigi.contrib.hdfs.error.**HDFSCliError**(*command*, *returncode*, *stdout*, *stderr*)

Bases: exceptions.Exception

luigi.contrib.hdfs.format module

class luigi.contrib.hdfs.format.**HdfsReadPipe**(*path*)

Bases: *luigi.format.InputPipeProcessWrapper*

class luigi.contrib.hdfs.format.**HdfsAtomicWritePipe**(*path*)

Bases: *luigi.format.OutputPipeProcessWrapper*

File like object for writing to HDFS

The referenced file is first written to a temporary location and then renamed to final location on close(). If close() isn't called the temporary file will be cleaned up when this object is garbage collected

TODO: if this is buggy, change it so it first writes to a local temporary file and then uploads it on completion

abort()

close()

class luigi.contrib.hdfs.format.**HdfsAtomicWriteDirPipe**(*path*, *data_extension='‘*)

Bases: *luigi.format.OutputPipeProcessWrapper*

Writes a data<data_extension> file to a directory at <path>.

abort()

close()

class luigi.contrib.hdfs.format.**PlainFormat**

Bases: *luigi.format.Format*

input = 'bytes'

output = 'hdfs'

hdfs_writer(*path*)

hdfs_reader(*path*)

pipe_reader(*path*)

pipe_writer(*output_pipe*)

class luigi.contrib.hdfs.format.**PlainDirFormat**

Bases: *luigi.format.Format*

input = 'bytes'

output = 'hdfs'

hdfs_writer (*path*)

hdfs_reader (*path*)

pipe_reader (*path*)

pipe_writer (*path*)

class luigi.contrib.hdfs.format.**CompatibleHdfsFormat** (*writer*, *reader*, *input=None*)

Bases: *luigi.format.Format*

output = 'hdfs'

pipe_writer (*output*)

pipe_reader (*input*)

hdfs_writer (*output*)

hdfs_reader (*input*)

luigi.contrib.hdfs.hadoopcli_clients module The implementations of the hdfs clients. The hadoop cli client and the snakebite client.

luigi.contrib.hdfs.hadoopcli_clients.**create_hadoopcli_client** ()

Given that we want one of the hadoop cli clients (unlike snakebite), this one will return the right one.

class luigi.contrib.hdfs.hadoopcli_clients.**HdfsClient**

Bases: *luigi.contrib.hdfs.abstract_client.HdfsFileSystem*

This client uses Apache 2.x syntax for file system commands, which also matched CDH4.

recursive_listdir_cmd = ['-ls', '-R']

static call_check (*command*)

exists (*path*)

Use `hadoop fs -stat` to check file existence.

rename (*path*, *dest*)

remove (*path*, *recursive=True*, *skip_trash=False*)

chmod (*path*, *permissions*, *recursive=False*)

chown (*path*, *owner*, *group*, *recursive=False*)

count (*path*)

copy (*path*, *destination*)

put (*local_path*, *destination*)

get (*path*, *local_destination*)

getmerge (*path*, *local_destination*, *new_line=False*)

mkdir (*path*, *parents=True*, *raise_if_exists=False*)

listdir (*path*, *ignore_directories=False*, *ignore_files=False*, *include_size=False*, *include_type=False*, *include_time=False*, *recursive=False*)

touchz (*path*)

class luigi.contrib.hdfs.hadoopcli_clients.**HdfsClientCdh3**

Bases: *luigi.contrib.hdfs.hadoopcli_clients.HdfsClient*

This client uses CDH3 syntax for file system commands.

mkdir (*path*)

No -p switch, so this will fail creating ancestors.

remove (*path*, *recursive=True*, *skip_trash=False*)

class luigi.contrib.hdfs.hadoopcli_clients.**HdfsClientApache1**

Bases: *luigi.contrib.hdfs.hadoopcli_clients.HdfsClientCdh3*

This client uses Apache 1.x syntax for file system commands, which are similar to CDH3 except for the file existence check.

recursive_listdir_cmd = ['-lsr']

exists (*path*)

luigi.contrib.hdfs.snakebite_client module A luigi file system client that wraps around snakebite

Originally written by Alan Brenner <alan@magnetic.com> github.com/alanbbr

class luigi.contrib.hdfs.snakebite_client.**SnakebiteHdfsClient**

Bases: *luigi.contrib.hdfs.abstract_client.HdfsFileSystem*

A hdfs client using snakebite. Since Snakebite has a python API, it'll be about 100 times faster than the hadoop cli client, which does shell out to a java program on each file system operation.

static list_path (*path*)

get_bite ()

If Luigi has forked, we have a different PID, and need to reconnect.

exists (*path*)

Use snakebite.test to check file existence.

Parameters **path** (*string*) – path to test

Returns boolean, True if path exists in HDFS

rename (*path*, *dest*)

Use snakebite.rename, if available.

Parameters

- **path** (*either a string or sequence of strings*) – source file(s)
- **dest** (*string*) – destination file (single input) or directory (multiple)

Returns list of renamed items

rename_dont_move (*path*, *dest*)

Use snakebite.rename_dont_move, if available.

Parameters

- **path** (*string*) – source path (single input)
- **dest** (*string*) – destination path

Returns True if succeeded

Raises snakebite.errors.FileAlreadyExistsException

remove (*path*, *recursive=True*, *skip_trash=False*)

Use snakebite.delete, if available.

Parameters

- **path** (*either a string or a sequence of strings*) – delete-able file(s) or directory(ies)
- **recursive** (*boolean, default is True*) – delete directories trees like *nix: rm -r
- **skip_trash** (*boolean, default is False (use trash)*) – do or don't move deleted items into the trash first

Returns list of deleted items

chmod (*path*, *permissions*, *recursive=False*)

Use snakebite.chmod, if available.

Parameters

- **path** (*either a string or sequence of strings*) – update-able file(s)
- **permissions** (*octal*) – *nix style permission number
- **recursive** (*boolean, default is False*) – change just listed entry(ies) or all in directories

Returns list of all changed items

chown (*path*, *owner*, *group*, *recursive=False*)

Use snakebite.chown/chgrp, if available.

One of owner or group must be set. Just setting group calls chgrp.

Parameters

- **path** (*either a string or sequence of strings*) – update-able file(s)
- **owner** (*string*) – new owner, can be blank
- **group** (*string*) – new group, can be blank
- **recursive** (*boolean, default is False*) – change just listed entry(ies) or all in directories

Returns list of all changed items

count (*path*)

Use snakebite.count, if available.

Parameters **path** (*string*) – directory to count the contents of

Returns dictionary with content_size, dir_count and file_count keys

copy (*path*, *destination*)

Raise a NotImplementedError exception.

put (*local_path*, *destination*)

Raise a NotImplementedError exception.

get (*path*, *local_destination*)

Use snakebite.copyToLocal, if available.

Parameters

- **path** (*string*) – HDFS file
- **local_destination** (*string*) – path on the system running Luigi

mkdir (*path*, *parents=True*, *mode=493*, *raise_if_exists=False*)

Use snakebite.mkdir, if available.

Snakebite's mkdir method allows control over full path creation, so by default, tell it to build a full path to work like `hadoop fs -mkdir`.

Parameters

- **path** (*string*) – HDFS path to create
- **parents** (*boolean, default is True*) – create any missing parent directories
- **mode** (*octal, default 0755*) – *nix style owner/group/other permissions

listdir (*path*, *ignore_directories=False*, *ignore_files=False*, *include_size=False*, *include_type=False*, *include_time=False*, *recursive=False*)

Use snakebite.ls to get the list of items in a directory.

Parameters

- **path** (*string*) – the directory to list
- **ignore_directories** (*boolean, default is False*) – if True, do not yield directory entries
- **ignore_files** (*boolean, default is False*) – if True, do not yield file entries
- **include_size** (*boolean, default is False (do not include)*) – include the size in bytes of the current item
- **include_type** (*boolean, default is False (do not include)*) – include the type (d or f) of the current item
- **include_time** (*boolean, default is False (do not include)*) – include the last modification time of the current item
- **recursive** (*boolean, default is False (do not recurse)*) – list subdirectory contents

Returns yield with a string, or if any of the include_* settings are true, a tuple starting with the path, and include_* items in order

touchz (*path*)

Raise a NotImplementedError exception.

luigi.contrib.hdfs.target module Provides access to HDFS using the `HdfsTarget`, a subclass of `Target`.

class luigi.contrib.hdfs.target.**HdfsTarget** (*path=None*, *format=None*, *is_tmp=False*, *fs=None*)

Bases: `luigi.target.FileSystemTarget`

fs

glob_exists (*expected_files*)

open (*mode='r'*)

remove (*skip_trash=False*)

rename (*path*, *raise_if_exists=False*)

Rename does not change self.path, so be careful with assumptions.

Not recommendeed for directories. Use move_dir. spotify/luigi#522

move (*path*, *raise_if_exists=False*)
Move does not change self.path, so be careful with assumptions.

Not recommendeed for directories. Use move_dir. spotify/luigi#522

move_dir (*path*)
Rename a directory.

The implementation uses *rename_dont_move*, which on some clients is just a normal *mv* operation, which can cause nested directories.

One could argue that the implementation should use the mkdir+raise_if_exists approach, but we at Spotify have had more trouble with that over just using plain mv. See spotify/luigi#557

is_writable ()
Currently only works with hadoopcli

Module contents

Provides access to HDFS using the `HdfsTarget`, a subclass of *`Target`*. You can configure what client by setting the "client" config under the "hdfs" section in the configuration, or using the `--hdfs-client` command line option. "hadoopcli" is the slowest, but should work out of the box. "snakebite" is the fastest, but requires Snakebite to be installed.

Currently (4th May) the *`luigi.contrib.hdfs`* module is under reorganization. We recommend importing the reexports from *`luigi.contrib.hdfs`* instead of the sub-modules, as we're not yet sure how the final structure of the sub-modules will be. Eventually this module will be empty and you'll have to import directly from the sub modules like *`luigi.contrib.hdfs.config`*.

11.2.2 Submodules

luigi.contrib.bigquery module

class `luigi.contrib.bigquery.`**`CreateDisposition`**
Bases: `object`

CREATE_IF_NEEDED = 'CREATE_IF_NEEDED'

CREATE_NEVER = 'CREATE_NEVER'

class `luigi.contrib.bigquery.`**`WriteDisposition`**
Bases: `object`

WRITE_TRUNCATE = 'WRITE_TRUNCATE'

WRITE_APPEND = 'WRITE_APPEND'

WRITE_EMPTY = 'WRITE_EMPTY'

class `luigi.contrib.bigquery.`**`QueryMode`**
Bases: `object`

INTERACTIVE = 'INTERACTIVE'

BATCH = 'BATCH'

class `luigi.contrib.bigquery.`**`SourceFormat`**
Bases: `object`

CSV = 'CSV'

DATASTORE_BACKUP = 'DATASTORE_BACKUP'

NEWLINE_DELIMITED_JSON = 'NEWLINE_DELIMITED_JSON'

class `luigi.contrib.bigquery.`**`BQDataset`**(*project_id*, *dataset_id*)

Bases: `tuple`

`dataset_id`

Alias for field number 1

`project_id`

Alias for field number 0

class `luigi.contrib.bigquery.`**`BQTable`**

Bases: *`luigi.contrib.bigquery.BQTable`*

`dataset`

class `luigi.contrib.bigquery.`**`BigqueryClient`**(*oauth_credentials=None*, *descriptor=''*, *http_=None*)

Bases: `object`

A client for Google BigQuery.

For details of how authentication and the descriptor work, see the documentation for the GCS client. The descriptor URL for BigQuery is https://www.googleapis.com/discovery/v1/apis/bigquery/v2/rest

`dataset_exists`(*dataset*)

Returns whether the given dataset exists.

Parameters **`dataset`** (BQDataset) –

`table_exists`(*table*)

Returns whether the given table exists.

Parameters **`table`** (BQTable) –

`make_dataset`(*dataset*, *raise_if_exists=False*, *body={}*)

Creates a new dataset with the default permissions.

Parameters

- **`dataset`** (BQDataset) –
- **`raise_if_exists`** – whether to raise an exception if the dataset already exists.

Raises ***luigi.target.FileAlreadyExists*** if raise_if_exists=True and the dataset exists

`delete_dataset`(*dataset*, *delete_nonempty=True*)

Deletes a dataset (and optionally any tables in it), if it exists.

Parameters

- **`dataset`** (BQDataset) –
- **`delete_nonempty`** – if true, will delete any tables before deleting the dataset

`delete_table`(*table*)

Deletes a table, if it exists.

Parameters **`table`** (BQTable) –

`list_datasets`(*project_id*)

Returns the list of datasets in a given project.

Parameters **`project_id`** (*str*) –

list_tables (*dataset*)

Returns the list of tables in a given dataset.

Parameters **dataset** (BQDataset) –

run_job (*project_id*, *body*, *dataset=None*)

Runs a bigquery "job". See the documentation for the format of body.

Note: You probably don't need to use this directly. Use the tasks defined below.

Parameters **dataset** (BQDataset) –

copy (*source_table*, *dest_table*, *create_disposition='CREATE_IF_NEEDED'*, *write_disposition='WRITE_TRUNCATE'*)

Copies (or appends) a table to another table.

Parameters

- **source_table** (BQTable) –
- **dest_table** (BQTable) –
- **create_disposition** (CreateDisposition) – whether to create the table if needed
- **write_disposition** (WriteDisposition) – whether to append/truncate/fail if the table exists

class luigi.contrib.bigquery.**BigqueryTarget** (*project_id*, *dataset_id*, *table_id*, *client=None*)

Bases: *luigi.target.Target*

classmethod from_bqtable (*table*, *client=None*)

A constructor that takes a *BQTable*.

Parameters **table** (BQTable) –

exists ()

class luigi.contrib.bigquery.**BigqueryLoadTask** (**args*, ***kwargs*)

Bases: *luigi.task.Task*

Load data into bigquery from GCS.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

source_format

The source format to use (see *SourceFormat*).

write_disposition

What to do if the table already exists. By default this will fail the job.

See *WriteDisposition*

schema

Schema in the format defined at https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load.schema.

If the value is falsy, it is omitted and inferred by bigquery, which only works for CSV inputs.

max_bad_records

source_uris
Source data which should be in GCS.

run()

task_namespace = None

class luigi.contrib.bigquery.**BigqueryRunQueryTask**(**args*, ***kwargs*)
Bases: *luigi.task.Task*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

write_disposition
What to do if the table already exists. By default this will fail the job.

See *WriteDisposition*

create_disposition
Whether to create the table or not. See *CreateDisposition*

query
The query, in text form.

query_mode
The query mode. See *QueryMode*.

run()

task_namespace = None

luigi.contrib.esindex module

Support for Elasticsearch (1.0.0 or newer).

Provides an *ElasticsearchTarget* and a *CopyToIndex* template task.

Modeled after *luigi.contrib.rdbms.CopyToTable*.

A minimal example (assuming elasticsearch is running on localhost:9200):

```
class ExampleIndex(CopyToIndex):
    index = 'example'

    def docs(self):
        return [{'_id': 1, 'title': 'An example document.'}]

if __name__ == '__main__':
    task = ExampleIndex()
    luigi.build([task], local_scheduler=True)
```

All options:

```
class ExampleIndex(CopyToIndex):
    host = 'localhost'
    port = 9200
    index = 'example'
    doc_type = 'default'
    purge_existing_index = True
    marker_index_hist_size = 1

    def docs(self):
        return [{'_id': 1, 'title': 'An example document.'}]

if __name__ == '__main__':
    task = ExampleIndex()
    luigi.build([task], local_scheduler=True)
```

Host, *port*, *index*, *doc_type* parameters are standard elasticsearch.

purge_existing_index will delete the index, whenever an update is required. This is useful, when one deals with "dumps" that represent the whole data, not just updates.

marker_index_hist_size sets the maximum number of entries in the 'marker' index:

- 0 (default) keeps all updates,
- 1 to only remember the most recent update to the index.

This can be useful, if an index needs to recreated, even though the corresponding indexing task has been run sometime in the past - but a later indexing task might have altered the index in the meantime.

There are a two luigi *client.cfg* configuration options:

```
[elasticsearch]

marker-index = update_log
marker-doc-type = entry
```

class `luigi.contrib.esindex.`**`ElasticsearchTarget`**(*host*, *port*, *index*, *doc_type*, *update_id*, *marker_index_hist_size=0*, *http_auth=None*, *timeout=10*, *extra_elasticsearch_args={}*)

Bases: *`luigi.target.Target`*

Target for a resource in Elasticsearch.

Parameters

- **`host`** (*str*) – Elasticsearch server host
- **`port`** (*int*) – Elasticsearch server port
- **`index`** (*str*) – index name
- **`doc_type`** (*str*) – doctype name
- **`update_id`** (*str*) – an identifier for this data set
- **`marker_index_hist_size`** (*int*) – list of changes to the index to remember
- **`timeout`** (*int*) – Elasticsearch connection timeout
- **`extra_elasticsearch_args`** – extra args for Elasticsearch

`marker_index` = **'update_log'**

`marker_doc_type` = **'entry'**

`marker_index_document_id`()

Generate an id for the indicator document.

touch()

Mark this update as complete.

The document id would be sufficent but, for documentation, we index the parameters *update_id*, *target_index*, *target_doc_type* and *date* as well.

exists()

Test, if this task has been run.

create_marker_index()

Create the index that will keep track of the tasks if necessary.

ensure_hist_size()

Shrink the history of updates for a *index/doc_type* combination down to *self.marker_index_hist_size*.

class luigi.contrib.esindex.**CopyToIndex**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Template task for inserting a data set into Elasticsearch.

Usage:

1. Subclass and override the required *index* attribute.
2. Implement a custom *docs* method, that returns an iterable over the documents. A document can be a JSON string, e.g. from a newline-delimited JSON (ldj) file (default implementation) or some dictionary.

Optional attributes:

- doc_type (default),
- host (localhost),
- port (9200),
- settings ({'settings': {}})
- mapping (None),
- chunk_size (2000),
- raise_on_error (True),
- purge_existing_index (False),
- marker_index_hist_size (0)

If settings are defined, they are only applied at index creation time.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

host

ES hostname.

port

ES port.

http_auth

ES optional http auth information as either ':' separated string or a tuple, e.g. *('user', 'pass')* or *"user:pass"*.

index
The target index.

May exist or not.

doc_type
The target doc_type.

mapping
Dictionary with custom mapping or *None*.

settings
Settings to be used at index creation time.

chunk_size
Single API call for this number of docs.

raise_on_error
Whether to fail fast.

purge_existing_index
Whether to delete the *index* completely before any indexing.

marker_index_hist_size
Number of event log entries in the marker index. 0: unlimited.

timeout
Timeout.

extra_elasticsearch_args
Extra arguments to pass to the Elasticsearch constructor

docs()
Return the documents to be indexed.

Beside the user defined fields, the document may contain an *_index*, *_type* and *_id*.

create_index()
Override to provide code for creating the target index.

