Contents

1 Background 3
2 Visualiser page 5
3 Dependency graph example 7
4 Philosophy 9
5 Who uses Luigi? 11
6 External links 15
7 Authors 17
8 Table of Contents 19
  8.1 Example – Top Artists .......................... 19
  8.2 Building workflows ...................................... 23
  8.3 Tasks ........................................ 28
  8.4 Parameters ....................................... 33
  8.5 Running Luigi .................................... 36
  8.6 Using the Central Scheduler ...................... 38
  8.7 Execution Model .................................. 41
  8.8 Luigi Patterns .................................. 43
  8.9 Configuration .................................. 48
  8.10 Configure logging .......................... 60
  8.11 Design and limitations ........................ 61
9 API Reference 63
  9.1 luigi package .................................. 63
  9.2 Indices and tables .................................. 248
Python Module Index 249
Index 251
Luigi is a Python (2.7, 3.6, 3.7 tested) 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. Documentation for the latest release is hosted on readthedocs.

Run `pip install luigi[toml]` to install Luigi with TOML-based configs support.

For the bleeding edge code, `pip install git+https://github.com/spotify/luigi.git`. Bleeding edge documentation is also available.
CHAPTER 1

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.
The Luigi server comes with a web interface too, so you can search and filter among all your tasks.
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 visualiser, 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.
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.
Who uses Luigi?

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.

Since Luigi is open source and without any registration walls, the exact number of Luigi users is unknown. But based on the number of unique contributors, we expect hundreds of enterprises to use it. Some users have written blog posts or held presentations about Luigi:

- Spotify (presentation, 2014)
- Foursquare (presentation, 2013)
- Mortar Data (Datadog) (documentation / tutorial)
- Stripe (presentation, 2014)
- Buffer (blog, 2014)
- SeatGeek (blog, 2015)
- Treasure Data (blog, 2015)
- Growth Intelligence (presentation, 2015)
- AdRoll (blog, 2015)
- 17zuoye (presentation, 2015)
- Custobar (presentation, 2016)
- Blendle (presentation)
- TrustYou (presentation, 2015)
- Groupon / OrderUp (alternative implementation)
- GetNinjas (blog, 2017)
- voyages-sncf.com (presentation, 2017)
Some more companies are using Luigi but haven’t had a chance yet to write about it:

- Schibsted
- enbrite.ly
- Dow Jones / The Wall Street Journal
- Hotels.com
- Newsea
- Squarespace
- OAO
- Grovo
- Weebly
- Deloitte
- Stacktome
- LINX+Neemu+Chaordic
- Foxberry
- Okko
- ISVWorld
- Big Data
- Movio
- Bonnier News
- Starsky Robotics
- BaseTIS
- Hopper
- VOYAGE GROUP/Zucks
- Textpert
- Whizar
- xtream
- Skyscanner
• Jodel
• Mekar
• M3

We’re more than happy to have your company added here. Just send a PR on GitHub.
External links

- Mailing List for discussions and asking questions. (Google Groups)
- Releases (PyPI)
- Source code (GitHub)
- Hubot Integration plugin for Slack, Hipchat, etc (GitHub)
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.
8.1 Example – Top Artists

This is a very simplified case of something we do at Spotify a lot. All user actions are logged to Google Cloud Storage (previously HDFS) where we run a bunch of processing jobs to transform the data. The processing code itself is implemented in a scalable data processing framework, such as Scio, Scalding, or Spark, but the jobs are orchestrated with Luigi. At some point we might end up with a smaller data set that we can bulk ingest into Cassandra, Postgres, or other storage suitable for serving or exploration.

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.

8.1.1 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

(continues on next page)
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.

• LocalTarget is a built in class that makes it easy to read/write from/to the local filesystem. It also makes all file operations atomic, which is nice in case your script crashes for any reason.

8.1.2 Running this Locally

Try running this using eg.

```
$ cd examples
$ luigi --module top_artists AggregateArtists --local-scheduler --date-interval 2012-06
```

Note that top_artists needs to be in your PYTHONPATH, or else this can produce an error (ImportError: No module named top_artists). Add the current working directory to the command PYTHONPATH with:

```
$ PYTHONPATH=./ luigi --module top_artists 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.

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.

8.1.3 Step 1b - Aggregate artists with Spark

While Luigi can process data inline, it is normally used to orchestrate external programs that perform the actual processing. In this example, we will demonstrate how top artists instead can be read from HDFS and calculated with Spark, orchestrated by Luigi.
class AggregateArtistsSpark(luigi.contrib.spark.SparkSubmitTask):
    date_interval = luigi.DateIntervalParameter()

    app = 'top_artists_spark.py'
    master = 'local[*]'

    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 app_options(self):
        # :func:`~luigi.task.Task.input` returns the targets produced by the tasks in
        # `~luigi.task.Task.requires`.
        return [','.join([p.path for p in self.input()]),
                self.output().path]

luigi.contrib.hadoop.SparkSubmitTask doesn’t require you to implement a run() method. Instead,
you specify the command line parameters to send to spark-submit, as well as any other configuration specific to
Spark.

Python code for the Spark job is found below.