By default it will be created without any special settings or mappings.

delete_index()
Delete the index, if it exists.

update_id()
This id will be a unique identifier for this indexing task.

output()
Returns a ElasticsearchTarget representing the inserted dataset.

Normally you don't override this.

run()
Run task, namely:

- purge existing index, if requested (*purge_existing_index*),
- create the index, if missing,
- apply mappings, if given,
- set refresh interval to -1 (disable) for performance reasons,
- bulk index in batches of size *chunk_size* (2000),

- set refresh interval to 1s,
- refresh Elasticsearch,
- create entry in marker index.

task_namespace = **None**

luigi.contrib.ftp module

This library is a wrapper of ftplib. It is convenient to move data from/to FTP.

There is an example on how to use it (example/ftp_experiment_outputs.py)

You can also find unittest for each class.

Be aware that normal ftp do not provide secure communication.

class `luigi.contrib.ftp.`**`RemoteFileSystem`**(*host*, *username=None*, *password=None*, *port=21*, *tls=False*)

Bases: *`luigi.target.FileSystem`*

`exists`(*path*, *mtime=None*)

Return *True* if file or directory at *path* exist, False otherwise.

Additional check on modified time when mtime is passed in.

Return False if the file's modified time is older mtime.

`remove`(*path*, *recursive=True*)

Remove file or directory at location `path`.

Parameters

- **`path`** (*str*) – a path within the FileSystem to remove.
- **`recursive`** (*bool*) – if the path is a directory, recursively remove the directory and all of its descendants. Defaults to `True`.

`put`(*local_path*, *path*)

`get`(*path*, *local_path*)

class `luigi.contrib.ftp.`**`AtomicFtpFile`**(*fs*, *path*)

Bases: *`luigi.target.AtomicLocalFile`*

Simple class that writes to a temp file and upload to ftp on close().

Also cleans up the temp file if close is not invoked.

Initializes an AtomicFtpfile instance. :param fs: :param path: :type path: str

`move_to_final_destination`()

`fs`

class `luigi.contrib.ftp.`**`RemoteTarget`**(*path*, *host*, *format=None*, *username=None*, *password=None*, *port=21*, *mtime=None*, *tls=False*)

Bases: *`luigi.target.FileSystemTarget`*

Target used for reading from remote files.

The target is implemented using ssh commands streaming data over the network.

`fs`

open (*mode*)

Open the FileSystem target.

This method returns a file-like object which can either be read from or written to depending on the specified mode.

Parameters mode (*str*) – the mode *r* opens the FileSystemTarget in read-only mode, whereas *w* will open the FileSystemTarget in write mode. Subclasses can implement additional options.

exists ()

put (*local_path*)

get (*local_path*)

luigi.contrib.gcs module

luigi bindings for Google Cloud Storage

exception luigi.contrib.gcs.**InvalidDeleteException**

Bases: *luigi.target.FileSystemException*

class luigi.contrib.gcs.**GCSClient** (*oauth_credentials=None*, *descriptor=’‘*, *http_=None*)

Bases: *luigi.target.FileSystem*

An implementation of a FileSystem over Google Cloud Storage.

There are several ways to use this class. By default it will use the app default credentials, as described at https://developers.google.com/identity/protocols/application-default-credentials . Alternatively, you may pass an oauth2client credentials object. e.g. to use a service account:

```
credentials = oauth2client.client.SignedJwtAssertionCredentials(
    '012345678912-ThisIsARandomServiceAccountEmail@developer.gserviceaccount.com',
    'These are the contents of the p12 file that came with the service account',
    scope='https://www.googleapis.com/auth/devstorage.read_write')
client = GCSClient(oauth_credentails=credentails)
```

Warning: By default this class will use “automated service discovery” which will require a connection to the web. The google api client downloads a JSON file to “create” the library interface on the fly. If you want a more hermetic build, you can pass the contents of this file (currently found at https://www.googleapis.com/discovery/v1/apis/storage/v1/rest) as the `descriptor` argument.

exists (*path*)

isdir (*path*)

remove (*path*, *recursive=True*)

put (*filename*, *dest_path*, *mimetype=None*)

put_string (*contents*, *dest_path*, *mimetype=None*)

mkdir (*path*, *parents=True*, *raise_if_exists=False*)

copy (*source_path*, *destination_path*)

rename (*source_path*, *destination_path*)

Rename/move an object from one S3 location to another.

listdir (*path*)

Get an iterable with S3 folder contents. Iterable contains paths relative to queried path.

download (*path*)

class `luigi.contrib.gcs.`**`AtomicGCSFile`**(*path*, *gcs_client*)

Bases: *`luigi.target.AtomicLocalFile`*

A GCS file that writes to a temp file and put to GCS on close.

`move_to_final_destination`()

class `luigi.contrib.gcs.`**`GCSTarget`**(*path*, *format=None*, *client=None*)

Bases: *`luigi.target.FileSystemTarget`*

`fs` = **None**

`open`(*mode='r'*)

class `luigi.contrib.gcs.`**`GCSFlagTarget`**(*path*, *format=None*, *client=None*, *flag='_SUCCESS'*)

Bases: *`luigi.contrib.gcs.GCSTarget`*

Defines a target directory with a flag-file (defaults to *_SUCCESS*) used to signify job success.

This checks for two things:

- the path exists (just like the GCSTarget)
- the _SUCCESS file exists within the directory.

Because Hadoop outputs into a directory and not a single file, the path is assumed to be a directory.

This is meant to be a handy alternative to AtomicGCSFile.

The AtomicFile approach can be burdensome for GCS since there are no directories, per se.

If we have 1,000,000 output files, then we have to rename 1,000,000 objects.

Initializes a S3FlagTarget.

Parameters

- **`path`** (*str*) – the directory where the files are stored.
- **`client`** –
- **`flag`** (*str*) –

`fs` = **None**

`exists`()

luigi.contrib.hadoop module

Run Hadoop Mapreduce jobs using Hadoop Streaming. To run a job, you need to subclass *`luigi.contrib.hadoop.JobTask`* and implement a `mapper` and `reducer` methods. See Example – Top Artists for an example of how to run a Hadoop job.

class `luigi.contrib.hadoop.`**`BaseHadoopJobTask`**(**args*, ***kwargs*)

Bases: *`luigi.task.Task`*

Constructor to resolve values for all Parameters.

For example, the Task:

```python
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`batch_counter_default` = **1**

deps()

final_combiner = NotImplemented

final_mapper = NotImplemented

final_reducer = NotImplemented

init_hadoop()

init_local()

Implement any work to setup any internal datastructure etc here.

You can add extra input using the requires_local/input_local methods.

Anything you set on the object will be pickled and available on the Hadoop nodes.

input_hadoop()

input_local()

job_runner()

jobconfs()

mr_priority = NotImplemented

on_failure(*exception*)

pool = <luigi.parameter.Parameter object>

requires_hadoop()

requires_local()

Default impl - override this method if you need any local input to be accessible in init().

run()

task_id = None

task_namespace = None

class luigi.contrib.hadoop.**DefaultHadoopJobRunner**

Bases: *luigi.contrib.hadoop.HadoopJobRunner*

The default job runner just reads from config and sets stuff.

exception luigi.contrib.hadoop.**HadoopJobError**(*message*, *out=None*, *err=None*)

Bases: exceptions.RuntimeError

class luigi.contrib.hadoop.**HadoopJobRunner**(*streaming_jar*, *modules=None*, *streaming_args=None*, *libjars=None*, *libjars_in_hdfs=None*, *jobconfs=None*, *input_format=None*, *output_format=None*, *end_job_with_atomic_move_dir=True*)

Bases: *luigi.contrib.hadoop.JobRunner*

Takes care of uploading & executing a Hadoop job using Hadoop streaming.

TODO: add code to support Elastic Mapreduce (using boto) and local execution.

finish()

run_job(*job*)

class luigi.contrib.hadoop.**HadoopRunContext**

Bases: object

kill_job(*captured_signal=None*, *stack_frame=None*)

class `luigi.contrib.hadoop.`**`JobRunner`**

Bases: `object`

`run_job` **= NotImplemented**

class `luigi.contrib.hadoop.`**`JobTask`**(**args*, ***kwargs*)

Bases: *`luigi.contrib.hadoop.BaseHadoopJobTask`*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`add_link`(*src*, *dst*)

`combiner` **= NotImplemented**

`data_interchange_format` **= ‘python’**

`deserialize`

`dump`(*directory=’‘*)

Dump instance to file.

`extra_files`()

Can be overriden in subclass.

Each element is either a string, or a pair of two strings (src, dst).

- *src* can be a directory (in which case everything will be copied recursively).
- *dst* can include subdirectories (foo/bar/baz.txt etc)

Uses Hadoop’s -files option so that the same file is reused across tasks.

`extra_modules`()

`incr_counter`(**args*, ***kwargs*)

Increments a Hadoop counter.

Since counters can be a bit slow to update, this batches the updates.

`init_combiner`()

`init_mapper`()

`init_reducer`()

`internal_reader`(*input_stream*)

Reader which uses python eval on each part of a tab separated string. Yields a tuple of python objects.

`internal_serialize`

`internal_writer`(*outputs*, *stdout*)

Writer which outputs the python repr for each item.

`job_runner`()

Get the MapReduce runner for this job.

If all outputs are HdfsTargets, the DefaultHadoopJobRunner will be used. Otherwise, the LocalJobRunner which streams all data through the local machine will be used (great for testing).

`jobconfs`()

mapper (*item*)
Re-define to process an input item (usually a line of input data).

Defaults to identity mapper that sends all lines to the same reducer.

n_reduce_tasks = 25

reader (*input_stream*)
Reader is a method which iterates over input lines and outputs records.

The default implementation yields one argument containing the line for each line in the input.

reducer = NotImplemented

run_combiner (*stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>*)

run_mapper (*stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>*)
Run the mapper on the hadoop node.

run_reducer (*stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>*)
Run the reducer on the hadoop node.

serialize

task_namespace = None

writer (*outputs, stdout, stderr=<open file '<stderr>', mode 'w'>*)
Writer format is a method which iterates over the output records from the reducer and formats them for output.

The default implementation outputs tab separated items.

class luigi.contrib.hadoop.**LocalJobRunner** (*samplelines=None*)
Bases: *luigi.contrib.hadoop.JobRunner*

Will run the job locally.

This is useful for debugging and also unit testing. Tries to mimic Hadoop Streaming.

TODO: integrate with JobTask

group (*input_stream*)

run_job (*job*)

sample (*input_stream, n, output*)

luigi.contrib.hadoop.**attach** (**packages*)
Attach a python package to hadoop map reduce tarballs to make those packages available on the hadoop cluster.

luigi.contrib.hadoop.**create_packages_archive** (*packages, filename*)
Create a tar archive which will contain the files for the packages listed in packages.

luigi.contrib.hadoop.**dereference** (*f*)

luigi.contrib.hadoop.**fetch_task_failures** (*tracking_url*)
Uses mechanize to fetch the actual task logs from the task tracker.

This is highly opportunistic, and we might not succeed. So we set a low timeout and hope it works. If it does not, it's not the end of the world.

TODO: Yarn has a REST API that we should probably use instead: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html

luigi.contrib.hadoop.**flatten** (*sequence*)
A simple generator which flattens a sequence.

Only one level is flattened.

```
(1, (2, 3), 4) -> (1, 2, 3, 4)
```

luigi.contrib.hadoop.**get_extra_files**(*extra_files*)

class luigi.contrib.hadoop.**hadoop**(**args*, ***kwargs*)

Bases: *luigi.task.Config*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

pool = <luigi.parameter.Parameter object>

task_namespace = None

luigi.contrib.hadoop.**run_and_track_hadoop_job**(*arglist*, *tracking_url_callback=None*, *env=None*)

Runs the job by invoking the command from the given arglist. Finds tracking urls from the output and attempts to fetch errors using those urls if the job fails. Throws HadoopJobError with information about the error (including stdout and stderr from the process) on failure and returns normally otherwise.

Parameters

- **arglist** –
- **tracking_url_callback** –
- **env** –

Returns

luigi.contrib.hadoop_jar module

Provides functionality to run a Hadoop job using a Jar

luigi.contrib.hadoop_jar.**fix_paths**(*job*)

Coerce input arguments to use temporary files when used for output.

Return a list of temporary file pairs (tmpfile, destination path) and a list of arguments.

Converts each HdfsTarget to a string for the path.

exception luigi.contrib.hadoop_jar.**HadoopJarJobError**

Bases: exceptions.Exception

class luigi.contrib.hadoop_jar.**HadoopJarJobRunner**

Bases: *luigi.contrib.hadoop.JobRunner*

JobRunner for *hadoop jar* commands. Used to run a HadoopJarJobTask.

run_job(*job*)

class luigi.contrib.hadoop_jar.**HadoopJarJobTask**(**args*, ***kwargs*)

Bases: *luigi.contrib.hadoop.BaseHadoopJobTask*

A job task for *hadoop jar* commands that define a jar and (optional) main method.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

jar()
Path to the jar for this Hadoop Job.

main()
optional main method for this Hadoop Job.

job_runner()

atomic_output()
If True, then rewrite output arguments to be temp locations and atomically move them into place after the job finishes.

ssh()
Set this to run hadoop command remotely via ssh. It needs to be a dict that looks like {"host": "myhost", "key_file": None, "username": None, ["no_host_key_check": False]}

args()
Returns an array of args to pass to the job (after hadoop jar <jar> <main>).

task_namespace = None

luigi.contrib.hive module

exception luigi.contrib.hive.**HiveCommandError**(*message*, *out=None*, *err=None*)
Bases: exceptions.RuntimeError

luigi.contrib.hive.**load_hive_cmd**()

luigi.contrib.hive.**get_hive_syntax**()

luigi.contrib.hive.**run_hive**(*args*, *check_return_code=True*)
Runs the *hive* from the command line, passing in the given args, and returning stdout.

With the apache release of Hive, so of the table existence checks (which are done using DESCRIBE do not exit with a return code of 0 so we need an option to ignore the return code and just return stdout for parsing

luigi.contrib.hive.**run_hive_cmd**(*hivecmd*, *check_return_code=True*)
Runs the given hive query and returns stdout.

luigi.contrib.hive.**run_hive_script**(*script*)
Runs the contents of the given script in hive and returns stdout.

class luigi.contrib.hive.**HiveClient**
Bases: object

table_location(*table*, *database='default'*, *partition=None*)
Returns location of db.table (or db.table.partition). partition is a dict of partition key to value.

table_schema(*table*, *database='default'*)
Returns list of [(name, type)] for each column in database.table.

table_exists(*table*, *database='default'*, *partition=None*)
Returns true if db.table (or db.table.partition) exists. partition is a dict of partition key to value.

partition_spec(*partition*)
Turn a dict into a string partition specification

class `luigi.contrib.hive.`**`HiveCommandClient`**

Bases: *`luigi.contrib.hive.HiveClient`*

Uses *hive* invocations to find information.

`table_location`(*table*, *database='default'*, *partition=None*)

`table_exists`(*table*, *database='default'*, *partition=None*)

`table_schema`(*table*, *database='default'*)

`partition_spec`(*partition*)

Turns a dict into the a Hive partition specification string.

class `luigi.contrib.hive.`**`ApacheHiveCommandClient`**

Bases: *`luigi.contrib.hive.HiveCommandClient`*

A subclass for the HiveCommandClient to (in some cases) ignore the return code from the hive command so that we can just parse the output.

`table_schema`(*table*, *database='default'*)

class `luigi.contrib.hive.`**`MetastoreClient`**

Bases: *`luigi.contrib.hive.HiveClient`*

`table_location`(*table*, *database='default'*, *partition=None*)

`table_exists`(*table*, *database='default'*, *partition=None*)

`table_schema`(*table*, *database='default'*)

`partition_spec`(*partition*)

class `luigi.contrib.hive.`**`HiveThriftContext`**

Bases: `object`

Context manager for hive metastore client.

`luigi.contrib.hive.`**`get_default_client`**()

class `luigi.contrib.hive.`**`HiveQueryTask`**(**args*, ***kwargs*)

Bases: *`luigi.contrib.hadoop.BaseHadoopJobTask`*

Task to run a hive query.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`n_reduce_tasks` = **None**

`bytes_per_reducer` = **None**

`reducers_max` = **None**

`query`()

Text of query to run in hive

`hiverc`()

Location of an rc file to run before the query if hiverc-location key is specified in client.cfg, will default to the value there otherwise returns None.

Returning a list of rc files will load all of them in order.

hiveconfs()

Returns an dict of key=value settings to be passed along to the hive command line via –hiveconf. By default, sets mapred.job.name to task_id and if not None, sets:

- mapred.reduce.tasks (n_reduce_tasks)
- mapred.fairscheduler.pool (pool) or mapred.job.queue.name (pool)
- hive.exec.reducers.bytes.per.reducer (bytes_per_reducer)
- hive.exec.reducers.max (reducers_max)

job_runner()

task_namespace = None

class luigi.contrib.hive.**HiveQueryRunner**

Bases: *luigi.contrib.hadoop.JobRunner*

Runs a HiveQueryTask by shelling out to hive.

prepare_outputs(*job*)

Called before job is started.

If output is a *FileSystemTarget*, create parent directories so the hive command won't fail

run_job(*job*)

class luigi.contrib.hive.**HiveTableTarget**(*table*, *database='default'*, *client=None*)

Bases: *luigi.target.Target*

exists returns true if the table exists.

exists()

path

Returns the path to this table in HDFS.

open(*mode*)

class luigi.contrib.hive.**HivePartitionTarget**(*table*, *partition*, *database='default'*, *fail_missing_table=True*, *client=None*)

Bases: *luigi.target.Target*

exists returns true if the table's partition exists.

exists()

path

Returns the path for this HiveTablePartitionTarget's data.

open(*mode*)

class luigi.contrib.hive.**ExternalHiveTask**(**args*, ***kwargs*)

Bases: *luigi.task.ExternalTask*

External task that depends on a Hive table/partition.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

database = <luigi.parameter.Parameter object>

table = <luigi.parameter.Parameter object>

partition = <luigi.parameter.Parameter object>

task_namespace = None

output ()

luigi.contrib.mysqldb module

class luigi.contrib.mysqldb.**MySqlTarget** (*host*, *database*, *user*, *password*, *table*, *update_id*)

Bases: *luigi.target.Target*

Target for a resource in MySql.

Initializes a MySqlTarget instance.

Parameters

- **host** (*str*) – MySql server address. Possibly a host:port string.
- **database** (*str*) – database name.
- **user** (*str*) – database user
- **password** (*str*) – password for specified user.
- **update_id** (*str*) – an identifier for this data set.

marker_table = 'table_updates'

touch (*connection=None*)

Mark this update as complete.

IMPORTANT, If the marker table doesn't exist, the connection transaction will be aborted and the connection reset. Then the marker table will be created.

exists (*connection=None*)

connect (*autocommit=False*)

create_marker_table ()

Create marker table if it doesn't exist.

Using a separate connection since the transaction might have to be reset.

luigi.contrib.pig module

Apache Pig support. Example configuration section in client.cfg:

```
[pig]
# pig home directory
home: /usr/share/pig
```

class luigi.contrib.pig.**PigJobTask** (**args*, ***kwargs*)

Bases: *luigi.task.Task*

Constructor to resolve values for all Parameters.

For example, the Task:

```
    class MyTask(luigi.Task):
        count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

pig_home()

pig_command_path()

pig_env_vars()

Dictionary of environment variables that should be set when running Pig.