```python
import operator
import os
from pyspark.sql import SparkSession

def main(argv):
    input_paths = argv[1].split(',')
    output_path = argv[2]

    spark = SparkSession.builder.getOrCreate()

    streams = spark.read.option('sep', '\t').csv(input_paths[0])
    for stream_path in input_paths[1:]:
        streams.union(spark.read.option('sep', '\t').csv(stream_path))

    # The second field is the artist
    counts = streams.map(lambda row: (row[1], 1))
                   .reduceByKey(operator.add)
    counts.write.option('sep', '\t').csv(output_path)
```

In a typical deployment scenario, the Luigi orchestration definition above as well as the Pyspark processing code
would be packaged into a deployment package, such as a container image. The processing code does not have to be
implemented in Python, any program can be packaged in the image and run from Luigi.
8.1.4 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.

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

    def requires(self):
        if self.use_hadoop:
            return AggregateArtistsSpark(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, streams

    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.

```bash
$ luigi --module examples.top_artists Top10Artists --local-scheduler --date-interval 2012-07
```

This will run both tasks.

8.1.5 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.

```python
class ArtistToplistToDatabase(luigi.contrib.postgres.CopyToTable):
    date_interval = luigi.DateIntervalParameter()
    use_hadoop = luigi.BoolParameter()
    host = "localhost"
    database = "toplists"
    user = "luigi"
    password = "abc123" ;)
    table = "top10"
```

(continues on next page)
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.

### 8.1.6 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 workflow.

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

```bash
$ luigid
```

in the background and then run your task without the `--local-scheduler` flag, then your script will now schedule through a centralized server. You need Tornado for this to work.

Launching [http://localhost:8082](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.

### 8.2 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.

#### 8.2.1 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 with a toolbox of several useful Targets. In particular, `LocalTarget` and `HdfsTarget`, but there is also support for other file systems: `luigi.contrib.s3.S3Target`, `luigi.contrib.ssh.RemoteTarget`, `luigi.contrib.ftp.RemoteTarget`, `luigi.contrib.mysqldb.MySqlTarget`, `luigi.contrib.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.

8.2.2 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.

![Diagram](image1)

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.

![Diagram](image2)

### 8.2.3 Parameter

The Task class corresponds to some type of job that is run, but in general you want to allow some form of parameterization 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.

![Diagram](image3)

### 8.2.4 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)
8.2. Building workflows
8.3 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:

```python
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/luigi/outputs/%s.txt' % self.param)

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

### 8.3.1 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

```python
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.

### 8.3.2 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:
This also makes it easier to add parameters:

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

### 8.3.3 Task.output

The `output()` method returns one or more `Target` objects. Similarly to `requires`, you can return them wrapped 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.)

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

### 8.3.4 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:

```python
class GenerateWords(luigi.Task):
    def output(self):
        return luigi.LocalTarget('words.txt')
    def run(self):
        # write a dummy list of words to output file
        words = ['apple', 'banana', 'grapefruit']
        with self.output().open('w') as f:
            for word in words:
                f.write('{word}
'.format(word=word))

class CountLetters(luigi.Task):
    def requires(self):
```

(continues on next page)
return GenerateWords()

def output(self):
    return luigi.LocalTarget('letter_counts.txt')

def run(self):
    # read in file as list
    with self.input().open('r') as infile:
        words = infile.read().splitlines()

    # write each word to output file with its corresponding letter count
    with self.output().open('w') as outfile:
        for word in words:
            outfile.write('{word} | {letter_count}
'.format(word=word,
                    letter_count=len(word))

It's useful to note that if you're writing to a binary file, Luigi automatically strips the 'b' flag due to how atomic writes/reads work. In order to write a binary file, such as a pickle file, you should instead use format=Nop when calling LocalTarget. Following the above example:

class GenerateWords(luigi.Task):
    def output(self):
        return luigi.LocalTarget('words.pckl', format=Nop)

    def run(self):
        import pickle

        # write a dummy list of words to output file
        words = ['apple',
                 'banana',
                 'grapefruit']

        with self.output().open('w') as f:
            pickle.dump(words, f)

8.3.5 Task.input

As seen in the example above, input() 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)]}
8.3.6 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 yield a list of tasks.

```python
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.

8.3.7 Task status tracking

For long-running or remote tasks it is convenient to see extended status information not only on the command line or in your logs but also in the GUI of the central scheduler. Luigi implements dynamic status messages, progress bar and tracking urls which may point to an external monitoring system. You can set this information using callbacks within Task.run:

```python
class MyTask(luigi.Task):
    def run(self):
        # set a tracking url
        self.set_tracking_url("http://...")
        # set status messages during the workload
        for i in range(100):
            # do some hard work here
            if i % 10 == 0:
                self.set_status_message("Progress: %d / 100" % i)
                # displays a progress bar in the scheduler UI
                self.set_progress_percentage(i)
```

8.3.8 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()

8.3.9 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.target.HdfsTarget class anywhere by just instantiating it:

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

8.3.10 Task priority

The scheduler decides which task to run next from the set of all tasks 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.

### 8.3.11 Namespaces, families and ids

In order to avoid name clashes and to be able to have an identifier for tasks, Luigi introduces the concepts `task_namespace`, `task_family` and `task_id`. The namespace and family operate on class level meanwhile the task id only exists on instance level. The concepts are best illustrated using code.

```python
import luigi
class MyTask(luigi.Task):
    my_param = luigi.Parameter()
    task_namespace = 'my_namespace'

my_task = MyTask(my_param='hello')
print(my_task)  # --> my_namespace.MyTask(my_param=hello)
print(my_task.get_task_namespace())  # --> my_namespace
print(my_task.get_task_family())  # --> my_namespace.MyTask
print(my_task.task_id)  # --> my_namespace.MyTask_hello_890907e7ce

print(MyTask.get_task_namespace())  # --> my_namespace
print(MyTask.get_task_family())  # --> my_namespace.MyTask
print(MyTask.task_id)  # --> Error!
```

The full documentation for this machinery exists in the `task` module.

### 8.3.12 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.

### 8.4 Parameters

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

```python
    date = luigi.DateParameter(default=datetime.date.today())
    # ...
```

By doing this, Luigi can 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 2012-05-10`.

The parameters are all set to their values on the Task object instance, i.e.
Luigi Documentation, Release 2.8.13

```python
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.

### 8.4.1 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:

```python
>>> 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 is d
True
```

### 8.4.2 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:

```python
>>> class DateTask2(DateTask):
...    other = luigi.Parameter(significant=False)
...
>>> c = DateTask2(date=a, other="foo")
>>> d = DateTask2(date=b, other="bar")
>>> c is d
False
>>> hash(c) == hash(d)
True
```
8.4.3 Parameter visibility

Using `ParameterVisibility` you can configure parameter visibility. By default, all parameters are public, but you can also set them hidden or private.

```python
>>> import luigi
>>> from luigi.parameter import ParameterVisibility

>>> luigi.Parameter(visibility=ParameterVisibility.PRIVATE)

ParameterVisibility.PUBLIC (default) - visible everywhere
ParameterVisibility.HIDDEN - ignored in WEB-view, but saved into database if save db_history is true
ParameterVisibility.PRIVATE - visible only inside task.
```

8.4.4 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 statically 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).

8.4.5 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`:

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

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

You can run `TaskB` on the command line: `luigi TaskB --y 42`. But you can also set the class value of `TaskA` by running `luigi 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)`.

8.4.6 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.

8.5 Running Luigi

8.5.1 Running from the Command Line

The preferred way to run Luigi tasks is through the luigi command line tool that will be installed with the pip package.

```python
# my_module.py, available in your sys.path
import luigi

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

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

Should be run like this

```bash
$ luigi --module my_module MyTask --x 123 --y 456 --local-scheduler
```

Or alternatively like this:

```bash
$ python -m luigi --module my_module MyTask --x 100 --local-scheduler
```

Note that if a parameter name contains '_', it should be replaced by '-' For example, if MyTask had a parameter called 'my_parameter':

```bash
$ luigi --module my_module MyTask --my-parameter 100 --local-scheduler
```

Note: Please make sure to always place task parameters behind the task family!

8.5.2 Running from Python code

Another way to start tasks from Python code is using `luigi.build(tasks, worker_scheduler_factory=None, **env_params)` from luigi.interface module.

This way of running luigi tasks is useful if you want to get some dynamic parameters from another source, such as database, or provide additional logic before you start tasks.

One notable difference is that `build` defaults to not using the identical process lock. If you want to change this behaviour, just pass `no_lock=False`.
class MyTask1(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=0)

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

class MyTask2(luigi.Task):
    x = luigi.IntParameter()
    y = luigi.IntParameter(default=1)
    z = luigi.IntParameter(default=2)

    def run(self):
        print(self.x * self.y * self.z)

if __name__ == '__main__':
    luigi.build([MyTask1(x=10), MyTask2(x=15, z=3)])

Also, it is possible to pass additional parameters to build such as host, port, workers and local_scheduler:

if __name__ == '__main__':
    luigi.build([MyTask1(x=1)], workers=5, local_scheduler=True)

To achieve some special requirements you can pass to build your worker_scheduler_factory which will return your worker and/or scheduler implementations:

class MyWorker(Worker):
    # some custom logic

class MyFactory(object):
    def create_local_scheduler(self):
        return scheduler.Scheduler(prune_on_get_work=True, record_task_history=False)

    def create_remote_scheduler(self, url):
        return rpc.RemoteScheduler(url)

    def create_worker(self, scheduler, worker_processes, assistant=False):
        return MyWorker(
            scheduler=scheduler, worker_processes=worker_processes, assistant=assistant)

if __name__ == '__main__':
    luigi.build([MyTask1(x=1)], worker_scheduler_factory=MyFactory())

In some cases (like task queue) it may be useful.