Ex:: return { 'PIG_CLASSPATH': '/your/path' }

pig_properties()

Dictionary of properties that should be set when running Pig.

Example:

```
return { 'pig.additional.jars':'/path/to/your/jar' }
```

pig_parameters()

Dictionary of parameters that should be set for the Pig job.

Example:

```
return { 'YOUR_PARAM_NAME':'Your param value' }
```

pig_options()

List of options that will be appended to the Pig command.

Example:

```
return ['-x', 'local']
```

output()

pig_script_path()

Return the path to the Pig script to be run.

run()

track_and_progress(*cmd*)

task_namespace = None

class luigi.contrib.pig.**PigRunContext**

Bases: `object`

kill_job(*captured_signal=None*, *stack_frame=None*)

exception luigi.contrib.pig.**PigJobError**(*message*, *out=None*, *err=None*)

Bases: `exceptions.RuntimeError`

luigi.contrib.pyspark_runner module

The pyspark program.

This module will be run by spark-submit for PySparkTask jobs.

The first argument is a path to the pickled instance of the PySparkTask, other arguments are the ones returned by PySparkTask.app_options()

class luigi.contrib.pyspark_runner.**PySparkRunner**(*job*, **args*)

Bases: `object`

run()

luigi.contrib.rdbms module

A common module for posgres like databases, such as postgres or redshift

class `luigi.contrib.rdbms.`**`CopyToTable`**(**args*, ***kwargs*)

Bases: *`luigi.task.MixinNaiveBulkComplete`*, *`luigi.task.Task`*

An abstract task for inserting a data set into RDBMS.

Usage:

Subclass and override the following attributes:

- *host*,
- *database*,
- *user*,
- *password*,
- *table*
- *columns*

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`host`

`database`

`user`

`password`

`table`

`columns` = []

`null_values` = (None,)

`column_separator` = '\t'

`create_table`(*connection*)

Override to provide code for creating the target table.

By default it will be created using types (optionally) specified in columns.

If overridden, use the provided connection object for setting up the table in order to create the table and insert data using the same transaction.

`update_id`()

This update id will be a unique identifier for this insert on this table.

`output`()

`init_copy`(*connection*)

Override to perform custom queries.

Any code here will be formed in the same transaction as the main copy, just prior to copying data. Example use cases include truncating the table or removing all data older than X in the database to keep a rolling window of data available in the table.

copy (*cursor*, *file*)

task_namespace = None

luigi.contrib.redis_store module

class luigi.contrib.redis_store.**RedisTarget** (*host*, *port*, *db*, *update_id*, *password=None*, *socket_timeout=None*, *expire=None*)

Bases: *luigi.target.Target*

Target for a resource in Redis.

Parameters

- **host** (*str*) – Redis server host
- **port** (*int*) – Redis server port
- **db** (*int*) – database index
- **update_id** (*str*) – an identifier for this data hash
- **password** (*str*) – a password to connect to the redis server
- **socket_timeout** (*int*) – client socket timeout
- **expire** (*int*) – timeout before the target is deleted

marker_prefix = <luigi.parameter.Parameter object>

marker_key ()

Generate a key for the indicator hash.

touch ()

Mark this update as complete.

We index the parameters *update_id* and *date*.

exists ()

Test, if this task has been run.

luigi.contrib.redshift module

class luigi.contrib.redshift.**RedshiftTarget** (*host*, *database*, *user*, *password*, *table*, *update_id*)

Bases: *luigi.postgres.PostgresTarget*

Target for a resource in Redshift.

Redshift is similar to postgres with a few adjustments required by redshift.

Args: host (str): Postgres server address. Possibly a host:port string. database (str): Database name user (str): Database user password (str): Password for specified user update_id (str): An identifier for this data set

marker_table = 'table_updates'

use_db_timestamps = False

class `luigi.contrib.redshift.`**`S3CopyToTable`**(**args*, ***kwargs*)

Bases: `luigi.contrib.rdbms.CopyToTable`

Template task for inserting a data set into Redshift from s3.

Usage:

- Subclass and override the required attributes: * *host*, * *database*, * *user*, * *password*, * *table*, * *columns*, * *aws_access_key_id*, * *aws_secret_access_key*, * *s3_load_path*.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`s3_load_path`

Override to return the load path.

`aws_access_key_id`

Override to return the key id.

`aws_secret_access_key`

Override to return the secret access key.

`copy_options`

Add extra copy options, for example:

- TIMEFORMAT 'auto'
- IGNOREHEADER 1
- TRUNCATECOLUMNS
- IGNOREBLANKLINES

`table_attributes`()

Add extra table attributes, for example: DISTSTYLE KEY DISTKEY (MY_FIELD) SORTKEY (MY_FIELD_2, MY_FIELD_3)

`do_truncate_table`()

Return True if table should be truncated before copying new data in.

`truncate_table`(*connection*)

`create_table`(*connection*)

Override to provide code for creating the target table.

By default it will be created using types (optionally) specified in columns.

If overridden, use the provided connection object for setting up the table in order to create the table and insert data using the same transaction.

`run`()

If the target table doesn't exist, self.create_table will be called to attempt to create the table.

`copy`(*cursor*, *f*)

Defines copying from s3 into redshift.

`output`()

Returns a RedshiftTarget representing the inserted dataset.

Normally you don't override this.

does_table_exist (*connection*)

Determine whether the table already exists.

task_namespace = None

class luigi.contrib.redshift.**S3CopyJSONToTable** (**args*, ***kwargs*)

Bases: *luigi.contrib.redshift.S3CopyToTable*

Template task for inserting a JSON data set into Redshift from s3.

Usage:

- Subclass and override the required attributes:
  - *host*,
  - *database*,
  - *user*,
  - *password*,
  - *table*,
  - *columns*,
  - *aws_access_key_id*,
  - *aws_secret_access_key*,
  - *s3_load_path*,
  - *jsonpath*,
  - *copy_json_options*.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

jsonpath

Override the jsonpath schema location for the table.

copy_json_options

Add extra copy options, for example:

- GZIP
- LZOP

copy (*cursor*, *f*)

Defines copying JSON from s3 into redshift.

task_namespace = None

class luigi.contrib.redshift.**RedshiftManifestTask** (**args*, ***kwargs*)

Bases: *luigi.s3.S3PathTask*

Generic task to generate a manifest file that can be used in S3CopyToTable in order to copy multiple files from your s3 folder into a redshift table at once.

For full description on how to use the manifest file see http://docs.aws.amazon.com/redshift/latest/dg/loading-data-files-using-manifest.html

Usage:

- **requires parameters**
  - **path - s3 path to the generated manifest file, including the** name of the generated file to be copied into a redshift table
  - folder_paths - s3 paths to the folders containing files you wish to be copied

Output:

- generated manifest file

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`folder_paths` = **<luigi.parameter.Parameter object>**

`text_target` = **True**

`run`()

`task_namespace` = **None**

class `luigi.contrib.redshift.`**`KillOpenRedshiftSessions`**(**args*, ***kwargs*)

Bases: *`luigi.task.Task`*

An task for killing any open Redshift sessions in a given database. This is necessary to prevent open user sessions with transactions against the table from blocking drop or truncate table commands.

Usage:

Subclass and override the required *host*, *database*, *user*, and *password* attributes.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`connection_reset_wait_seconds` = **<luigi.parameter.IntParameter object>**

`host`

`database`

`user`

`password`

`update_id`()

This update id will be a unique identifier for this insert on this table.

`output`()

Returns a RedshiftTarget representing the inserted dataset.

Normally you don’t override this.

run()
Kill any open Redshift sessions for the given database.

task_namespace = **None**

luigi.contrib.scalding module

luigi.contrib.scalding.**logger** = **<logging.Logger object>**
Scalding support for Luigi.

Example configuration section in client.cfg:

```
[scalding]
# scala home directory, which should include a lib subdir with scala jars.
scala-home: /usr/share/scala

# scalding home directory, which should include a lib subdir with
# scalding-*-assembly-* jars as built from the official Twitter build script.
scalding-home: /usr/share/scalding

# provided dependencies, e.g. jars required for compiling but not executing
# scalding jobs. Currently requred jars:
# org.apache.hadoop/hadoop-core/0.20.2
# org.slf4j/slf4j-log4j12/1.6.6
# log4j/log4j/1.2.15
# commons-httpclient/commons-httpclient/3.1
# commons-cli/commons-cli/1.2
# org.apache.zookeeper/zookeeper/3.3.4
scalding-provided: /usr/share/scalding/provided

# additional jars required.
scalding-libjars: /usr/share/scalding/libjars
```

class luigi.contrib.scalding.**ScaldingJobRunner**
Bases: *luigi.contrib.hadoop.JobRunner*

JobRunner for *pyscald* commands. Used to run a ScaldingJobTask.

get_scala_jars(*include_compiler=False*)

get_scalding_jars()

get_scalding_core()

get_provided_jars()

get_libjars()

get_tmp_job_jar(*source*)

get_build_dir(*source*)

get_job_class(*source*)

build_job_jar(*job*)

run_job(*job*)

class luigi.contrib.scalding.**ScaldingJobTask**(**args*, ***kwargs*)
Bases: *luigi.contrib.hadoop.BaseHadoopJobTask*

A job task for Scalding that define a scala source and (optional) main method.

requires() should return a dictionary where the keys are Scalding argument names and values are sub tasks or lists of subtasks.

For example:

```
{'input1': A, 'input2': C} => --input1 <Aoutput> --input2 <Coutput>
{'input1': [A, B], 'input2': [C]} => --input1 <Aoutput> <Boutput> --input2 <Coutput>
```

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

relpath(*current_file*, *rel_path*)
Compute path given current file and relative path.

source()
Path to the scala source for this Scalding Job

Either one of source() or jar() must be specified.

jar()
Path to the jar file for this Scalding Job

Either one of source() or jar() must be specified.

extra_jars()
Extra jars for building and running this Scalding Job.

job_class()
optional main job class for this Scalding Job.

job_runner()

atomic_output()
If True, then rewrite output arguments to be temp locations and atomically move them into place after the job finishes.

requires()

job_args()
Extra arguments to pass to the Scalding job.

task_namespace = None

args()
Returns an array of args to pass to the job.

luigi.contrib.spark module

class `luigi.contrib.spark.`**SparkRunContext**(*proc*)
Bases: `object`

kill_job(*captured_signal=None*, *stack_frame=None*)

exception `luigi.contrib.spark.`**SparkJobError**(*message*, *out=None*, *err=None*)
Bases: `exceptions.RuntimeError`

class `luigi.contrib.spark.`**`SparkSubmitTask`**(**args*, ***kwargs*)

Bases: *`luigi.task.Task`*

Template task for running a Spark job

Supports running jobs on Spark local, standalone, Mesos or Yarn

See http://spark.apache.org/docs/latest/submitting-applications.html for more information

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`name` **= None**

`entry_class` **= None**

`app` **= None**

`app_options`()

Subclass this method to map your task parameters to the app's arguments

`spark_submit`

`master`

`deploy_mode`

`jars`

`py_files`

`files`

`conf`

`properties_file`

`driver_memory`

`driver_java_options`

`driver_library_path`

`driver_class_path`

`executor_memory`

`driver_cores`

`supervise`

`total_executor_cores`

`executor_cores`

`queue`

`num_executors`

`archives`

`hadoop_conf_dir`

`get_environment`()

spark_command()

app_command()

run()

task_namespace = None

class luigi.contrib.spark.**PySparkTask**(**args*, ***kwargs*)

Bases: *luigi.contrib.spark.SparkSubmitTask*

Template task for running an inline PySpark job

Simply implement the `main` method in your subclass

You can optionally define package names to be distributed to the cluster with `py_packages` (uses luigi's global py-packages configuration by default)

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

app = '/home/docs/checkouts/readthedocs.org/user_builds/luigi/checkouts/latest/luigi/contrib/pyspark_runner.py'

deploy_mode = 'client'

name

py_packages

setup(*conf*)

Called by the pyspark_runner with a SparkConf instance that will be used to instantiate the SparkContext

Parameters conf – SparkConf

setup_remote(*sc*)

main(*sc*, **args*)

Called by the pyspark_runner with a SparkContext and any arguments returned by `app_options()`

Parameters

- **sc** – SparkContext
- **args** – arguments list

app_command()

run()

task_namespace = None

class luigi.contrib.spark.**SparkJob**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

Deprecated since version 1.1.1: Use `SparkSubmitTask` or `PySparkTask` instead.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

spark_workers = None

spark_master_memory = None

spark_worker_memory = None

queue = <luigi.parameter.Parameter object>

temp_hadoop_output_file = None

requires_local()
Default impl - override this method if you need any local input to be accessible in init().

requires_hadoop()

input_local()

input()

deps()

jar()

job_class()

job_args()

output()

run()

track_progress(*proc*)

task_namespace = None

class `luigi.contrib.spark.`**Spark1xBackwardCompat**(**args*, ***kwargs*)
Bases: *luigi.contrib.spark.SparkSubmitTask*

Adapts SparkSubmitTask interface to (Py)Spark1xJob interface

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

master

output()

spark_options()

dependency_jars()

job_args()

jars

app_options()

spark_command()

task_namespace = None

class luigi.contrib.spark.**Spark1xJob**(**args*, ***kwargs*)
Bases: *luigi.contrib.spark.Spark1xBackwardCompat*

Deprecated since version 1.1.1: Use SparkSubmitTask or PySparkTask instead.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

job_class()

jar()

entry_class

app

run()

task_namespace = None

class luigi.contrib.spark.**PySpark1xJob**(**args*, ***kwargs*)
Bases: *luigi.contrib.spark.Spark1xBackwardCompat*

Deprecated since version 1.1.1: Use SparkSubmitTask or PySparkTask instead.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

program()

app

run()

task_namespace = None

luigi.contrib.sparkey module

class luigi.contrib.sparkey.**SparkeyExportTask**(**args*, ***kwargs*)
Bases: *luigi.task.Task*

A luigi task that writes to a local sparkey log file.

Subclasses should implement the requires and output methods. The output must be a luigi.LocalTarget.

The resulting sparkey log file will contain one entry for every line in the input, mapping from the first value to a tab-separated list of the rest of the line.

To generate a simple key-value index, yield "key", "value" pairs from the input(s) to this task.

separator = '\t'

run()

task_namespace = None

luigi.contrib.sqla module

Support for SQLAlchmey. Provides SQLAlchemyTarget for storing in databases supported by SQLAlchemy. The user would be responsible for installing the required database driver to connect using SQLAlchemy.

Minimal example of a job to copy data to database using SQLAlchemy is as shown below:

```
from sqlalchemy import String
import luigi
from luigi.contrib import sqla

class SQLATask(sqla.CopyToTable):
    # columns defines the table schema, with each element corresponding
    # to a column in the format (args, kwargs) which will be sent to
    # the sqlalchemy.Column(*args, **kwargs)
    columns = [
        (["item", String(64)], {"primary_key": True}),
        (["property", String(64)], {})
    ]
    connection_string = "sqlite://"  # in memory SQLite database
    table = "item_property"  # name of the table to store data

    def rows(self):
        for row in [("item1" "property1"), ("item2", "property2")]:
            yield row

if __name__ == '__main__':
    task = SQLATask()
    luigi.build([task], local_scheduler=True)
```

If the target table where the data needs to be copied already exists, then the column schema definition can be skipped and instead the reflect flag can be set as True. Here is a modified version of the above example:

```
from sqlalchemy import String
import luigi
from luigi.contrib import sqla

class SQLATask(sqla.CopyToTable):
    # If database table is already created, then the schema can be loaded
    # by setting the reflect flag to True
    reflect = True
    connection_string = "sqlite://"  # in memory SQLite database
    table = "item_property"  # name of the table to store data

    def rows(self):
        for row in [("item1" "property1"), ("item2", "property2")]:
            yield row

if __name__ == '__main__':
    task = SQLATask()
    luigi.build([task], local_scheduler=True)
```

In the above examples, the data that needs to be copied was directly provided by overriding the rows method. Alternately, if the data comes from another task, the modified example would look as shown below:

```
from sqlalchemy import String
import luigi
from luigi.contrib import sqla
from luigi.mock import MockFile
```

```
class BaseTask(luigi.Task):
    def output(self):
        return MockFile("BaseTask")

    def run(self):
        out = self.output().open("w")
        TASK_LIST = ["item%d\tproperty%d\n" % (i, i) for i in range(10)]
        for task in TASK_LIST:
            out.write(task)
        out.close()

class SQLATask(sqla.CopyToTable):
    # columns defines the table schema, with each element corresponding
    # to a column in the format (args, kwargs) which will be sent to
    # the sqlalchemy.Column(*args, **kwargs)
    columns = [
        (["item", String(64)], {"primary_key": True}),
        (["property", String(64)], {})
    ]
    connection_string = "sqlite://"  # in memory SQLite database
    table = "item_property"  # name of the table to store data

    def requires(self):
        return BaseTask()

if __name__ == '__main__':
    task1, task2 = SQLATask(), BaseTask()
    luigi.build([task1, task2], local_scheduler=True)
```

In the above example, the output from *BaseTask* is copied into the database. Here we did not have to implement the *rows* method because by default *rows* implementation assumes every line is a row with column values separated by a tab. One can define *column_separator* option for the task if the values are say comma separated instead of tab separated.

You can pass in database specific connection arguments by setting the connect_args dictionary. The options will be passed directly to the DBAPI's connect method as keyword arguments.

The other option to *sqla.CopyToTable* that can be of help with performance aspect is the *chunk_size*. The default is 5000. This is the number of rows that will be inserted in a transaction at a time. Depending on the size of the inserts, this value can be tuned for performance.

See here for a tutorial on building task pipelines using luigi and using SQLAlchemy in workflow pipelines.

Author: Gouthaman Balaraman Date: 01/02/2015

class `luigi.contrib.sqla.`**`SQLAlchemyTarget`**(*connection_string*, *target_table*, *update_id*, *echo=False*, *connect_args=None*)

Bases: *`luigi.target.Target`*

Database target using SQLAlchemy.

This will rarely have to be directly instantiated by the user.

Typical usage would be to override *luigi.contrib.sqla.CopyToTable* class to create a task to write to the database.

Constructor for the SQLAlchemyTarget.

Parameters

- **`connection_string`** (*str*) – SQLAlchemy connection string
- **`target_table`** (*str*) – The table name for the data

- **update_id** (*str*) – An identifier for this data set
- **echo** (*bool*) – Flag to setup SQLAlchemy logging
- **connect_args** (*dict*) – A dictionary of connection arguments

Returns

marker_table = None

class Connection (*engine*, *pid*)

Bases: tuple

engine

Alias for field number 0

pid

Alias for field number 1

SQLAlchemyTarget.**connect_args = {}**

SQLAlchemyTarget.**engine**

Return an engine instance, creating it if it doesn't exist.