### 8.5.3 Response of luigi.build()/luigi.run()

- **Default response** By default `luigi.build()`/`luigi.run()` returns True if there were no scheduling errors. This is the same as the attribute `LuigiRunResult.scheduling_succeeded`.

- **Detailed response** This is a response of type `LuigiRunResult`. This is obtained by passing a keyword argument `detailed_summary=True` to `build/run`. This response contains detailed information about the
8.5.4 Luigi on Windows

Most Luigi functionality works on Windows. Exceptions:

- Specifying multiple worker processes using the `workers` argument for `luigi.build`, or using the `--workers` command line argument. (Similarly, specifying `--worker-force-multiprocessing`). For most programs, this will result in failure (a common sight is `BrokenPipeError`). The reason is that worker processes are assumed to be forked from the main process. Forking is not possible on Windows.

- Running the Luigi central scheduling server as a daemon (i.e. with `--background`). Again, a Unix-only concept.

8.6 Using the Central Scheduler

While the `--local-scheduler` flag is useful for development purposes, it’s not recommended for production usage. The centralized scheduler serves 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 jobs. This model may seem limited, but we believe that it makes things far more intuitive and easy to understand.
8.6.1 The luigid server

To run the server as a daemon run:

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

Note that this requires python-daemon. By default, the server starts on AF_INET and AF_INET6 port 8082 (which can be changed with the --port flag) and listens on all IPs. (To use an AF_UNIX socket use the --unix-socket flag)

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. luigi.cfg or /etc/luigi/client.cfg by default).

8.6.2 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 luigi.cfg and specify db_connection under [task_history]. The db_connection string is 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Last Action</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>None</td>
<td>2014-12-31 20:16:58.505362</td>
<td>DONE</td>
</tr>
<tr>
<td>WordCount</td>
<td>None</td>
<td>2014-12-31 20:16:56.602269</td>
<td>DONE</td>
</tr>
<tr>
<td>InputText</td>
<td>None</td>
<td>2014-12-31 20:16:52.233931</td>
<td>PENDING</td>
</tr>
<tr>
<td>WordCount</td>
<td>None</td>
<td>2014-12-31 20:16:52.210956</td>
<td>PENDING</td>
</tr>
</tbody>
</table>

- /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:
### Info

<table>
<thead>
<tr>
<th>Task Id</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Name</td>
<td>WordCount</td>
</tr>
<tr>
<td>Host</td>
<td>None</td>
</tr>
<tr>
<td>More</td>
<td>All &quot;WordCount&quot; runs</td>
</tr>
</tbody>
</table>

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>date_interval</td>
<td>2014-12-31</td>
</tr>
</tbody>
</table>

### Actions

<table>
<thead>
<tr>
<th>Status</th>
<th>Action Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONE</td>
<td>2014-12-31 20:16:58.505362</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Host</th>
<th>Last Action</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>None</td>
<td>2014-12-31 20:16:52.210956</td>
<td>PENDING</td>
</tr>
<tr>
<td>WordCount</td>
<td>None</td>
<td>2014-12-31 20:16:56.602269</td>
<td>DONE</td>
</tr>
<tr>
<td>WordCount</td>
<td>None</td>
<td>2014-12-31 20:16:58.505362</td>
<td>DONE</td>
</tr>
</tbody>
</table>
8.7 Execution Model

Luigi has a quite simple model for execution and triggering.

8.7.1 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 a centralized view of the dependency graph and that the same job will not be executed by multiple workers simultaneously.
8.7.2 Scheduler

A client only starts the `run()` method of a task when the single-threaded central scheduler has permitted it. Since the number of tasks is usually very small (in comparison with the petabytes of data one task is processing), we can afford the convenience of a simple centralised server.

The gif is from this presentation, which is about the client and server interaction.

8.7.3 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.

```python
# my_tasks.py

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

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.window)]

    def run(self):
        run_some_cool_stuff(self.input())

    def output(self):
        return luigi.contrib.hdfs.HdfsTarget('/aggregated-%s-%d' % (self.date, self.window))


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)

In your crontab you would then have something like

```
30 0 * * * my-user luigi RunAll --module my_tasks
```
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 give you some form of parallelization (e.g. `AggregationTask(2013-01-09)` might run in parallel with `AggregationTask(2013-01-08)`).

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

### 8.8 Luigi Patterns

#### 8.8.1 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.

#### 8.8.2 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.

```python
class AllReports(luigi.WrapperTask):
    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).