Recreate the engine connection if it wasn't originally created by the current process.

SQLAlchemyTarget.**touch**()

Mark this update as complete.

SQLAlchemyTarget.**exists**()

SQLAlchemyTarget.**create_marker_table**()

Create marker table if it doesn't exist.

Using a separate connection since the transaction might have to be reset.

SQLAlchemyTarget.**open**(*mode*)

class luigi.contrib.sqla.**CopyToTable**(**args*, ***kwargs*)

Bases: *luigi.task.Task*

An abstract task for inserting a data set into SQLAlchemy RDBMS

Usage:

- subclass and override the required *connection_string*, *table* and *columns* attributes.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as MyTask(count=10).

echo = False

connect_args = {}

connection_string()

table

columns = []

column_separator = '\t'

chunk_size = 5000

reflect = False

create_table (*engine*)
Override to provide code for creating the target table.

By default it will be created using types specified in columns. If the table exists, then it binds to the existing table.

If overridden, use the provided connection object for setting up the table in order to create the table and insert data using the same transaction. :param engine: The sqlalchemy engine instance :type engine: object

update_id ()
This update id will be a unique identifier for this insert on this table.

output ()

rows ()
Return/yield tuples or lists corresponding to each row to be inserted.

This method can be overridden for custom file types or formats.

run ()

copy (*conn*, *ins_rows*, *table_bound*)
This method does the actual insertion of the rows of data given by ins_rows into the database. A task that needs row updates instead of insertions should overload this method. :param conn: The sqlalchemy connection object :param ins_rows: The dictionary of rows with the keys in the format _<column_name>. For example if you have a table with a column name "property", then the key in the dictionary would be "_property". This format is consistent with the bindparam usage in sqlalchemy. :param table_bound: The object referring to the table :return:

task_namespace = None

luigi.contrib.ssh module

Light-weight remote execution library and utilities.

There are some examples in the unittest, but I added another more luigi-specific in the examples directory (examples/ssh_remote_execution.py

RemoteContext is meant to provide functionality similar to that of the standard library subprocess module, but where the commands executed are run on a remote machine instead, without the user having to think about prefixing everything with "ssh" and credentials etc.

Using this mini library (which is just a convenience wrapper for subprocess), *RemoteTarget* is created to let you stream data from a remotely stored file using the luigi *FileSystemTarget* semantics.

As a bonus, *RemoteContext* also provides a really cool feature that let's you set up ssh tunnels super easily using a python context manager (there is an example in the integration part of unittests).

This can be super convenient when you want secure communication using a non-secure protocol or circumvent firewalls (as long as they are open for ssh traffic).

class luigi.contrib.ssh.**RemoteContext** (*host*, ***kwargs*)
Bases: object

Popen (*cmd*, ***kwargs*)
Remote Popen.

check_output (*cmd*)
Execute a shell command remotely and return the output.

Simplified version of Popen when you only want the output as a string and detect any errors.

tunnel(**args*, ***kwds*)

Open a tunnel between localhost:local_port and remote_host:remote_port via the host specified by this context.

Remember to close() the returned “tunnel” object in order to clean up after yourself when you are done with the tunnel.

class `luigi.contrib.ssh.`**RemoteFileSystem**(*host*, ***kwargs*)

Bases: `luigi.target.FileSystem`

exists(*path*)

Return *True* if file or directory at *path* exist, False otherwise.

listdir(*path*)

isdir(*path*)

Return *True* if directory at *path* exist, False otherwise.

remove(*path*, *recursive=True*)

Remove file or directory at location *path*.

mkdir(*path*, *parents=True*, *raise_if_exists=False*)

put(*local_path*, *path*)

get(*path*, *local_path*)

class `luigi.contrib.ssh.`**AtomicRemoteFileWriter**(*fs*, *path*)

Bases: `luigi.format.OutputPipeProcessWrapper`

close()

tmp_path

fs

class `luigi.contrib.ssh.`**RemoteTarget**(*path*, *host*, *format=None*, ***kwargs*)

Bases: `luigi.target.FileSystemTarget`

Target used for reading from remote files.

The target is implemented using ssh commands streaming data over the network.

fs

open(*mode='r'*)

put(*local_path*)

get(*local_path*)

luigi.contrib.target module

class `luigi.contrib.target.`**CascadingClient**(*clients*, *method_names=None*)

Bases: `object`

A FilesystemClient that will cascade failing function calls through a list of clients.

Which clients are used are specified at time of construction.

ALL_METHOD_NAMES = [’exists’, ‘rename’, ‘remove’, ‘chmod’, ‘chown’, ‘count’, ‘copy’, ‘get’, ‘put’, ‘mkdir’, ‘list’, ‘listdi

luigi.contrib.webhdfs module

Provides a `WebHdfsTarget` and `WebHdfsClient` using the Python hdfs

class `luigi.contrib.webhdfs.`**`WebHdfsTarget`**(*path*, *client=None*, *format=None*)
Bases: `luigi.target.FileSystemTarget`

`fs` = None

`open`(*mode='r'*)

class `luigi.contrib.webhdfs.`**`ReadableWebHdfsFile`**(*path*, *client*)
Bases: `object`

`read`()

`readlines`(*char='\n'*)

`close`()

class `luigi.contrib.webhdfs.`**`AtomicWebHdfsFile`**(*path*, *client*)
Bases: `luigi.target.AtomicLocalFile`

An Hdfs file that writes to a temp file and put to WebHdfs on close.

`move_to_final_destination`()

class `luigi.contrib.webhdfs.`**`WebHdfsClient`**(*host=None*, *port=None*, *user=None*)
Bases: `object`

`get_config`(*key*)

`walk`(*path*, *depth=1*)

`exists`(*path*)
Returns true if the path exists and false otherwise.

`upload`(*hdfs_path*, *local_path*, *overwrite=False*)

`download`(*hdfs_path*, *local_path*, *overwrite=False*, *n_threads=-1*)

`remove`(*hdfs_path*, *recursive=False*)

`read`(*hdfs_path*, *offset=0*, *length=None*, *buffer_size=None*, *chunk_size=1024*, *buffer_char=None*)

11.2.3 Module contents

Package containing optional and-on functionality.

11.3 luigi.tools package

11.3.1 Submodules

luigi.tools.deps module

`luigi.tools.deps.`**`get_task_requires`**(*task*)

`luigi.tools.deps.`**`dfs_paths`**(*start_task*, *goal_task_family*, *path=None*)

class luigi.tools.deps.**upstream**(**args*, ***kwargs*)

Bases: luigi.task.Config

Used to provide the parameter upstream-task-family

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

family = <luigi.parameter.Parameter object>

task_namespace = None

luigi.tools.deps.**find_deps**(*task*, *upstream_task_family*)

Finds all dependencies that start with the given task and have a path to upstream_task_family

Returns all deps on all paths between task and upstream

luigi.tools.deps.**find_deps_cli**()

Finds all tasks on all paths from provided CLI task

luigi.tools.deps.**main**()

luigi.tools.luigi_grep module

class luigi.tools.luigi_grep.**LuigiGrep**(*host*, *port*)

Bases: object

graph_url

prefix_search(*job_name_prefix*)

searches for jobs matching the given job_name_prefix.

status_search(*status*)

searches for jobs matching the given status

luigi.tools.luigi_grep.**main**()

luigi.tools.parse_task module

luigi.tools.parse_task.**id_to_name_and_params**(*task_id*)

Turn a task_id into a (task_family, {params}) tuple.

E.g. calling with `Foo(bar=bar, baz=baz)` returns `('Foo', {'bar': 'bar', 'baz': 'baz'})`.

luigi.tools.range module

Produces contiguous completed ranges of recurring tasks.

See RangeDaily and RangeHourly for basic usage.

Caveat - if gaps accumulate, their causes (e.g. missing dependencies) going unmonitored/unmitigated, then this will eventually keep retrying the same gaps over and over and make no progress to more recent times. (See 'task_limit' and 'reverse' parameters.) TODO foolproof against that kind of misuse?

class `luigi.tools.range.`**`RangeEvent`**

Bases: *`luigi.event.Event`*

Events communicating useful metrics.

COMPLETE_COUNT would normally be nondecreasing, and its derivative would describe performance (how many instances complete invocation-over-invocation).

COMPLETE_FRACTION reaching 1 would be a telling event in case of a backfill with defined start and stop. Would not be strikingly useful for a typical recurring task without stop defined, fluctuating close to 1.

DELAY is measured from the first found missing datehour till (current time + hours_forward), or till stop if it is defined. In hours for Hourly. TBD different units for other frequencies? TODO any different for reverse mode? From first missing till last missing? From last gap till stop?

`COMPLETE_COUNT` = 'event.tools.range.complete.count'

`COMPLETE_FRACTION` = 'event.tools.range.complete.fraction'

`DELAY` = 'event.tools.range.delay'

class `luigi.tools.range.`**`RangeBase`**`(`**args*, ***kwargs*`)`

Bases: *`luigi.task.WrapperTask`*

Produces a contiguous completed range of a recurring task.

Made for the common use case where a task is parameterized by e.g. DateParameter, and assurance is needed that any gaps arising from downtime are eventually filled.

Emits events that one can use to monitor gaps and delays.

At least one of start and stop needs to be specified.

(This is quite an abstract base class for subclasses with different datetime parameter class, e.g. DateParameter, DateHourParameter, ..., and different parameter naming, e.g. days_back/forward, hours_back/forward, ..., as well as different documentation wording, for good user experience.)

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`of` = <luigi.parameter.Parameter object>

`start` = <luigi.parameter.Parameter object>

`stop` = <luigi.parameter.Parameter object>

`reverse` = <luigi.parameter.BoolParameter object>

`task_limit` = <luigi.parameter.IntParameter object>

`now` = <luigi.parameter.IntParameter object>

`datetime_to_parameter``(`*dt*`)`

`parameter_to_datetime``(`*p*`)`

`moving_start``(`*now*`)`

Returns a datetime from which to ensure contiguousness in the case when start is None or unfeasibly far back.

moving_stop (*now*)

Returns a datetime till which to ensure contiguousness in the case when stop is None or unfeasibly far forward.

finite_datetimes (*finite_start*, *finite_stop*)

Returns the individual datetimes in interval [finite_start, finite_stop) for which task completeness should be required, as a sorted list.

requires ()

missing_datetimes (*task_cls*, *finite_datetimes*)

Override in subclasses to do bulk checks.

Returns a sorted list.

This is a conservative base implementation that brutally checks completeness, instance by instance.

Inadvisable as it may be slow.

task_namespace = None

class luigi.tools.range.**RangeDailyBase** (**args*, ***kwargs*)

Bases: *luigi.tools.range.RangeBase*

Produces a contiguous completed range of a daily recurring task.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

start = <luigi.parameter.DateParameter object>

stop = <luigi.parameter.DateParameter object>

days_back = <luigi.parameter.IntParameter object>

days_forward = <luigi.parameter.IntParameter object>

datetime_to_parameter (*dt*)

parameter_to_datetime (*p*)

moving_start (*now*)

moving_stop (*now*)

finite_datetimes (*finite_start*, *finite_stop*)

Simply returns the points in time that correspond to turn of day.

task_namespace = None

class luigi.tools.range.**RangeHourlyBase** (**args*, ***kwargs*)

Bases: *luigi.tools.range.RangeBase*

Produces a contiguous completed range of an hourly recurring task.

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`start` = <luigi.parameter.DateHourParameter object>

`stop` = <luigi.parameter.DateHourParameter object>

`hours_back` = <luigi.parameter.IntParameter object>

`hours_forward` = <luigi.parameter.IntParameter object>

`datetime_to_parameter`(*dt*)

`parameter_to_datetime`(*p*)

`moving_start`(*now*)

`moving_stop`(*now*)

`finite_datetimes`(*finite_start*, *finite_stop*)
Simply returns the points in time that correspond to whole hours.

`task_namespace` = None

`luigi.tools.range.`**`most_common`**(*items*)
Wanted functionality from Counters (new in Python 2.7).

`luigi.tools.range.`**`infer_bulk_complete_from_fs`**(*datetimes*, *datetime_to_task*, *datetime_to_re*)
Efficiently determines missing datetimes by filesystem listing.

The current implementation works for the common case of a task writing output to a FileSystemTarget whose path is built using strftime with format like '...%Y...%m...%d...%H...', without custom complete() or exists().

(Eventually Luigi could have ranges of completion as first-class citizens. Then this listing business could be factored away/be provided for explicitly in target API or some kind of a history server.)

class `luigi.tools.range.`**`RangeDaily`**(**args*, ***kwargs*)
Bases: *`luigi.tools.range.RangeDailyBase`*

Efficiently produces a contiguous completed range of a daily recurring task that takes a single DateParameter.

Falls back to infer it from output filesystem listing to facilitate the common case usage.

Convenient to use even from command line, like:

```
luigi --module your.module RangeDaily --of YourActualTask --start 2014-01-01
```

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

`missing_datetimes`(*task_cls*, *finite_datetimes*)

`task_namespace` = None

class `luigi.tools.range.`**`RangeHourly`**(**args*, ***kwargs*)
Bases: *`luigi.tools.range.RangeHourlyBase`*

Efficiently produces a contiguous completed range of an hourly recurring task that takes a single DateHourParameter.

Benefits from bulk_complete information to efficiently cover gaps.

Falls back to infer it from output filesystem listing to facilitate the common case usage.

Convenient to use even from command line, like:

```
luigi --module your.module RangeHourly --of YourActualTask --start 2014-01-01T00
```

Constructor to resolve values for all Parameters.

For example, the Task:

```
class MyTask(luigi.Task):
    count = luigi.IntParameter()
```

can be instantiated as `MyTask(count=10)`.

missing_datetimes(*task_cls*, *finite_datetimes*)

task_namespace = None

11.3.2 Module contents

Sort of a standard library for doing stuff with Tasks at a somewhat abstract level.

Submodule introduced to stop growing util.py unstructured.

11.4 Indices and tables

- genindex
- modindex
- search

l

luigi, 135
luigi.cmdline, 97
luigi.configuration, 97
luigi.contrib, 177
luigi.contrib.bigquery, 142
luigi.contrib.esindex, 145
luigi.contrib.ftp, 149
luigi.contrib.gcs, 150
luigi.contrib.hadoop, 151
luigi.contrib.hadoop_jar, 155
luigi.contrib.hdfs, 142
luigi.contrib.hdfs.abstract_client, 135
luigi.contrib.hdfs.clients, 136
luigi.contrib.hdfs.config, 136
luigi.contrib.hdfs.error, 137
luigi.contrib.hdfs.format, 137
luigi.contrib.hdfs.hadoopcli_clients, 138
luigi.contrib.hdfs.snakebite_client, 139
luigi.contrib.hdfs.target, 141
luigi.contrib.hive, 156
luigi.contrib.mysqldb, 159
luigi.contrib.pig, 159
luigi.contrib.pyspark_runner, 160
luigi.contrib.rdbms, 161
luigi.contrib.redis_store, 162
luigi.contrib.redshift, 162
luigi.contrib.scalding, 166
luigi.contrib.spark, 167
luigi.contrib.sparkey, 171
luigi.contrib.sqla, 172
luigi.contrib.ssh, 175
luigi.contrib.target, 176
luigi.contrib.webhdfs, 177
luigi.date_interval, 98
luigi.db_task_history, 100
luigi.deprecate_kwarg, 101
luigi.event, 102
luigi.file, 102
luigi.format, 103
luigi.hadoop, 105
luigi.hadoop_jar, 105
luigi.hdfs, 105
luigi.hive, 105
luigi.interface, 105
luigi.lock, 108
luigi.mock, 108
luigi.mrrunner, 109
luigi.notifications, 109
luigi.parameter, 110
luigi.postgres, 113
luigi.process, 115
luigi.rpc, 115
luigi.s3, 116
luigi.scalding, 119
luigi.scheduler, 119
luigi.server, 122
luigi.target, 123
luigi.task, 125
luigi.task_history, 130
luigi.task_register, 131
luigi.task_status, 131
luigi.tools, 182
luigi.tools.deps, 177
luigi.tools.luigi_grep, 178
luigi.tools.parse_task, 178
luigi.tools.range, 178
luigi.util, 132
luigi.webhdfs, 133
luigi.worker, 133

Index

A

abort() (luigi.contrib.hdfs.format.HdfsAtomicWriteDirPipe method), 53, 137
abort() (luigi.contrib.hdfs.format.HdfsAtomicWritePipe method), 53, 137
abort() (luigi.format.OutputPipeProcessWrapper method), 103
acquire_for() (in module luigi.lock), 108
add() (luigi.worker.Worker method), 135
add_config_path() (luigi.configuration.LuigiConfigParser class method), 98
add_failure() (luigi.scheduler.Failures method), 120
add_failure() (luigi.scheduler.Task method), 120
add_global_parameters() (in module luigi.interface), 106
add_info() (luigi.scheduler.Worker method), 120
add_link() (luigi.contrib.hadoop.JobTask method), 69, 153
add_task (luigi.scheduler.Scheduler attribute), 119
add_task() (luigi.rpc.RemoteScheduler method), 115
add_task() (luigi.scheduler.CentralPlannerScheduler method), 121
add_task_parameters() (in module luigi.interface), 106
add_to_cmdline_parser() (luigi.parameter.Parameter method), 112
add_worker() (luigi.rpc.RemoteScheduler method), 116
add_worker() (luigi.scheduler.CentralPlannerScheduler method), 121
ALL_METHOD_NAMES (luigi.contrib.target.CascadingClient attribute), 92, 176
AllRunHandler (class in luigi.server), 122
AMBIGUOUS_CLASS (luigi.task_register.Register attribute), 131
ApacheHiveCommandClient (class in luigi.contrib.hive), 73, 157
app (luigi.contrib.spark.PySpark1xJob attribute), 87, 171
app (luigi.contrib.spark.PySparkTask attribute), 84, 169
app (luigi.contrib.spark.Spark1xJob attribute), 86, 171
app (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
app() (in module luigi.server), 123
app_command() (luigi.contrib.spark.PySparkTask method), 85, 169
app_command() (luigi.contrib.spark.SparkSubmitTask method), 84, 169
app_options() (luigi.contrib.spark.Spark1xBackwardCompat method), 86, 170
app_options() (luigi.contrib.spark.SparkSubmitTask method), 83, 168
apply_async() (luigi.worker.SingleProcessPool method), 133
archives (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
ArgParseInterface (class in luigi.interface), 106
args() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 72, 156
args() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
assistant (luigi.interface.core attribute), 106
assistant (luigi.scheduler.Worker attribute), 120
AsyncCompletionException, 133
atomic_file (class in luigi.file), 102
atomic_output() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 72, 156
atomic_output() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
AtomicFtpFile (class in luigi.contrib.ftp), 65, 149
AtomicGCSFile (class in luigi.contrib.gcs), 66, 151
AtomicLocalFile (class in luigi.target), 125
AtomicRemoteFileWriter (class in luigi.contrib.ssh), 91, 176
AtomicS3File (class in luigi.s3), 117
AtomicWebHdfsFile (class in luigi.contrib.webhdfs), 92, 177
attach() (in module luigi.contrib.hadoop), 70, 154
aws_access_key_id (luigi.contrib.redshift.S3CopyToTable attribute), 78, 163
aws_secret_access_key (luigi.contrib.redshift.S3CopyToTable attribute), 79, 163

B

BaseHadoopJobTask (class in luigi.contrib.hadoop), 67, 151
BaseTaskHistoryHandler (class in luigi.server), 122
BaseWrapper (class in luigi.format), 103
BATCH (luigi.contrib.bigquery.QueryMode attribute), 58, 142
batch_counter_default (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 67, 151
BigqueryClient (class in luigi.contrib.bigquery), 59, 143
BigqueryLoadTask (class in luigi.contrib.bigquery), 60, 144
BigqueryRunQueryTask (class in luigi.contrib.bigquery), 61, 145
BigqueryTarget (class in luigi.contrib.bigquery), 60, 144
BooleanParameter (class in luigi.parameter), 113
BoolParameter (class in luigi.parameter), 113
BQDataset (class in luigi.contrib.bigquery), 59, 143
BQTable (class in luigi.contrib.bigquery), 59, 143
BROKEN_TASK (luigi.event.Event attribute), 102
build() (in module luigi.interface), 108
build_job_jar() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
bulk_complete() (luigi.task.MixinNaiveBulkComplete class method), 129
bulk_complete() (luigi.task.Task class method), 128
BulkCompleteNotImplementedError, 126
ByIdHandler (class in luigi.server), 123
ByNameHandler (class in luigi.server), 122
ByParamsHandler (class in luigi.server), 123
bytes_per_reducer (luigi.contrib.hive.HiveQueryTask attribute), 73, 157
Bzip2Format (class in luigi.format), 105

C

call_check() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient static method), 54, 138
can_disable() (luigi.scheduler.Task method), 120
CascadingClient (class in luigi.contrib.target), 92, 176
CentralPlannerScheduler (class in luigi.scheduler), 121
ChainFormat (class in luigi.format), 104
check_complete() (in module luigi.worker), 134
check_output() (luigi.contrib.ssh.RemoteContext method), 91, 175
check_pid() (in module luigi.process), 115
chmod() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 135
chmod() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
chmod() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 140
chown() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 135
chown() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
chown() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 140
chunk_size (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
chunk_size (luigi.contrib.sqla.CopyToTable attribute), 90, 174
clear() (luigi.mock.MockFileSystem method), 109
clear() (luigi.scheduler.Failures method), 120
clear_instance_cache() (luigi.task_register.Register class method), 131
client (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
client_version (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
clone() (luigi.task.Task method), 127
close() (luigi.contrib.hdfs.format.HdfsAtomicWriteDirPipe method), 53, 137
close() (luigi.contrib.hdfs.format.HdfsAtomicWritePipe method), 53, 137
close() (luigi.contrib.ssh.AtomicRemoteFileWriter method), 91, 176
close() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 92, 177
close() (luigi.format.InputPipeProcessWrapper method), 103
close() (luigi.format.OutputPipeProcessWrapper method), 103
close() (luigi.s3.ReadableS3File method), 117
close() (luigi.target.AtomicLocalFile method), 125
column_separator (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
column_separator (luigi.contrib.sqla.CopyToTable attribute), 90, 174
columns (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
columns (luigi.contrib.sqla.CopyToTable attribute), 90, 174
combiner (luigi.contrib.hadoop.JobTask attribute), 69, 153
command (luigi.contrib.hdfs.config.hadoopcli attribute), 53, 136
common_params() (in module luigi.util), 132
CompatibleHdfsFormat (class in luigi.contrib.hdfs.format), 54, 138
complete() (luigi.task.Task method), 127
complete() (luigi.task.WrapperTask method), 129
COMPLETE_COUNT (luigi.tools.range.RangeEvent attribute), 94, 179
COMPLETE_FRACTION (luigi.tools.range.RangeEvent attribute), 94, 179
conf (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
Config (class in luigi.task), 129

connect() (luigi.contrib.mysqldb.MySqlTarget method), 75, 159
connect() (luigi.postgres.PostgresTarget method), 114
connect_args (luigi.contrib.sqla.CopyToTable attribute), 90, 174
connect_args (luigi.contrib.sqla.SQLAlchemyTarget attribute), 89, 174
connection_reset_wait_seconds (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 81, 165
connection_string() (luigi.contrib.sqla.CopyToTable method), 90, 174
copies (class in luigi.util), 132
copy() (luigi.contrib.bigquery.BigqueryClient method), 60, 144
copy() (luigi.contrib.gcs.GCSClient method), 66, 150
copy() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 135
copy() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
copy() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 140
copy() (luigi.contrib.rdbms.CopyToTable method), 77, 162
copy() (luigi.contrib.redshift.S3CopyJSONToTable method), 80, 164
copy() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
copy() (luigi.contrib.sqla.CopyToTable method), 90, 175
copy() (luigi.file.LocalTarget method), 103
copy() (luigi.postgres.CopyToTable method), 115
copy() (luigi.s3.S3Client method), 116
copy_json_options (luigi.contrib.redshift.S3CopyJSONToTable attribute), 80, 164
copy_options (luigi.contrib.redshift.S3CopyToTable attribute), 79, 163
CopyToIndex (class in luigi.contrib.esindex), 63, 147
CopyToTable (class in luigi.contrib.rdbms), 76, 161
CopyToTable (class in luigi.contrib.sqla), 90, 174
CopyToTable (class in luigi.postgres), 114
core (class in luigi.interface), 105
count() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 135
count() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
count() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 140
count_uniques (luigi.worker.worker attribute), 134
counter (luigi.parameter.Parameter attribute), 111
create_disposition (luigi.contrib.bigquery.BigqueryRunQueryTask attribute), 61, 145
create_hadoopcli_client() (in module luigi.contrib.hdfs.hadoopcli_clients), 54, 138
CREATE_IF_NEEDED (luigi.contrib.bigquery.CreateDisposition attribute), 58, 142
create_index() (luigi.contrib.esindex.CopyToIndex method), 64, 148
create_local_scheduler() (luigi.interface.WorkerSchedulerFactory method), 106
create_marker_index() (luigi.contrib.esindex.ElasticsearchTarget method), 63, 147
create_marker_table() (luigi.contrib.mysqldb.MySqlTarget method), 75, 159
create_marker_table() (luigi.contrib.sqla.SQLAlchemyTarget method), 89, 174
create_marker_table() (luigi.postgres.PostgresTarget method), 114
CREATE_NEVER (luigi.contrib.bigquery.CreateDisposition attribute), 58, 142
create_packages_archive() (in module luigi.contrib.hadoop), 70, 154
create_remote_scheduler() (luigi.interface.WorkerSchedulerFactory method), 106
create_subprocess() (luigi.format.InputPipeProcessWrapper method), 103
create_table() (luigi.contrib.rdbms.CopyToTable method), 77, 161
create_table() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
create_table() (luigi.contrib.sqla.CopyToTable method), 90, 175
create_worker() (luigi.interface.WorkerSchedulerFactory method), 106
CreateDisposition (class in luigi.contrib.bigquery), 58, 142
CSV (luigi.contrib.bigquery.SourceFormat attribute), 58, 142
Custom (class in luigi.date_interval), 100

D

daemonize() (in module luigi.process), 115
data_interchange_format (luigi.contrib.hadoop.JobTask attribute), 69, 153
database (luigi.contrib.hive.ExternalHiveTask attribute), 74, 158
database (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
database (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 81, 165
dataset (luigi.contrib.bigquery.BQTable attribute), 59, 143
dataset_exists() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
dataset_id (luigi.contrib.bigquery.BQDataset attribute), 59, 143

DATASTORE_BACKUP (luigi.contrib.bigquery.SourceFormat attribute), 58, 142
Date (class in luigi.date_interval), 99
date_format (luigi.parameter.DateHourParameter attribute), 112
date_format (luigi.parameter.DateMinuteParameter attribute), 112
DateHourParameter (class in luigi.parameter), 112
DateInterval (class in luigi.date_interval), 98
DateIntervalParameter (class in luigi.parameter), 113
DateMinuteParameter (class in luigi.parameter), 112
DateParameter (class in luigi.parameter), 112
dates() (luigi.date_interval.DateInterval method), 99
datetime_to_parameter() (luigi.tools.range.RangeBase method), 95, 179
datetime_to_parameter() (luigi.tools.range.RangeDailyBase method), 95, 180
datetime_to_parameter() (luigi.tools.range.RangeHourlyBase method), 96, 181
days_back (luigi.tools.range.RangeDailyBase attribute), 95, 180
days_forward (luigi.tools.range.RangeDailyBase attribute), 95, 180
DbTaskHistory (class in luigi.db_task_history), 100
DefaultHadoopJobRunner (class in luigi.contrib.hadoop), 68, 152
DELAY (luigi.tools.range.RangeEvent attribute), 94, 179
delegates() (in module luigi.util), 132
delete_dataset() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
delete_index() (luigi.contrib.esindex.CopyToIndex method), 64, 148
delete_table() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
dep_graph() (luigi.rpc.RemoteScheduler method), 115
dep_graph() (luigi.scheduler.CentralPlannerScheduler method), 121
DEPENDENCY_DISCOVERED (luigi.event.Event attribute), 102
dependency_jars() (luigi.contrib.spark.Spark1xBackwardCompat method), 86, 170
DEPENDENCY_MISSING (luigi.event.Event attribute), 102
DEPENDENCY_PRESENT (luigi.event.Event attribute), 102
deploy_mode (luigi.contrib.spark.PySparkTask attribute), 84, 169
deploy_mode (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
deprecate_kwarg() (in module luigi.deprecate_kwarg), 101
deps() (luigi.contrib.hadoop.BaseHadoopJobTask method), 67, 151
deps() (luigi.contrib.spark.SparkJob method), 85, 170
deps() (luigi.task.Task method), 128
DequeQueue (class in luigi.worker), 133
dereference() (in module luigi.contrib.hadoop), 70, 154
deserialize (luigi.contrib.hadoop.JobTask attribute), 69, 153
dfs_paths() (in module luigi.tools.deps), 93, 177
disable_failures (luigi.scheduler.scheduler attribute), 119
disable_hard_timeout (luigi.scheduler.scheduler attribute), 119
disable_instance_cache() (luigi.task_register.Register class method), 131
disable_persist (luigi.scheduler.scheduler attribute), 119
disable_window (luigi.scheduler.scheduler attribute), 119
disabled (luigi.task.Task attribute), 126
do_truncate_table() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
doc_type (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
docs() (luigi.contrib.esindex.CopyToIndex method), 64, 148
does_table_exist() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
download() (luigi.contrib.gcs.GCSClient method), 66, 150
download() (luigi.contrib.webhdfs.WebHdfsClient method), 93, 177
driver_class_path (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
driver_cores (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
driver_java_options (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
driver_library_path (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
driver_memory (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
dump() (luigi.contrib.hadoop.JobTask method), 69, 153
dump() (luigi.scheduler.CentralPlannerScheduler method), 121
dump() (luigi.scheduler.SimpleTaskState method), 120
DuplicateParameterException, 110
DynamicArgParseInterface (class in luigi.interface), 107

E

echo (luigi.contrib.sqla.CopyToTable attribute), 90, 174
effective_user (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
ElasticsearchTarget (class in luigi.contrib.esindex), 62, 146
email_type() (in module luigi.notifications), 109
engine (luigi.contrib.sqla.SQLAlchemyTarget attribute), 89, 174

engine (luigi.contrib.sqla.SQLAlchemyTarget.Connection attribute), 89, 174
ensure_hist_size() (luigi.contrib.esindex.ElasticsearchTarget method), 63, 147
entry_class (luigi.contrib.spark.Spark1xJob attribute), 86, 171
entry_class (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
error_task_names() (in module luigi.interface), 106
Event (class in luigi.event), 102
event_handler() (luigi.task.Task class method), 127
event_name (luigi.db_task_history.TaskEvent attribute), 101
events (luigi.db_task_history.TaskRecord attribute), 101
executor_cores (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
executor_memory (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
exists() (luigi.contrib.bigquery.BigqueryTarget method), 60, 144
exists() (luigi.contrib.esindex.ElasticsearchTarget method), 63, 147
exists() (luigi.contrib.ftp.RemoteFileSystem method), 65, 149
exists() (luigi.contrib.ftp.RemoteTarget method), 66, 150
exists() (luigi.contrib.gcs.GCSClient method), 66, 150
exists() (luigi.contrib.gcs.GCSFlagTarget method), 67, 151
exists() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
exists() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClientApache1 method), 55, 139
exists() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 55, 139
exists() (luigi.contrib.hive.HivePartitionTarget method), 74, 158
exists() (luigi.contrib.hive.HiveTableTarget method), 74, 158
exists() (luigi.contrib.mysqldb.MySqlTarget method), 75, 159
exists() (luigi.contrib.redis_store.RedisTarget method), 78, 162
exists() (luigi.contrib.sqla.SQLAlchemyTarget method), 89, 174
exists() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
exists() (luigi.contrib.webhdfs.WebHdfsClient method), 92, 177
exists() (luigi.file.LocalFileSystem method), 102
exists() (luigi.mock.MockFileSystem method), 108
exists() (luigi.mock.MockTarget method), 109
exists() (luigi.postgres.PostgresTarget method), 114
exists() (luigi.s3.S3Client method), 116
exists() (luigi.s3.S3FlagTarget method), 118
exists() (luigi.target.FileSystem method), 124
exists() (luigi.target.FileSystemTarget method), 125
exists() (luigi.target.Target method), 123
ExternalHiveTask (class in luigi.contrib.hive), 74, 158
externalize() (in module luigi.task), 129
ExternalTask (class in luigi.task), 129
extra_elasticsearch_args (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
extra_files() (luigi.contrib.hadoop.JobTask method), 69, 153
extra_jars() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
extra_modules() (luigi.contrib.hadoop.JobTask method), 69, 153
extract_packages_archive() (luigi.mrrunner.Runner method), 109

F

FAILURE (luigi.event.Event attribute), 102
Failures (class in luigi.scheduler), 120
family (luigi.tools.deps.upstream attribute), 93, 178
fetch_error() (luigi.rpc.RemoteScheduler method), 116
fetch_error() (luigi.scheduler.CentralPlannerScheduler method), 122
fetch_task_failures() (in module luigi.contrib.hadoop), 70, 154
File (class in luigi.file), 103
FileAlreadyExists, 123
FileNotFoundException, 116
files (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
FileSystem (class in luigi.target), 124
FileSystemException, 123
FileSystemTarget (class in luigi.target), 124
FileWrapper (class in luigi.format), 103
final_combiner (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 67, 152
final_mapper (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 67, 152
final_reducer (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 67, 152
find_all_by_name() (luigi.db_task_history.DbTaskHistory method), 100
find_all_by_parameters() (luigi.db_task_history.DbTaskHistory method), 100
find_all_events() (luigi.db_task_history.DbTaskHistory method), 100
find_all_runs() (luigi.db_task_history.DbTaskHistory method), 100
find_deps() (in module luigi.tools.deps), 93, 178
find_deps_cli() (in module luigi.tools.deps), 93, 178
find_latest_runs() (luigi.db_task_history.DbTaskHistory method), 100

find_task_by_id() (luigi.db_task_history.DbTaskHistory method), 100
finish() (luigi.contrib.hadoop.HadoopJobRunner method), 68, 152
finite_datetimes() (luigi.tools.range.RangeBase method), 95, 180
finite_datetimes() (luigi.tools.range.RangeDailyBase method), 95, 180
finite_datetimes() (luigi.tools.range.RangeHourlyBase method), 96, 181
fix_paths() (in module luigi.contrib.hadoop_jar), 71, 155
fix_time() (in module luigi.scheduler), 119
flag (luigi.s3.S3FlagTask attribute), 118
flatten() (in module luigi.contrib.hadoop), 70, 154
flatten() (in module luigi.task), 130
flatten_output() (in module luigi.task), 130
FloatParameter (class in luigi.parameter), 112
fn (luigi.file.LocalTarget attribute), 103
folder_paths (luigi.contrib.redshift.RedshiftManifestTask attribute), 80, 165
Format (class in luigi.format), 104
from_bqtable() (luigi.contrib.bigquery.BigqueryTarget class method), 60, 144
from_date() (luigi.date_interval.Date class method), 99
from_date() (luigi.date_interval.DateInterval class method), 99
from_date() (luigi.date_interval.Month class method), 99
from_date() (luigi.date_interval.Week class method), 99
from_date() (luigi.date_interval.Year class method), 100
from_str_params() (luigi.task.Task class method), 127
from_utc() (in module luigi.server), 122
fs (luigi.contrib.ftp.AtomicFtpFile attribute), 65, 149
fs (luigi.contrib.ftp.RemoteTarget attribute), 65, 149
fs (luigi.contrib.gcs.GCSFlagTarget attribute), 67, 151
fs (luigi.contrib.gcs.GCSTarget attribute), 67, 151
fs (luigi.contrib.hdfs.target.HdfsTarget attribute), 57, 141
fs (luigi.contrib.ssh.AtomicRemoteFileWriter attribute), 92, 176
fs (luigi.contrib.ssh.RemoteTarget attribute), 92, 176
fs (luigi.contrib.webhdfs.WebHdfsTarget attribute), 92, 177
fs (luigi.file.LocalTarget attribute), 102
fs (luigi.mock.MockTarget attribute), 109
fs (luigi.s3.S3FlagTarget attribute), 117
fs (luigi.s3.S3Target attribute), 117
fs (luigi.target.FileSystemTarget attribute), 125

G

GCSClient (class in luigi.contrib.gcs), 66, 150
GCSFlagTarget (class in luigi.contrib.gcs), 67, 151
GCSTarget (class in luigi.contrib.gcs), 67, 151
generate_email() (in module luigi.notifications), 109
generate_tmp_path() (luigi.file.atomic_file method), 102
generate_tmp_path() (luigi.target.AtomicLocalFile method), 125
get() (luigi.configuration.LuigiConfigParser method), 98
get() (luigi.contrib.ftp.RemoteFileSystem method), 65, 149
get() (luigi.contrib.ftp.RemoteTarget method), 66, 150
get() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 135
get() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 55, 138
get() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 140
get() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
get() (luigi.contrib.ssh.RemoteTarget method), 92, 176
get() (luigi.server.AllRunHandler method), 122
get() (luigi.server.ByIdHandler method), 123
get() (luigi.server.ByNameHandler method), 122
get() (luigi.server.ByParamsHandler method), 123
get() (luigi.server.RecentRunHandler method), 122
get() (luigi.server.RootPathHandler method), 123
get() (luigi.server.RPCHandler method), 122
get() (luigi.server.SelectedRunHandler method), 122
get() (luigi.server.StaticFileHandler method), 123
get() (luigi.worker.DequeQueue method), 133
get_active_tasks() (luigi.scheduler.SimpleTaskState method), 120
get_active_workers() (luigi.scheduler.SimpleTaskState method), 121
get_all_data() (luigi.mock.MockFileSystem method), 108
get_all_params() (luigi.task_register.Register class method), 131
get_assistants() (luigi.scheduler.SimpleTaskState method), 121
get_autoconfig_client() (in module luigi.contrib.hdfs.clients), 52, 136
get_bite() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 55, 139
get_build_dir() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_config() (in module luigi.configuration), 98
get_config() (luigi.contrib.webhdfs.WebHdfsClient method), 92, 177
get_configured_hadoop_version() (in module luigi.contrib.hdfs.config), 53, 137
get_configured_hdfs_client() (in module luigi.contrib.hdfs.config), 53, 137
get_data() (luigi.mock.MockFileSystem method), 108
get_default_client() (in module luigi.contrib.hive), 73, 157
get_default_format() (in module luigi.format), 105
get_environment() (luigi.contrib.spark.SparkSubmitTask method), 84, 168
get_extra_files() (in module luigi.contrib.hadoop), 70,