You’ll need to use `WrapperTask` for this instead of the usual Task class, because this job will not produce any output of its own, and as such needs a way to indicate when it’s complete. This class is used for tasks that only wrap other tasks and that by definition are done if all their requirements exist.

#### 8.8.3 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
8.8.4 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 contiguosness 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.

8.8.5 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
```

8.8.6 Propagating parameters with Range

Some tasks you want to recur may include additional parameters which need to be configured. The Range classes provide a parameter which accepts a DictParameter and passes any parameters onwards for this purpose.

```
luigi RangeDaily --of MyTask --start 2014-10-31 --of-params '"my_string_param": "123", "my_int_param": 123'
```

Alternatively, you can specify parameters at the task family level (as described here), however these will not appear in the task name for the upstream Range task which can have implications in how the scheduler and visualizer handle task instances.

```
luigi RangeDaily --of MyTask --start 2014-10-31 --MyTask-my-param 123
```

8.8.7 Batching multiple parameter values into a single run

Sometimes it’ll be faster to run multiple jobs together as a single batch rather than running them each individually. When this is the case, you can mark some parameters with a batch_method in their constructor to tell the worker how to combine multiple values. One common way to do this is by simply running the maximum value. This is good for tasks that overwrite older data when a newer one runs. You accomplish this by setting the batch_method to max, like so:
What’s exciting about this is that if you send multiple As to the scheduler, it can combine them and return one. So if A(date=2016-07-28), A(date=2016-07-29) and A(date=2016-07-30) are all ready to run, you will start running A(date=2016-07-30). While this is running, the scheduler will show A(date=2016-07-28), A(date=2016-07-29) as batch running while A(date=2016-07-30) is running. When A(date=2016-07-30) is done running and becomes FAILED or DONE, the other two tasks will be updated to the same status.

If you want to limit how big a batch can get, simply set max_batch_size. So if you have

```python
class A(luigi.Task):
    date = luigi.DateParameter(batch_method=max)
    max_batch_size = 10
```

then the scheduler will batch at most 10 jobs together. You probably do not want to do this with the max batch method, but it can be helpful if you use other methods. You can use any method that takes a list of parameter values and returns a single parameter value.

If you have two max batch parameters, you’ll get the max values for both of them. If you have parameters that don’t have a batch method, they’ll be aggregated separately. So if you have a class like

```python
class A(luigi.Task):
    p1 = luigi.IntParameter(batch_method=max)
    p2 = luigi.IntParameter(batch_method=max)
    p3 = luigi.IntParameter()
```

and you create tasks A(p1=1, p2=2, p3=0), A(p1=2, p2=3, p3=0), A(p1=3, p2=4, p3=1), you’ll get them batched as A(p1=2, p2=3, p3=0) and A(p1=3, p2=4, p3=1).

Note that batched tasks do not take up [resources], only the task that ends up running will use resources. The scheduler only checks that there are sufficient resources for each task individually before batching them all together.

### 8.8.8 Tasks that regularly overwrite the same data source

If you are overwriting of the same data source with every run, you’ll need to ensure that two batches can’t run at the same time. You can do this pretty easily by setting batch_method to max and setting a unique resource:

```python
class A(luigi.Task):
    date = luigi.DateParameter(batch_method=max)
    resources = {'overwrite_resource': 1}
```

Now if you have multiple tasks such as A(date=2016-06-01), A(date=2016-06-02), A(date=2016-06-03), the scheduler will just tell you to run the highest available one and mark the lower ones as batch_running. Using a unique resource will prevent multiple tasks from writing to the same location at the same time if a new one becomes available while others are running.

### 8.8.9 Avoiding concurrent writes to a single file

Updating a single file from several tasks is almost always a bad idea, and you need to be very confident that no other good solution exists before doing this. If, however, you have no other option, then you will probably at least need to ensure that no two tasks try to write to the file _simultaneously_.

---

**8.8. Luigi Patterns**

45
By turning ‘resources’ into a Python property, it can return a value dependent on the task parameters or other dynamic attributes:

```python
class A(luigi.Task):
    ...

    @property
def resources(self):
        return {self.important_file_name: 1}
```

Since, by default, resources have a usage limit of 1, no two instances of Task A will now run if they have the same `important_file_name` property.

### 8.8.10 Decreasing resources of running tasks

At scheduling time, the luigi scheduler needs to be aware of the maximum resource consumption a task might have once it runs. For some tasks, however, it can be beneficial to decrease the amount of consumed resources between two steps within their run method (e.g. after some heavy computation). In this case, a different task waiting for that particular resource can already be scheduled.

```python
class A(luigi.Task):
    # set maximum resources a priori
    resources = {"some_resource": 3}

def run(self):
    # do something
    ...