155
get_global_parameters() (in module luigi.interface), 106
get_hive_syntax() (in module luigi.contrib.hive), 72, 156
get_info() (in module luigi.lock), 108
get_job_class() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_key() (luigi.s3.S3Client method), 116
get_libjars() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_log_format() (in module luigi.process), 115
get_necessary_tasks() (luigi.scheduler.SimpleTaskState method), 121
get_param_values() (luigi.task.Task class method), 127
get_params() (luigi.task.Task class method), 127
get_pending_tasks() (luigi.scheduler.SimpleTaskState method), 120
get_pending_tasks() (luigi.scheduler.Worker method), 120
get_previous_completed() (in module luigi.util), 133
get_provided_jars() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_running_tasks() (luigi.scheduler.SimpleTaskState method), 120
get_scala_jars() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_scalding_core() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_scalding_jars() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_spool_handler() (in module luigi.process), 115
get_task() (luigi.scheduler.SimpleTaskState method), 121
get_task_cls() (luigi.task_register.Register class method), 131
get_task_parameters() (in module luigi.interface), 106
get_task_requires() (in module luigi.tools.deps), 93, 177
get_template_path() (luigi.server.BaseTaskHistoryHandler method), 122
get_tmp_job_jar() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
get_work (luigi.scheduler.Scheduler attribute), 119
get_work() (luigi.rpc.RemoteScheduler method), 115
get_work() (luigi.scheduler.CentralPlannerScheduler method), 121
get_worker() (luigi.scheduler.SimpleTaskState method), 121
get_worker_ids() (luigi.scheduler.SimpleTaskState method), 121
getboolean() (luigi.configuration.LuigiConfigParser method), 98
getfloat() (luigi.configuration.LuigiConfigParser method), 98
getint() (luigi.configuration.LuigiConfigParser method), 98
getintdict() (luigi.configuration.LuigiConfigParser method), 98
getmerge() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 55, 138
getpaths() (in module luigi.task), 130
getpcmd() (in module luigi.lock), 108
glob_exists() (luigi.contrib.hdfs.target.HdfsTarget method), 57, 141
graph() (luigi.rpc.RemoteScheduler method), 115
graph() (luigi.scheduler.CentralPlannerScheduler method), 121
graph_url (luigi.tools.luigi_grep.LuigiGrep attribute), 93, 178
group() (luigi.contrib.hadoop.LocalJobRunner method), 70, 154
GzipFormat (class in luigi.format), 104

H

hadoop (class in luigi.contrib.hadoop), 70, 155
hadoop_conf_dir (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
hadoopcli (class in luigi.contrib.hdfs.config), 52, 136
HadoopJarJobError, 71, 155
HadoopJarJobRunner (class in luigi.contrib.hadoop_jar), 71, 155
HadoopJarJobTask (class in luigi.contrib.hadoop_jar), 71, 155
HadoopJobError, 68, 152
HadoopJobRunner (class in luigi.contrib.hadoop), 68, 152
HadoopRunContext (class in luigi.contrib.hadoop), 68, 152
has_excessive_failures() (luigi.scheduler.Task method), 120
has_task() (luigi.scheduler.SimpleTaskState method), 121
has_task_value() (luigi.parameter.Parameter method), 111
has_value (luigi.parameter.Parameter attribute), 111
hdfs (class in luigi.contrib.hdfs.config), 52, 136
hdfs_reader() (luigi.contrib.hdfs.format.CompatibleHdfsFormat method), 54, 138
hdfs_reader() (luigi.contrib.hdfs.format.PlainDirFormat method), 54, 138
hdfs_reader() (luigi.contrib.hdfs.format.PlainFormat method), 54, 137
hdfs_writer() (luigi.contrib.hdfs.format.CompatibleHdfsFormat method), 54, 138
hdfs_writer() (luigi.contrib.hdfs.format.PlainDirFormat method), 54, 138
hdfs_writer() (luigi.contrib.hdfs.format.PlainFormat method), 54, 137
HdfsAtomicWriteDirPipe (class in luigi.contrib.hdfs.format), 53, 137
HdfsAtomicWritePipe (class in luigi.contrib.hdfs.format), 53, 137

HdfsClient (class in luigi.contrib.hdfs.hadoopcli_clients), 54, 138
HdfsClientApache1 (class in luigi.contrib.hdfs.hadoopcli_clients), 55, 139
HdfsClientCdh3 (class in luigi.contrib.hdfs.hadoopcli_clients), 55, 139
HDFSCliError, 53, 137
HdfsFileSystem (class in luigi.contrib.hdfs.abstract_client), 51, 135
HdfsReadPipe (class in luigi.contrib.hdfs.format), 53, 137
HdfsTarget (class in luigi.contrib.hdfs.target), 57, 141
HiveClient (class in luigi.contrib.hive), 72, 156
HiveCommandClient (class in luigi.contrib.hive), 72, 156
HiveCommandError, 72, 156
hiveconfs() (luigi.contrib.hive.HiveQueryTask method), 73, 158
HivePartitionTarget (class in luigi.contrib.hive), 74, 158
HiveQueryRunner (class in luigi.contrib.hive), 74, 158
HiveQueryTask (class in luigi.contrib.hive), 73, 157
hiverc() (luigi.contrib.hive.HiveQueryTask method), 73, 157
HiveTableTarget (class in luigi.contrib.hive), 74, 158
HiveThriftContext (class in luigi.contrib.hive), 73, 157
host (luigi.contrib.esindex.CopyToIndex attribute), 63, 147
host (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
host (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 81, 165
host (luigi.db_task_history.TaskRecord attribute), 101
hours() (luigi.date_interval.DateInterval method), 99
hours_back (luigi.tools.range.RangeHourlyBase attribute), 96, 181
hours_forward (luigi.tools.range.RangeHourlyBase attribute), 96, 181
http_auth (luigi.contrib.esindex.CopyToIndex attribute), 63, 147

I

id (luigi.db_task_history.TaskEvent attribute), 101
id (luigi.db_task_history.TaskRecord attribute), 101
id_to_name_and_params() (in module luigi.task), 126
id_to_name_and_params() (in module luigi.tools.parse_task), 94, 178
inactivate_tasks() (luigi.scheduler.SimpleTaskState method), 121
inactivate_workers() (luigi.scheduler.SimpleTaskState method), 121
incr_counter() (luigi.contrib.hadoop.JobTask method), 69, 153
index (luigi.contrib.esindex.CopyToIndex attribute), 63, 147
infer_bulk_complete_from_fs() (in module luigi.tools.range), 96, 181
inherits (class in luigi.util), 132
init_combiner() (luigi.contrib.hadoop.JobTask method), 69, 153
init_copy() (luigi.contrib.rdbms.CopyToTable method), 77, 161
init_hadoop() (luigi.contrib.hadoop.BaseHadoopJobTask method), 67, 152
init_local() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
init_mapper() (luigi.contrib.hadoop.JobTask method), 69, 153
init_reducer() (luigi.contrib.hadoop.JobTask method), 69, 153
initialize() (luigi.server.BaseTaskHistoryHandler method), 122
initialize() (luigi.server.RPCHandler method), 122
initialized() (luigi.task.Task method), 127
input (luigi.contrib.hdfs.format.PlainDirFormat attribute), 54, 138
input (luigi.contrib.hdfs.format.PlainFormat attribute), 53, 137
input (luigi.format.Bzip2Format attribute), 105
input (luigi.format.GzipFormat attribute), 105
input (luigi.format.NewlineFormat attribute), 104
input (luigi.format.TextFormat attribute), 104
input() (luigi.contrib.spark.SparkJob method), 85, 170
input() (luigi.task.Task method), 128
input_hadoop() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
input_local() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
input_local() (luigi.contrib.spark.SparkJob method), 85, 170
InputPipeProcessWrapper (class in luigi.format), 103
instance() (luigi.configuration.LuigiConfigParser class method), 98
INTERACTIVE (luigi.contrib.bigquery.QueryMode attribute), 58, 142
Interface (class in luigi.interface), 106
internal_reader() (luigi.contrib.hadoop.JobTask method), 69, 153
internal_serialize (luigi.contrib.hadoop.JobTask attribute), 69, 153
internal_writer() (luigi.contrib.hadoop.JobTask method), 69, 153
IntParameter (class in luigi.parameter), 112
InvalidDeleteException, 66, 116, 150
inverse_dep_graph() (luigi.rpc.RemoteScheduler method), 115
inverse_dep_graph() (luigi.scheduler.CentralPlannerScheduler method), 122
is_dir() (luigi.s3.S3Client method), 117

is_trivial_worker() (luigi.scheduler.Worker method), 120
is_writable() (luigi.contrib.hdfs.target.HdfsTarget method), 58, 142
isdir() (luigi.contrib.gcs.GCSClient method), 66, 150
isdir() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
isdir() (luigi.file.LocalFileSystem method), 102
isdir() (luigi.mock.MockFileSystem method), 109
isdir() (luigi.s3.S3Client method), 117
isdir() (luigi.target.FileSystem method), 124

J

jar() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 71, 156
jar() (luigi.contrib.scalding.ScaldingJobTask method), 82, 167
jar() (luigi.contrib.spark.Spark1xJob method), 86, 171
jar() (luigi.contrib.spark.SparkJob method), 85, 170
jars (luigi.contrib.spark.Spark1xBackwardCompat attribute), 86, 170
jars (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
job_args() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
job_args() (luigi.contrib.spark.Spark1xBackwardCompat method), 86, 170
job_args() (luigi.contrib.spark.SparkJob method), 85, 170
job_class() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
job_class() (luigi.contrib.spark.Spark1xJob method), 86, 171
job_class() (luigi.contrib.spark.SparkJob method), 85, 170
job_runner() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
job_runner() (luigi.contrib.hadoop.JobTask method), 69, 153
job_runner() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 72, 156
job_runner() (luigi.contrib.hive.HiveQueryTask method), 74, 158
job_runner() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
jobconfs() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
jobconfs() (luigi.contrib.hadoop.JobTask method), 69, 153
JobRunner (class in luigi.contrib.hadoop), 68, 153
JobTask (class in luigi.contrib.hadoop), 68, 153
jsonpath (luigi.contrib.redshift.S3CopyJSONToTable attribute), 80, 164

K

keep_alive (luigi.worker.worker attribute), 134
KeepAliveThread (class in luigi.worker), 134
kill_job() (luigi.contrib.hadoop.HadoopRunContext method), 68, 152
kill_job() (luigi.contrib.pig.PigRunContext method), 76, 160
kill_job() (luigi.contrib.spark.SparkRunContext method), 83, 167
KillOpenRedshiftSessions (class in luigi.contrib.redshift), 81, 165

L

list() (luigi.s3.S3Client method), 117
list_datasets() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
list_path() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient static method), 55, 139
list_tables() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
listdir() (luigi.contrib.gcs.GCSClient method), 66, 150
listdir() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 136
listdir() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 55, 138
listdir() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 57, 141
listdir() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
listdir() (luigi.file.LocalFileSystem method), 102
listdir() (luigi.mock.MockFileSystem method), 108
listdir() (luigi.s3.S3Client method), 116
listdir() (luigi.target.FileSystem method), 124
load() (luigi.scheduler.CentralPlannerScheduler method), 121
load() (luigi.scheduler.SimpleTaskState method), 120
load_hadoop_cmd() (in module luigi.contrib.hdfs.config), 53, 136
load_hive_cmd() (in module luigi.contrib.hive), 72, 156
load_task() (in module luigi.task_register), 131
local_scheduler (luigi.interface.core attribute), 106
LocalFileSystem (class in luigi.file), 102
LocalJobRunner (class in luigi.contrib.hadoop), 70, 154
LocalTarget (class in luigi.file), 102
lock_pid_dir (luigi.interface.core attribute), 106
lock_size (luigi.interface.core attribute), 106
logger (in module luigi.contrib.scalding), 81, 166
logging_conf_file (luigi.interface.core attribute), 106
luigi (module), 135
luigi.cmdline (module), 97
luigi.configuration (module), 97
luigi.contrib (module), 93, 177
luigi.contrib.bigquery (module), 58, 142
luigi.contrib.esindex (module), 61, 145
luigi.contrib.ftp (module), 65, 149
luigi.contrib.gcs (module), 66, 150

luigi.contrib.hadoop (module), 67, 151
luigi.contrib.hadoop_jar (module), 71, 155
luigi.contrib.hdfs (module), 58, 142
luigi.contrib.hdfs.abstract_client (module), 51, 135
luigi.contrib.hdfs.clients (module), 52, 136
luigi.contrib.hdfs.config (module), 52, 136
luigi.contrib.hdfs.error (module), 53, 137
luigi.contrib.hdfs.format (module), 53, 137
luigi.contrib.hdfs.hadoopcli_clients (module), 54, 138
luigi.contrib.hdfs.snakebite_client (module), 55, 139
luigi.contrib.hdfs.target (module), 57, 141
luigi.contrib.hive (module), 72, 156
luigi.contrib.mysqldb (module), 75, 159
luigi.contrib.pig (module), 75, 159
luigi.contrib.pyspark_runner (module), 76, 160
luigi.contrib.rdbms (module), 76, 161
luigi.contrib.redis_store (module), 77, 162
luigi.contrib.redshift (module), 78, 162
luigi.contrib.scalding (module), 81, 166
luigi.contrib.spark (module), 83, 167
luigi.contrib.sparkey (module), 87, 171
luigi.contrib.sqla (module), 87, 172
luigi.contrib.ssh (module), 91, 175
luigi.contrib.target (module), 92, 176
luigi.contrib.webhdfs (module), 92, 177
luigi.date_interval (module), 98
luigi.db_task_history (module), 100
luigi.deprecate_kwarg (module), 101
luigi.event (module), 102
luigi.file (module), 102
luigi.format (module), 103
luigi.hadoop (module), 105
luigi.hadoop_jar (module), 105
luigi.hdfs (module), 105
luigi.hive (module), 105
luigi.interface (module), 105
luigi.lock (module), 108
luigi.mock (module), 108
luigi.mrrunner (module), 109
luigi.notifications (module), 109
luigi.parameter (module), 110
luigi.postgres (module), 113
luigi.process (module), 115
luigi.rpc (module), 115
luigi.s3 (module), 116
luigi.scalding (module), 119
luigi.scheduler (module), 119
luigi.server (module), 122
luigi.target (module), 123
luigi.task (module), 125
luigi.task_history (module), 130
luigi.task_register (module), 131
luigi.task_status (module), 131
luigi.tools (module), 97, 182
luigi.tools.deps (module), 93, 177
luigi.tools.luigi_grep (module), 93, 178
luigi.tools.parse_task (module), 94, 178
luigi.tools.range (module), 94, 178
luigi.util (module), 132
luigi.webhdfs (module), 133
luigi.worker (module), 133
luigi_run() (in module luigi.cmdline), 97
LuigiConfigParser (class in luigi.configuration), 98
luigid() (in module luigi.cmdline), 97
LuigiGrep (class in luigi.tools.luigi_grep), 93, 178

M

main() (in module luigi.mrrunner), 109
main() (in module luigi.tools.deps), 93, 178
main() (in module luigi.tools.luigi_grep), 93, 178
main() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 72, 156
main() (luigi.contrib.spark.PySparkTask method), 85, 169
make_dataset() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
makedirs() (luigi.file.LocalTarget method), 102
map_column() (luigi.postgres.CopyToTable method), 115
mapper() (luigi.contrib.hadoop.JobTask method), 69, 153
mapping (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
marker_doc_type (luigi.contrib.esindex.ElasticsearchTarget attribute), 62, 146
marker_index (luigi.contrib.esindex.ElasticsearchTarget attribute), 62, 146
marker_index_document_id() (luigi.contrib.esindex.ElasticsearchTarget method), 62, 146
marker_index_hist_size (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
marker_key() (luigi.contrib.redis_store.RedisTarget method), 78, 162
marker_prefix (luigi.contrib.redis_store.RedisTarget attribute), 78, 162
marker_table (luigi.contrib.mysqldb.MySqlTarget attribute), 75, 159
marker_table (luigi.contrib.redshift.RedshiftTarget attribute), 78, 162
marker_table (luigi.contrib.sqla.SQLAlchemyTarget attribute), 89, 174
marker_table (luigi.postgres.PostgresTarget attribute), 114
master (luigi.contrib.spark.Spark1xBackwardCompat attribute), 86, 170
master (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
max_bad_records (luigi.contrib.bigquery.BigqueryLoadTask attribute), 60, 144
max_reschedules (luigi.worker.worker attribute), 134

max_shown_tasks (luigi.scheduler.scheduler attribute), 119
MetastoreClient (class in luigi.contrib.hive), 73, 157
missing_datetimes() (luigi.tools.range.RangeBase method), 95, 180
missing_datetimes() (luigi.tools.range.RangeDaily method), 97, 181
missing_datetimes() (luigi.tools.range.RangeHourly method), 97, 182
MissingParameterException, 110
MissingParentDirectory, 123
MixedUnicodeBytesFormat (class in luigi.format), 104
MixedUnicodeBytesWrapper (class in luigi.format), 103
MixinNaiveBulkComplete (class in luigi.task), 129
mkdir() (luigi.contrib.gcs.GCSClient method), 66, 150
mkdir() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 136
mkdir() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 55, 138
mkdir() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClientCdh3 method), 55, 139
mkdir() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 57, 140
mkdir() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
mkdir() (luigi.file.LocalFileSystem method), 102
mkdir() (luigi.mock.MockFileSystem method), 109
mkdir() (luigi.s3.S3Client method), 117
mkdir() (luigi.target.FileSystem method), 124
MockFile (class in luigi.mock), 109
MockFileSystem (class in luigi.mock), 108
MockTarget (class in luigi.mock), 109
module (luigi.interface.core attribute), 106
Month (class in luigi.date_interval), 99
most_common() (in module luigi.tools.range), 96, 181
move() (luigi.contrib.hdfs.target.HdfsTarget method), 58, 141
move() (luigi.file.LocalTarget method), 102
move_dir() (luigi.contrib.hdfs.target.HdfsTarget method), 58, 142
move_dir() (luigi.file.LocalTarget method), 102
move_to_final_destination() (luigi.contrib.ftp.AtomicFtpFile method), 65, 149
move_to_final_destination() (luigi.contrib.gcs.AtomicGCSFile method), 66, 151
move_to_final_destination() (luigi.contrib.webhdfs.AtomicWebHdfsFile method), 92, 177
move_to_final_destination() (luigi.file.atomic_file method), 102
move_to_final_destination() (luigi.s3.AtomicS3File method), 117
move_to_final_destination() (luigi.target.AtomicLocalFile method), 125
moving_start() (luigi.tools.range.RangeBase method), 95, 179
moving_start() (luigi.tools.range.RangeDailyBase method), 95, 180
moving_start() (luigi.tools.range.RangeHourlyBase method), 96, 181
moving_stop() (luigi.tools.range.RangeBase method), 95, 179
moving_stop() (luigi.tools.range.RangeDailyBase method), 95, 180
moving_stop() (luigi.tools.range.RangeHourlyBase method), 96, 181
mr_priority (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 68, 152
MultiReplacer (class in luigi.postgres), 113
MySqlTarget (class in luigi.contrib.mysqldb), 75, 159