    # decrease consumption of "some_resource" by one
    self.decrease_running_resources({"some_resource": 1})

    # continue with reduced resources
    ...
```

### 8.8.11 Monitoring task pipelines

Luigi comes with some existing ways in `luigi.notifications` to receive notifications whenever tasks crash. Email is the most common way.

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.

### 8.8.12 Atomic Writes Problem

A very common mistake done by luigi plumbers is to write data partially to the final destination, that is, not atomically. The problem arises because completion checks in luigi are exactly as naive as running `luigi.target.Target.exists()`. And in many cases it just means to check if a folder exist on disk. During the time we have partially written data, a task depending on that output would think its input is complete. This can have devastating effects, as in the thanksgiving bug.

The concept can be illustrated by imagining that we deal with data stored on local disk and by running commands:
As stated earlier, the problem is that only partial data exists for a duration, yet we consider the data to be `complete()` because the output folder already exists. Here is a robust version of this:

```bash
# This is the good way
$ mkdir /outputs/final_output-tmp-123456
$ big-slow-calculation > /outputs/final_output-tmp-123456/foo.data
$ mv --no-target-directory --no-clobber /outputs/final_output{-tmp-123456,}
$ [[ -d /outputs/final_output-tmp-123456 ]] && rm -r /outputs/final_output-tmp-123456
```

Indeed, the good way is not as trivial. It involves coming up with a unique directory name and a pretty complex `mv` line, the reason `mv` need all those is because we don’t want `mv` to move a directory into a potentially existing directory. A directory could already exist in exceptional cases, for example when central locking fails and the same task would somehow run twice at the same time. Lastly, in the exceptional case where the file was never moved, one might want to remove the temporary directory that never got used.

Note that this was an example where the storage was on local disk. But for every storage (hard disk file, hdfs file, database table, etc.) this procedure will look different. But do every luigi user need to implement that complexity? Nope, thankfully luigi developers are aware of these and luigi comes with many built-in solutions. In the case of you’re dealing with a file system (`FileSystemTarget`), you should consider using `temporary_path()`. For other targets, you should ensure that the way you’re writing your final output directory is atomic.

### 8.8.13 Sending messages to tasks

The central scheduler is able to send messages to particular tasks. When a running task accepts messages, it can access a `multiprocessing.Queue` object storing incoming messages. You can implement custom behavior to react and respond to messages:

```python
class Example(luigi.Task):
    # common task setup
    ...

    # configure the task to accept all incoming messages
    accepts_messages = True

    def run(self):
        # this example runs some loop and listens for the
        # "terminate" message, and responds to all other messages
        for _ in some_loop():
            # check incoming messages
            if not self.scheduler_messages.empty():
                msg = self.scheduler_messages.get()
                if msg.content == "terminate":
                    break
                else:
                    msg.respond("unknown message")

        # finalize
        ...
```

Messages can be sent right from the scheduler UI which also displays responses (if any). Note that this feature is only available when the scheduler is configured to send messages (see the `[scheduler]` config), and the task is configured to
accept them.

8.9 Configuration

All configuration can be done by adding configuration files.

Supported config parsers:

- `cfg` (default), based on Python’s standard `ConfigParser`. Values may refer to environment variables using `${ENVVAR}` syntax.
- `toml`

You can choose right parser via `LUIGI_CONFIG_PARSER` environment variable. For example, `LUIGI_CONFIG_PARSER=toml`.

Default (cfg) parser are looked for in:

- `/etc/luigi/client.cfg` (deprecated)
- `/etc/luigi/luigi.cfg`
- `client.cfg` (deprecated)
- `luigi.cfg`
- `LUIGI_CONFIG_PATH` environment variable

TOML parser are looked for in:

- `/etc/luigi/luigi.toml`
- `luigi.toml`
- `LUIGI_CONFIG_PATH` environment variable

Both config lists increase in priority (from low to high). The order only matters in case of key conflicts (see docs for `ConfigParser.read`). These files are meant for both the client and `luigid`. If you decide to specify your own configuration you should make sure that both the client and `luigid` load it properly.

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

Example cfg config:

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

[core]
scheduler_host=luigi-host.mycompany.foo
```

Example toml config:

```
[hadoop]
version = "cdh4"
streaming_jar = "/usr/lib/hadoop-xyz/hadoop-streaming-xyz-123.jar"

[core]
scheduler_host = "luigi-host.mycompany.foo"
```

Also see examples/config.toml for more complex example.
8.9.1 Parameters from config Ingestion

All parameters can be overridden from configuration files. For instance if you have a Task definition:

```python
    date = luigi.DateParameter(default=datetime.date.today())
    # ...
```

Then you can override the default value for `DailyReport().date` by providing it in the configuration:

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

### Configuration classes

Using the `Parameters from config Ingestion` method, we derive the conventional way to do global configuration. Imagine this configuration.

```
[mysection]
option=hello
intoption=123
```

We can create a `Config` class:

```python
import luigi

# Config classes should be camel cased
class mysection(luigi.Config):
    option = luigi.Parameter(default='world')
    intoption = luigi.IntParameter(default=555)

mysection().option
mysection().intoption
```

8.9.2 Configurable options

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

8.9.3 [core]

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

- **autoload_range** New in version 2.8.11.
  
  If false, prevents range tasks from autoloading. They can still be loaded using `--module luigi.tools.range`. Defaults to true. Setting this to true explicitly disables the deprecation warning.

- **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.

- **default_scheduler_url** Full path to remote scheduler. Defaults to `http://localhost:8082/`. For TLS support use the URL scheme: `https`, example: `https://luigi.example.com:443/` (Note: you will have to
terminate TLS using an HTTP proxy) You can also use this to connect to a local Unix socket using the non-
standard URI scheme: http+unix example: http+unix://%2Fvar%2Frun%2Fluigid%2Fluigid.
sock/
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.

log_level The default log level to use when no logging_conf_file is set. Must be a valid name of a Python log level.
Default is DEBUG.

logging_conf_file Location of the logging configuration file.

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
task lists in the visualiser. Small values can alleviate frozen browsers when there are too many done tasks. This
defaults to 100000 (one hundred thousand).

max_graph_nodes New in version 2.0.0.
The maximum number of nodes returned by a dep_graph or inverse_dep_graph api call. Small values can greatly
speed up graph display in the visualiser by limiting the number of nodes shown. Some of the nodes that are
not sent to the visualiser will still show up as dependencies of nodes that were sent. These nodes are given
TRUNCATED status.

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. Defaults to false.

parallel_scheduling_processes The number of processes to use for parallel scheduling. If not specified the default
number of processes will be the total number of CPUs available.

rpc_connect_timeout Number of seconds to wait before timing out when making an API call. Defaults to 10.0

rpc_retry_attempts The maximum number of retries to connect the central scheduler before giving up. Defaults to 3

rpc_retry_wait Number of seconds to wait before the next attempt will be started to connect to the central scheduler
between two retry attempts. Defaults to 30

8.9.4 [cors]

New in version 2.8.0.

These parameters control /api/<method> CORS behaviour (see: W3C Cross-Origin Resource Sharing).

enabled Enables CORS support. Defaults to false.

allowed_origins A list of allowed origins. Used only if allow_any_origin is false. Configure in JSON array
format, e.g. ["foo", "bar"]. Defaults to empty.

allow_any_origin Accepts requests from any origin. Defaults to false.

allow_null_origin Allows the request to set null value of the Origin header. Defaults to false.

max_age Content of Access-Control-Max-Age. Defaults to 86400 (24 hours).

allowed_methods Content of Access-Control-Allow-Methods. Defaults to GET, OPTIONS.

allowed_headers Content of Access-Control-Allow-Headers. Defaults to Accept, Content-Type, Origin.
exposed_headers Content of `Access-Control-Expose-Headers`. Defaults to empty string (will NOT be sent as a response header).

allow_credentials Indicates that the actual request can include user credentials. Defaults to false.

## 8.9.5 [worker]

These parameters control Luigi worker behavior.

**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`.

**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`.

**ping_interval** Number of seconds to wait between pinging scheduler to let it know that the worker is still alive. Defaults to `1.0`.

**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.

**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. Default value is `0`, meaning no timeout.

**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.

**wait_jitter** Size of jitter to add to the worker wait interval such that the multiple workers do not ask the scheduler for another job at the same time. Default: `5.0`.

**max_keep_alive_idle_duration** New in version 2.8.4.

Maximum duration to keep worker alive while in idle state. Default: `0` (Indefinitely)

**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`.

**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: `retry_count` 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.

**no_install_shutdown_handler** By default, workers will stop requesting new work and finish running pending tasks after receiving a `SIGUSR1` signal. This provides a hook for gracefully shutting down workers that are in the process of running (potentially expensive) tasks. If set to true, Luigi will NOT install this shutdown hook on workers. Note this hook does not work on Windows operating systems, or when jobs are launched outside the main execution thread. Defaults to false.
send_failure_email Controls whether the worker will send e-mails on task and scheduling failures. If set to false, workers will only send e-mails on framework errors during scheduling and all other e-mail must be handled by the scheduler. Defaults to true.

check_unfulfilled_deps If true, the worker checks for completeness of dependencies before running a task. In case unfulfilled dependencies are detected, an exception is raised and the task will not run. This mechanism is useful to detect situations where tasks do not create their outputs properly, or when targets were removed after the dependency tree was built. It is recommended to disable this feature only when the completeness checks are known to be bottlenecks, e.g. when the exists() calls of the dependencies’ outputs are resource-intensive. Defaults to true.

force_multiprocessing By default, luigi uses multiprocessing when more than one worker process is requested. When set to true, multiprocessing is used independent of the the number of workers. Defaults to false.

8.9.6 [elasticsearch]

These parameters control use of elasticsearch

marker_index Defaults to “update_log”.

marker_doc_type Defaults to “entry”.