N

n_reduce_tasks (luigi.contrib.hadoop.JobTask attribute), 69, 154
n_reduce_tasks (luigi.contrib.hive.HiveQueryTask attribute), 73, 157
name (luigi.contrib.spark.PySparkTask attribute), 84, 169
name (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
name (luigi.db_task_history.TaskParameter attribute), 101
name (luigi.db_task_history.TaskRecord attribute), 101
namenode_host (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
namenode_port (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
namespace() (in module luigi.task), 125
NEWLINE_DELIMITED_JSON (luigi.contrib.bigquery.SourceFormat attribute), 59, 143
NewlineFormat (class in luigi.format), 104
NewlineWrapper (class in luigi.format), 103
next() (luigi.date_interval.DateInterval method), 99
NO_DEFAULT (luigi.configuration.LuigiConfigParser attribute), 98
no_lock (luigi.interface.core attribute), 106
NopFormat (class in luigi.format), 104
NopHistory (class in luigi.task_history), 130
NotADirectory, 123
now (luigi.tools.range.RangeBase attribute), 95, 179
null_values (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
num_executors (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
num_failures() (luigi.scheduler.Failures method), 120

O

of (luigi.tools.range.RangeBase attribute), 94, 179
on_failure() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
on_failure() (luigi.task.Task method), 128
on_success() (luigi.task.Task method), 128
open() (luigi.contrib.ftp.RemoteTarget method), 65, 149
open() (luigi.contrib.gcs.GCSTarget method), 67, 151
open() (luigi.contrib.hdfs.target.HdfsTarget method), 57, 141
open() (luigi.contrib.hive.HivePartitionTarget method), 74, 158
open() (luigi.contrib.hive.HiveTableTarget method), 74, 158
open() (luigi.contrib.sqla.SQLAlchemyTarget method), 90, 174
open() (luigi.contrib.ssh.RemoteTarget method), 92, 176
open() (luigi.contrib.webhdfs.WebHdfsTarget method), 92, 177
open() (luigi.file.LocalTarget method), 102
open() (luigi.mock.MockTarget method), 109
open() (luigi.postgres.PostgresTarget method), 114
open() (luigi.s3.S3Target method), 117
open() (luigi.target.FileSystemTarget method), 125
OptParseInterface (class in luigi.interface), 107
output (luigi.contrib.hdfs.format.CompatibleHdfsFormat attribute), 54, 138
output (luigi.contrib.hdfs.format.PlainDirFormat attribute), 54, 138
output (luigi.contrib.hdfs.format.PlainFormat attribute), 54, 137
output (luigi.format.Bzip2Format attribute), 105
output (luigi.format.GzipFormat attribute), 105
output (luigi.format.MixedUnicodeBytesFormat attribute), 104
output (luigi.format.NewlineFormat attribute), 104
output (luigi.format.TextFormat attribute), 104
output() (luigi.contrib.esindex.CopyToIndex method), 64, 148
output() (luigi.contrib.hive.ExternalHiveTask method), 74, 159
output() (luigi.contrib.pig.PigJobTask method), 76, 160
output() (luigi.contrib.rdbms.CopyToTable method), 77, 161
output() (luigi.contrib.redshift.KillOpenRedshiftSessions method), 81, 165
output() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
output() (luigi.contrib.spark.Spark1xBackwardCompat method), 86, 170
output() (luigi.contrib.spark.SparkJob method), 85, 170
output() (luigi.contrib.sqla.CopyToTable method), 90, 175
output() (luigi.postgres.CopyToTable method), 115
output() (luigi.s3.S3EmrTask method), 118
output() (luigi.s3.S3FlagTask method), 118
output() (luigi.s3.S3PathTask method), 118
output() (luigi.task.Task method), 128
OutputPipeProcessWrapper (class in luigi.format), 103

P

parallel_scheduling (luigi.interface.core attribute), 106
Parameter (class in luigi.parameter), 110
parameter_to_datetime() (luigi.tools.range.RangeBase method), 95, 179
parameter_to_datetime() (luigi.tools.range.RangeDailyBase method), 95, 180
parameter_to_datetime() (luigi.tools.range.RangeHourlyBase method), 96, 181
ParameterException, 110
parameters (luigi.db_task_history.TaskRecord attribute), 101
parse() (luigi.date_interval.Custom class method), 100
parse() (luigi.date_interval.Date class method), 99
parse() (luigi.date_interval.DateInterval class method), 99
parse() (luigi.date_interval.Month class method), 99
parse() (luigi.date_interval.Week class method), 99
parse() (luigi.date_interval.Year class method), 100
parse() (luigi.interface.ArgParseInterface method), 107
parse() (luigi.interface.DynamicArgParseInterface method), 107
parse() (luigi.interface.Interface method), 106
parse() (luigi.interface.OptParseInterface method), 107
parse() (luigi.parameter.BoolParameter method), 113
parse() (luigi.parameter.DateHourParameter method), 112
parse() (luigi.parameter.DateIntervalParameter method), 113
parse() (luigi.parameter.DateParameter method), 112
parse() (luigi.parameter.FloatParameter method), 112
parse() (luigi.parameter.IntParameter method), 112
parse() (luigi.parameter.Parameter method), 111
parse() (luigi.parameter.TimeDeltaParameter method), 113
parse_from_args() (luigi.parameter.Parameter method), 112
parse_from_input() (luigi.parameter.Parameter method), 111
parse_task() (luigi.interface.ArgParseInterface method), 107
parser_dest() (luigi.parameter.Parameter method), 112
partition (luigi.contrib.hive.ExternalHiveTask attribute), 74, 159
partition_spec() (luigi.contrib.hive.HiveClient method), 72, 156
partition_spec() (luigi.contrib.hive.HiveCommandClient method), 73, 157

partition_spec() (luigi.contrib.hive.MetastoreClient method), 73, 157
PassThroughOptionParser (class in luigi.interface), 107
password (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
password (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 81, 165
path (luigi.contrib.hive.HivePartitionTarget attribute), 74, 158
path (luigi.contrib.hive.HiveTableTarget attribute), 74, 158
path (luigi.mock.MockTarget attribute), 109
path (luigi.s3.S3EmrTask attribute), 118
path (luigi.s3.S3FlagTask attribute), 118
path (luigi.s3.S3PathTask attribute), 118
pid (luigi.contrib.sqla.SQLAlchemyTarget.Connection attribute), 89, 174
pig_command_path() (luigi.contrib.pig.PigJobTask method), 75, 160
pig_env_vars() (luigi.contrib.pig.PigJobTask method), 75, 160
pig_home() (luigi.contrib.pig.PigJobTask method), 75, 160
pig_options() (luigi.contrib.pig.PigJobTask method), 76, 160
pig_parameters() (luigi.contrib.pig.PigJobTask method), 76, 160
pig_properties() (luigi.contrib.pig.PigJobTask method), 75, 160
pig_script_path() (luigi.contrib.pig.PigJobTask method), 76, 160
PigJobError, 76, 160
PigJobTask (class in luigi.contrib.pig), 75, 159
PigRunContext (class in luigi.contrib.pig), 76, 160
ping (luigi.scheduler.Scheduler attribute), 119
ping() (luigi.rpc.RemoteScheduler method), 115
ping() (luigi.scheduler.CentralPlannerScheduler method), 121
ping_interval (luigi.worker.worker attribute), 134
pipe_reader() (luigi.contrib.hdfs.format.CompatibleHdfsFormat method), 54, 138
pipe_reader() (luigi.contrib.hdfs.format.PlainDirFormat method), 54, 138
pipe_reader() (luigi.contrib.hdfs.format.PlainFormat method), 54, 137
pipe_reader() (luigi.format.Bzip2Format method), 105
pipe_reader() (luigi.format.ChainFormat method), 104
pipe_reader() (luigi.format.Format class method), 104
pipe_reader() (luigi.format.GzipFormat method), 105
pipe_reader() (luigi.format.NopFormat method), 104
pipe_reader() (luigi.format.WrappedFormat method), 104
pipe_writer() (luigi.contrib.hdfs.format.CompatibleHdfsFormat method), 54, 138
pipe_writer() (luigi.contrib.hdfs.format.PlainDirFormat method), 54, 138
pipe_writer() (luigi.contrib.hdfs.format.PlainFormat method), 54, 137
pipe_writer() (luigi.format.Bzip2Format method), 105
pipe_writer() (luigi.format.ChainFormat method), 104
pipe_writer() (luigi.format.Format class method), 104
pipe_writer() (luigi.format.GzipFormat method), 105
pipe_writer() (luigi.format.NopFormat method), 104
pipe_writer() (luigi.format.WrappedFormat method), 104
PlainDirFormat (class in luigi.contrib.hdfs.format), 54, 138
PlainFormat (class in luigi.contrib.hdfs.format), 53, 137
pool (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 68, 152
pool (luigi.contrib.hadoop.hadoop attribute), 71, 155
Popen() (luigi.contrib.ssh.RemoteContext method), 91, 175
port (luigi.contrib.esindex.CopyToIndex attribute), 63, 147
post() (luigi.server.RPCHandler method), 122
PostgresTarget (class in luigi.postgres), 114
prefix_search() (luigi.tools.luigi_grep.LuigiGrep method), 93, 178
prepare_outputs() (luigi.contrib.hive.HiveQueryRunner method), 74, 158
prev() (luigi.date_interval.DateInterval method), 99
previous() (in module luigi.util), 132
print_exception() (in module luigi.mrrunner), 109
priority (luigi.task.Task attribute), 126
process_resources() (luigi.task.Task method), 128
PROCESSING_TIME (luigi.event.Event attribute), 102
program() (luigi.contrib.spark.PySpark1xJob method), 87, 171
project_id (luigi.contrib.bigquery.BQDataset attribute), 59, 143
properties_file (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
prune() (luigi.rpc.RemoteScheduler method), 116
prune() (luigi.scheduler.CentralPlannerScheduler method), 121
prune() (luigi.scheduler.SimpleTaskState method), 121
prune() (luigi.scheduler.Worker method), 120
prune_done_tasks (luigi.scheduler.scheduler attribute), 119
purge_existing_index (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
put() (luigi.contrib.ftp.RemoteFileSystem method), 65, 149
put() (luigi.contrib.ftp.RemoteTarget method), 66, 150
put() (luigi.contrib.gcs.GCSClient method), 66, 150
put() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 135
put() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 55, 138

put() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 140
put() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
put() (luigi.contrib.ssh.RemoteTarget method), 92, 176
put() (luigi.s3.S3Client method), 116
put() (luigi.worker.DequeQueue method), 133
put_multipart() (luigi.s3.S3Client method), 116
put_string() (luigi.contrib.gcs.GCSClient method), 66, 150
put_string() (luigi.s3.S3Client method), 116
py_files (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
py_packages (luigi.contrib.spark.PySparkTask attribute), 84, 169
PySpark1xJob (class in luigi.contrib.spark), 86, 171
PySparkRunner (class in luigi.contrib.pyspark_runner), 76, 160
PySparkTask (class in luigi.contrib.spark), 84, 169

Q

query (luigi.contrib.bigquery.BigqueryRunQueryTask attribute), 61, 145
query() (luigi.contrib.hive.HiveQueryTask method), 73, 157
query_mode (luigi.contrib.bigquery.BigqueryRunQueryTask attribute), 61, 145
QueryMode (class in luigi.contrib.bigquery), 58, 142
queue (luigi.contrib.spark.SparkJob attribute), 85, 170
queue (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168

R

raise_on_error (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
RangeBase (class in luigi.tools.range), 94, 179
RangeDaily (class in luigi.tools.range), 96, 181
RangeDailyBase (class in luigi.tools.range), 95, 180
RangeEvent (class in luigi.tools.range), 94, 178
RangeHourly (class in luigi.tools.range), 97, 181
RangeHourlyBase (class in luigi.tools.range), 96, 180
re_enable() (luigi.scheduler.SimpleTaskState method), 121
re_enable_task() (luigi.rpc.RemoteScheduler method), 116
re_enable_task() (luigi.scheduler.CentralPlannerScheduler method), 122
read() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 92, 177
read() (luigi.contrib.webhdfs.WebHdfsClient method), 93, 177
read() (luigi.format.NewlineWrapper method), 103
read() (luigi.s3.ReadableS3File method), 117
readable() (luigi.format.InputPipeProcessWrapper method), 103
readable() (luigi.format.OutputPipeProcessWrapper method), 103
readable() (luigi.s3.ReadableS3File method), 117
ReadableS3File (class in luigi.s3), 117
ReadableWebHdfsFile (class in luigi.contrib.webhdfs), 92, 177
reader() (luigi.contrib.hadoop.JobTask method), 70, 154
readlines() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 92, 177
RecentRunHandler (class in luigi.server), 122
record_task_history (luigi.scheduler.scheduler attribute), 119
recursive_listdir_cmd (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient attribute), 54, 138
recursive_listdir_cmd (luigi.contrib.hdfs.hadoopcli_clients.HdfsClientApac attribute), 55, 139
RedisTarget (class in luigi.contrib.redis_store), 77, 162
RedshiftManifestTask (class in luigi.contrib.redshift), 80, 164
RedshiftTarget (class in luigi.contrib.redshift), 78, 162
reducer (luigi.contrib.hadoop.JobTask attribute), 70, 154
reducers_max (luigi.contrib.hive.HiveQueryTask attribute), 73, 157
reflect (luigi.contrib.sqla.CopyToTable attribute), 90, 175
Register (class in luigi.task_register), 131
reload() (luigi.configuration.LuigiConfigParser class method), 98
relpath() (luigi.contrib.scalding.ScaldingJobTask method), 82, 167
RemoteContext (class in luigi.contrib.ssh), 91, 175
RemoteFileSystem (class in luigi.contrib.ftp), 65, 149
RemoteFileSystem (class in luigi.contrib.ssh), 91, 176
RemoteScheduler (class in luigi.rpc), 115
RemoteTarget (class in luigi.contrib.ftp), 65, 149
RemoteTarget (class in luigi.contrib.ssh), 92, 176
remove() (luigi.contrib.ftp.RemoteFileSystem method), 65, 149
remove() (luigi.contrib.gcs.GCSClient method), 66, 150
remove() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 51, 135
remove() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
remove() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClientCdh3 method), 55, 139
remove() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 56, 139
remove() (luigi.contrib.hdfs.target.HdfsTarget method), 57, 141
remove() (luigi.contrib.ssh.RemoteFileSystem method), 91, 176
remove() (luigi.contrib.webhdfs.WebHdfsClient method), 93, 177

remove() (luigi.file.LocalFileSystem method), 102
remove() (luigi.file.LocalTarget method), 103
remove() (luigi.mock.MockFileSystem method), 108
remove() (luigi.s3.S3Client method), 116
remove() (luigi.target.FileSystem method), 124
remove() (luigi.target.FileSystemTarget method), 125
remove_delay (luigi.scheduler.scheduler attribute), 119
rename() (luigi.contrib.gcs.GCSClient method), 66, 150
rename() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 51, 135
rename() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 54, 138
rename() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 55, 139
rename() (luigi.contrib.hdfs.target.HdfsTarget method), 57, 141
rename() (luigi.mock.MockTarget method), 109
rename() (luigi.s3.S3Client method), 116
rename_dont_move() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 51, 135
rename_dont_move() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 55, 139
requires (class in luigi.util), 132
requires() (luigi.contrib.scalding.ScaldingJobTask method), 83, 167
requires() (luigi.task.Task method), 128
requires() (luigi.tools.range.RangeBase method), 95, 180
requires_hadoop() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
requires_hadoop() (luigi.contrib.spark.SparkJob method), 85, 170
requires_local() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
requires_local() (luigi.contrib.spark.SparkJob method), 85, 170
reset_global() (luigi.parameter.Parameter method), 111
resources (luigi.task.Task attribute), 126
retry_delay (luigi.scheduler.scheduler attribute), 119
retry_external_tasks (luigi.worker.worker attribute), 134
reverse (luigi.tools.range.RangeBase attribute), 95, 179
RootPathHandler (class in luigi.server), 123
rows() (luigi.contrib.sqla.CopyToTable method), 90, 175
rows() (luigi.postgres.CopyToTable method), 114
RPCError, 115
RPCHandler (class in luigi.server), 122
run (luigi.task.ExternalTask attribute), 129
run() (in module luigi.interface), 107
run() (in module luigi.server), 123
run() (luigi.contrib.bigquery.BigqueryLoadTask method), 61, 145
run() (luigi.contrib.bigquery.BigqueryRunQueryTask method), 61, 145
run() (luigi.contrib.esindex.CopyToIndex method), 64, 148
run() (luigi.contrib.hadoop.BaseHadoopJobTask method), 68, 152
run() (luigi.contrib.pig.PigJobTask method), 76, 160
run() (luigi.contrib.pyspark_runner.PySparkRunner method), 76, 160
run() (luigi.contrib.redshift.KillOpenRedshiftSessions method), 81, 165
run() (luigi.contrib.redshift.RedshiftManifestTask method), 80, 165
run() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
run() (luigi.contrib.spark.PySpark1xJob method), 87, 171
run() (luigi.contrib.spark.PySparkTask method), 85, 169
run() (luigi.contrib.spark.Spark1xJob method), 86, 171
run() (luigi.contrib.spark.SparkJob method), 85, 170
run() (luigi.contrib.spark.SparkSubmitTask method), 84, 169
run() (luigi.contrib.sparkey.SparkeyExportTask method), 87, 171
run() (luigi.contrib.sqla.CopyToTable method), 90, 175
run() (luigi.interface.Interface static method), 106
run() (luigi.mrrunner.Runner method), 109
run() (luigi.postgres.CopyToTable method), 115
run() (luigi.task.Task method), 128
run() (luigi.worker.KeepAliveThread method), 134
run() (luigi.worker.TaskProcess method), 133
run() (luigi.worker.Worker method), 135
run_and_track_hadoop_job() (in module luigi.contrib.hadoop), 71, 155
run_combiner() (luigi.contrib.hadoop.JobTask method), 70, 154
run_hive() (in module luigi.contrib.hive), 72, 156
run_hive_cmd() (in module luigi.contrib.hive), 72, 156
run_hive_script() (in module luigi.contrib.hive), 72, 156
run_job (luigi.contrib.hadoop.JobRunner attribute), 68, 153
run_job() (luigi.contrib.bigquery.BigqueryClient method), 60, 144
run_job() (luigi.contrib.hadoop.HadoopJobRunner method), 68, 152
run_job() (luigi.contrib.hadoop.LocalJobRunner method), 70, 154
run_job() (luigi.contrib.hadoop_jar.HadoopJarJobRunner method), 71, 155
run_job() (luigi.contrib.hive.HiveQueryRunner method), 74, 158
run_job() (luigi.contrib.scalding.ScaldingJobRunner method), 82, 166
run_mapper() (luigi.contrib.hadoop.JobTask method), 70, 154
run_reducer() (luigi.contrib.hadoop.JobTask method), 70, 154
Runner (class in luigi.mrrunner), 109

S

s3_load_path (luigi.contrib.redshift.S3CopyToTable attribute), 78, 163
S3Client (class in luigi.s3), 116
S3CopyJSONToTable (class in luigi.contrib.redshift), 79, 164
S3CopyToTable (class in luigi.contrib.redshift), 78, 162
S3EmrTarget (class in luigi.s3), 118
S3EmrTask (class in luigi.s3), 118
S3FlagTarget (class in luigi.s3), 117
S3FlagTask (class in luigi.s3), 118
S3PathTask (class in luigi.s3), 118
S3Target (class in luigi.s3), 117
sample() (luigi.contrib.hadoop.LocalJobRunner method), 70, 154
ScaldingJobRunner (class in luigi.contrib.scalding), 82, 166
ScaldingJobTask (class in luigi.contrib.scalding), 82, 166
Scheduler (class in luigi.scheduler), 119
scheduler (class in luigi.scheduler), 119
scheduler_host (luigi.interface.core attribute), 106
scheduler_port (luigi.interface.core attribute), 106
schema (luigi.contrib.bigquery.BigqueryLoadTask attribute), 60, 144
seekable() (luigi.format.InputPipeProcessWrapper method), 103
seekable() (luigi.format.OutputPipeProcessWrapper method), 103
seekable() (luigi.s3.ReadableS3File method), 117
SelectedRunHandler (class in luigi.server), 122
send_email() (in module luigi.notifications), 110
send_email_sendgrid() (in module luigi.notifications), 110
send_email_ses() (in module luigi.notifications), 110
send_email_smtp() (in module luigi.notifications), 110
send_error_email() (in module luigi.notifications), 110
separator (luigi.contrib.sparkey.SparkeyExportTask attribute), 87, 171
serialize (luigi.contrib.hadoop.JobTask attribute), 70, 154
serialize() (luigi.parameter.DateHourParameter method), 112
serialize() (luigi.parameter.Parameter method), 111
serialize_to_input() (luigi.parameter.Parameter method), 112
set() (luigi.configuration.LuigiConfigParser method), 98
set_global() (luigi.parameter.Parameter method), 111
set_global_from_args() (luigi.parameter.Parameter method), 112
set_global_parameters() (in module luigi.interface), 106
set_status() (luigi.scheduler.SimpleTaskState method), 121
settings (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
setup() (luigi.contrib.spark.PySparkTask method), 85, 169
setup_interface_logging() (in module luigi.interface), 105
setup_remote() (luigi.contrib.spark.PySparkTask method), 85, 169
SimpleTaskState (class in luigi.scheduler), 120
SingleProcessPool (class in luigi.worker), 133
snakebite_autoconfig (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
SnakebiteHdfsClient (class in luigi.contrib.hdfs.snakebite_client), 55, 139
source() (luigi.contrib.scalding.ScaldingJobTask method), 82, 167
source_format (luigi.contrib.bigquery.BigqueryLoadTask attribute), 60, 144
source_uris (luigi.contrib.bigquery.BigqueryLoadTask attribute), 61, 145
SourceFormat (class in luigi.contrib.bigquery), 58, 142
Spark1xBackwardCompat (class in luigi.contrib.spark), 86, 170
Spark1xJob (class in luigi.contrib.spark), 86, 170
spark_command() (luigi.contrib.spark.Spark1xBackwardCompat method), 86, 170
spark_command() (luigi.contrib.spark.SparkSubmitTask method), 84, 168
spark_master_memory (luigi.contrib.spark.SparkJob attribute), 85, 170
spark_options() (luigi.contrib.spark.Spark1xBackwardCompat method), 86, 170
spark_submit (luigi.contrib.spark.SparkSubmitTask attribute), 83, 168
spark_worker_memory (luigi.contrib.spark.SparkJob attribute), 85, 170
spark_workers (luigi.contrib.spark.SparkJob attribute), 85, 170
SparkeyExportTask (class in luigi.contrib.sparkey), 87, 171
SparkJob (class in luigi.contrib.spark), 85, 169
SparkJobError, 83, 167
SparkRunContext (class in luigi.contrib.spark), 83, 167
SparkSubmitTask (class in luigi.contrib.spark), 83, 167
SQLAlchemyTarget (class in luigi.contrib.sqla), 89, 173
SQLAlchemyTarget.Connection (class in luigi.contrib.sqla), 89, 174
ssh() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 72, 156
START (luigi.event.Event attribute), 102
start (luigi.tools.range.RangeBase attribute), 94, 179
start (luigi.tools.range.RangeDailyBase attribute), 95, 180
start (luigi.tools.range.RangeHourlyBase attribute), 96, 181
state_path (luigi.scheduler.scheduler attribute), 119
StaticFileHandler (class in luigi.server), 123
status_search() (luigi.tools.luigi_grep.LuigiGrep

method), 93, 178
stop (luigi.tools.range.RangeBase attribute), 94, 179
stop (luigi.tools.range.RangeDailyBase attribute), 95, 180
stop (luigi.tools.range.RangeHourlyBase attribute), 96, 181
stop() (in module luigi.server), 123
stop() (luigi.worker.KeepAliveThread method), 134
stop() (luigi.worker.Worker method), 134
SUCCESS (luigi.event.Event attribute), 102
supervise (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168

T

table (luigi.contrib.hive.ExternalHiveTask attribute), 74, 158
table (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
table (luigi.contrib.sqla.CopyToTable attribute), 90, 174
table_attributes() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
table_exists() (luigi.contrib.bigquery.BigqueryClient method), 59, 143
table_exists() (luigi.contrib.hive.HiveClient method), 72, 156
table_exists() (luigi.contrib.hive.HiveCommandClient method), 72, 157
table_exists() (luigi.contrib.hive.MetastoreClient method), 73, 157
table_location() (luigi.contrib.hive.HiveClient method), 72, 156
table_location() (luigi.contrib.hive.HiveCommandClient method), 72, 157
table_location() (luigi.contrib.hive.MetastoreClient method), 73, 157
table_schema() (luigi.contrib.hive.ApacheHiveCommandClient method), 73, 157
table_schema() (luigi.contrib.hive.HiveClient method), 72, 156
table_schema() (luigi.contrib.hive.HiveCommandClient method), 72, 157
table_schema() (luigi.contrib.hive.MetastoreClient method), 73, 157
Target (class in luigi.target), 123
Task (class in luigi.scheduler), 120
Task (class in luigi.task), 126
Task (class in luigi.task_history), 130
task_family (luigi.task.Task attribute), 127
task_family (luigi.task_register.Register attribute), 131
task_finished() (luigi.db_task_history.DbTaskHistory method), 100
task_finished() (luigi.task_history.NopHistory method), 130
task_finished() (luigi.task_history.TaskHistory method), 130
task_history (luigi.scheduler.CentralPlannerScheduler attribute), 122
task_id (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 68, 152
task_id (luigi.db_task_history.TaskEvent attribute), 101
task_id (luigi.db_task_history.TaskParameter attribute), 101
task_limit (luigi.tools.range.RangeBase attribute), 95, 179
task_limit (luigi.worker.worker attribute), 134
task_list() (luigi.rpc.RemoteScheduler method), 116
task_list() (luigi.scheduler.CentralPlannerScheduler method), 122
task_module (luigi.task.Task attribute), 127
task_names() (luigi.task_register.Register class method), 131
task_namespace (luigi.contrib.bigquery.BigqueryLoadTask attribute), 61, 145
task_namespace (luigi.contrib.bigquery.BigqueryRunQueryTask attribute), 61, 145
task_namespace (luigi.contrib.esindex.CopyToIndex attribute), 65, 149
task_namespace (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 68, 152
task_namespace (luigi.contrib.hadoop.hadoop attribute), 71, 155
task_namespace (luigi.contrib.hadoop.JobTask attribute), 70, 154
task_namespace (luigi.contrib.hadoop_jar.HadoopJarJobTask attribute), 72, 156
task_namespace (luigi.contrib.hdfs.config.hadoopcli attribute), 53, 136
task_namespace (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
task_namespace (luigi.contrib.hive.ExternalHiveTask attribute), 74, 159
task_namespace (luigi.contrib.hive.HiveQueryTask attribute), 74, 158
task_namespace (luigi.contrib.pig.PigJobTask attribute), 76, 160
task_namespace (luigi.contrib.rdbms.CopyToTable attribute), 77, 162
task_namespace (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 81, 166
task_namespace (luigi.contrib.redshift.RedshiftManifestTask attribute), 81, 165
task_namespace (luigi.contrib.redshift.S3CopyJSONToTable attribute), 80, 164
task_namespace (luigi.contrib.redshift.S3CopyToTable attribute), 79, 164
task_namespace (luigi.contrib.scalding.ScaldingJobTask attribute), 83, 167
task_namespace (luigi.contrib.spark.PySpark1xJob attribute), 87, 171

task_namespace (luigi.contrib.spark.PySparkTask attribute), 85, 169
task_namespace (luigi.contrib.spark.Spark1xBackwardCompat attribute), 86, 170
task_namespace (luigi.contrib.spark.Spark1xJob attribute), 86, 171
task_namespace (luigi.contrib.spark.SparkJob attribute), 86, 170
task_namespace (luigi.contrib.spark.SparkSubmitTask attribute), 84, 169
task_namespace (luigi.contrib.sparkey.SparkeyExportTask attribute), 87, 171
task_namespace (luigi.contrib.sqla.CopyToTable attribute), 91, 175
task_namespace (luigi.interface.core attribute), 106
task_namespace (luigi.postgres.CopyToTable attribute), 115
task_namespace (luigi.s3.S3EmrTask attribute), 118
task_namespace (luigi.s3.S3FlagTask attribute), 118
task_namespace (luigi.s3.S3PathTask attribute), 118
task_namespace (luigi.scheduler.scheduler attribute), 119
task_namespace (luigi.task.Config attribute), 130
task_namespace (luigi.task.ExternalTask attribute), 129
task_namespace (luigi.task.Task attribute), 129
task_namespace (luigi.task.WrapperTask attribute), 129
task_namespace (luigi.tools.deps.upstream attribute), 93, 178
task_namespace (luigi.tools.range.RangeBase attribute), 95, 180
task_namespace (luigi.tools.range.RangeDaily attribute), 97, 181
task_namespace (luigi.tools.range.RangeDailyBase attribute), 96, 180
task_namespace (luigi.tools.range.RangeHourly attribute), 97, 182
task_namespace (luigi.tools.range.RangeHourlyBase attribute), 96, 181
task_namespace (luigi.worker.worker attribute), 134
task_scheduled() (luigi.db_task_history.DbTaskHistory method), 100
task_scheduled() (luigi.task_history.NopHistory method), 130
task_scheduled() (luigi.task_history.TaskHistory method), 130
task_search() (luigi.rpc.RemoteScheduler method), 116
task_search() (luigi.scheduler.CentralPlannerScheduler method), 122
task_started() (luigi.db_task_history.DbTaskHistory method), 100
task_started() (luigi.task_history.NopHistory method), 130
task_started() (luigi.task_history.TaskHistory method), 130
task_value() (luigi.parameter.Parameter method), 111
task_wraps() (in module luigi.util), 132
TaskClassException, 131
TaskEvent (class in luigi.db_task_history), 101
TaskException, 133
TaskHistory (class in luigi.task_history), 130
TaskParameter (class in luigi.db_task_history), 100
TaskProcess (class in luigi.worker), 133
TaskRecord (class in luigi.db_task_history), 101
tasks_str() (luigi.task_register.Register class method), 131
temp_hadoop_output_file (luigi.contrib.spark.SparkJob attribute), 85, 170
text_target (luigi.contrib.redshift.RedshiftManifestTask attribute), 80, 165
TextFormat (class in luigi.format), 104
TextWrapper (class in luigi.format), 104
TimeDeltaParameter (class in luigi.parameter), 113
timeout (luigi.contrib.esindex.CopyToIndex attribute), 64, 148
timeout (luigi.worker.worker attribute), 134
tmp_dir (luigi.contrib.hdfs.config.hdfs attribute), 52, 136
tmp_path (luigi.contrib.ssh.AtomicRemoteFileWriter attribute), 92, 176
tmp_path (luigi.target.AtomicLocalFile attribute), 125
tmppath() (in module luigi.contrib.hdfs.config), 53, 137
to_str_params() (luigi.task.Task method), 127
to_string() (luigi.date_interval.Custom method), 100
to_string() (luigi.date_interval.Date method), 99
to_string() (luigi.date_interval.DateInterval method), 99
to_string() (luigi.date_interval.Month method), 99
to_string() (luigi.date_interval.Week method), 99
to_string() (luigi.date_interval.Year method), 100
total_executor_cores (luigi.contrib.spark.SparkSubmitTask attribute), 84, 168
touch() (luigi.contrib.esindex.ElasticsearchTarget method), 62, 146
touch() (luigi.contrib.mysqldb.MySqlTarget method), 75, 159
touch() (luigi.contrib.redis_store.RedisTarget method), 78, 162
touch() (luigi.contrib.sqla.SQLAlchemyTarget method), 89, 174
touch() (luigi.postgres.PostgresTarget method), 114
touchz() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 52, 136
touchz() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 55, 138
touchz() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 57, 141
TracebackWrapper (class in luigi.worker), 133
track_and_progress() (luigi.contrib.pig.PigJobTask method), 76, 160
track_progress() (luigi.contrib.spark.SparkJob method), 85, 170

trigger_event() (luigi.task.Task method), 127
truncate_table() (luigi.contrib.redshift.S3CopyToTable method), 79, 163
ts (luigi.db_task_history.TaskEvent attribute), 101
tunnel() (luigi.contrib.ssh.RemoteContext method), 91, 176

U

UnknownConfigException, 110
UnknownParameterException, 110
update() (luigi.scheduler.CentralPlannerScheduler method), 121
update() (luigi.scheduler.Worker method), 120
update_id() (luigi.contrib.esindex.CopyToIndex method), 64, 148
update_id() (luigi.contrib.rdbms.CopyToTable method), 77, 161
update_id() (luigi.contrib.redshift.KillOpenRedshiftSessions method), 81, 165
update_id() (luigi.contrib.sqla.CopyToTable method), 90, 175
update_resources() (luigi.rpc.RemoteScheduler method), 116
update_resources() (luigi.scheduler.CentralPlannerScheduler method), 121
upload() (luigi.contrib.webhdfs.WebHdfsClient method), 92, 177
upstream (class in luigi.tools.deps), 93, 177
UPSTREAM_SEVERITY_KEY() (in module luigi.scheduler), 119
use_cmdline_section (luigi.interface.core attribute), 106
use_cmdline_section (luigi.task.Task attribute), 127
use_db_timestamps (luigi.contrib.redshift.RedshiftTarget attribute), 78, 162
use_db_timestamps (luigi.postgres.PostgresTarget attribute), 114
user (luigi.contrib.rdbms.CopyToTable attribute), 77, 161
user (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 81, 165

V

value (luigi.db_task_history.TaskParameter attribute), 101
value (luigi.parameter.Parameter attribute), 111
version (luigi.contrib.hdfs.config.hadoopcli attribute), 53, 136
visualization_graph (luigi.scheduler.scheduler attribute), 119

W

wait_interval (luigi.worker.worker attribute), 134
walk() (luigi.contrib.webhdfs.WebHdfsClient method), 92, 177
WebHdfsClient (class in luigi.contrib.webhdfs), 92, 177
WebHdfsTarget (class in luigi.contrib.webhdfs), 92, 177
Week (class in luigi.date_interval), 99
Worker (class in luigi.scheduler), 120
Worker (class in luigi.worker), 134
worker (class in luigi.worker), 134
worker_disconnect_delay (luigi.scheduler.scheduler attribute), 119
worker_list() (luigi.rpc.RemoteScheduler method), 116
worker_list() (luigi.scheduler.CentralPlannerScheduler method), 122
worker_timeout (luigi.task.Task attribute), 127
workers (luigi.interface.core attribute), 106
WorkerSchedulerFactory (class in luigi.interface), 106
wrap_traceback() (in module luigi.notifications), 109
WrappedFormat (class in luigi.format), 104
wrapper_cls (luigi.format.MixedUnicodeBytesFormat attribute), 104
wrapper_cls (luigi.format.NewlineFormat attribute), 104
wrapper_cls (luigi.format.TextFormat attribute), 104
WrapperTask (class in luigi.task), 129
writable() (luigi.format.InputPipeProcessWrapper method), 103
writable() (luigi.format.OutputPipeProcessWrapper method), 103
writable() (luigi.s3.ReadableS3File method), 117
write() (luigi.format.MixedUnicodeBytesWrapper method), 104
write() (luigi.format.NewlineWrapper method), 103
write() (luigi.format.OutputPipeProcessWrapper method), 103
WRITE_APPEND (luigi.contrib.bigquery.WriteDisposition attribute), 58, 142
write_disposition (luigi.contrib.bigquery.BigqueryLoadTask attribute), 60, 144
write_disposition (luigi.contrib.bigquery.BigqueryRunQueryTask attribute), 61, 145
WRITE_EMPTY (luigi.contrib.bigquery.WriteDisposition attribute), 58, 142
write_pid() (in module luigi.process), 115
WRITE_TRUNCATE (luigi.contrib.bigquery.WriteDisposition attribute), 58, 142
WriteDisposition (class in luigi.contrib.bigquery), 58, 142
writeLine() (luigi.format.OutputPipeProcessWrapper method), 103
writelines() (luigi.format.MixedUnicodeBytesWrapper method), 104
writelines() (luigi.format.NewlineWrapper method), 103
writer() (luigi.contrib.hadoop.JobTask method), 70, 154
WRITES_BEFORE_FLUSH (luigi.format.OutputPipeProcessWrapper attribute), 103

Y

Year (class in luigi.date_interval), 99