8.9.7 [email]

General parameters

force_send If true, e-mails are sent in all run configurations (even if stdout is connected to a tty device). Defaults to False.

format Type of e-mail to send. Valid values are “plain”, “html” and “none”. When set to html, tracebacks are wrapped in <pre> tags to get fixed- width font. When set to none, no e-mails will be sent.

Prefix value is plain.

method Valid values are “smtp”, “sendgrid”, “ses” and “sns”. SES and SNS are services of Amazon web services. SendGrid is an email delivery service. The default value is “smtp”.

In order to send messages through Amazon SNS or SES set up your AWS config files or run Luigi on an EC2 instance with proper instance profile.

In order to use sendgrid, fill in your sendgrid API key in the [sendgrid] section.

In order to use smtp, fill in the appropriate fields in the [smtp] section.

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”

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

Set it to SNS Topic ARN if you want to receive notifications through Amazon SNS. Make sure to set method to sns in this case too.

sender User name in from field of error e-mails. Default value: luigi-client@<server_name>
8.9.8 [batch	_notifier]

Parameters controlling the contents of batch notifications sent from the scheduler

**email_interval** Number of minutes between e-mail sends. Making this larger results in fewer, bigger e-mails. Default to 60.

**batch_mode** Controls how tasks are grouped together in the e-mail. Suppose we have the following sequence of failures:

1. TaskA(a=1, b=1)
2. TaskA(a=1, b=1)
3. TaskA(a=2, b=1)
4. TaskA(a=1, b=2)
5. TaskB(a=1, b=1)

For any setting of batch_mode, the batch e-mail will record 5 failures and mention them in the subject. The difference is in how they will be displayed in the body. Here are example bodies with error_messages set to 0.

“all” only groups together failures for the exact same task:

- TaskA(a=1, b=1) (2 failures)
- TaskA(a=1, b=2) (1 failure)
- TaskA(a=2, b=1) (1 failure)
- TaskB(a=1, b=1) (1 failure)

“family” groups together failures for tasks of the same family:

- TaskA (4 failures)
- TaskB (1 failure)

“unbatched_params” groups together tasks that look the same after removing batched parameters. So if TaskA has a batch_method set for parameter a, we get the following:

- TaskA(b=1) (3 failures)
- TaskA(b=2) (1 failure)
- TaskB(a=1, b=2) (1 failure)

Defaults to “unbatched_params”, which is identical to “all” if you are not using batched parameters.

**error_lines** Number of lines to include from each error message in the batch e-mail. This can be used to keep e-mails shorter while preserving the more useful information usually found near the bottom of stack traces. This can be set to 0 to include all lines. If you don’t wish to see error messages, instead set error_messages to 0. Defaults to 20.

**error_messages** Number of messages to preserve for each task group. As most tasks that fail repeatedly do so for similar reasons each time, it’s not usually necessary to keep every message. This controls how many messages are kept for each task or task group. The most recent error messages are kept. Set to 0 to not include error messages in the e-mails. Defaults to 1.

**group_by_error_messages** Quite often, a system or cluster failure will cause many disparate task types to fail for the same reason. This can cause a lot of noise in the batch e-mails. This cuts down on the noise by listing items with identical error messages together. Error messages are compared after limiting by error_lines. Defaults to true.
8.9.9 [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”, “cdh4”, or “apache1”. Defaults to “cdh4”.

8.9.10 [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”, “webhdfs” and “hadoopcli”. Snakebite is much faster, so use of it is encouraged. webhdfs is fast and works with Python 3 as well, but has not been used that much in the wild. Both snakebite and webhdfs requires you to install it 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.

tmp_dir Path to where Luigi will put temporary files on hdfs

8.9.11 [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”.

8.9.12 [kubernetes]
Parameters controlling Kubernetes Job Tasks

auth_method Authorization method to access the cluster. Options are “kubeconfig” or “service-account”

kubeconfig_path Path to kubeconfig file, for cluster authentication. It defaults to ~/.kube/config, which is the default location when using minikube. When auth_method is “service-account” this property is ignored.
**max_retrials** Maximum number of retrials in case of job failure.

### 8.9.13 [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”.

### 8.9.14 [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”.

### 8.9.15 [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”.

### 8.9.16 [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.

### 8.9.17 [retcode]

Configure return codes for the Luigi binary. In the case of multiple return codes that could apply, for example a failing task and missing data, the *numerically greatest* return code is returned.

We recommend that you copy this set of exit codes to your `luigi.cfg` file:

```
[retcode]
# The following return codes are the recommended exit codes for Luigi
# They are in increasing level of severity (for most applications)
already_running=10
missing_data=20
```

(continues on next page)
already_running  This can happen in two different cases. Either the local lock file was taken at the time the invocation starts up. Or, the central scheduler have reported that some tasks could not have been run, because other workers are already running the tasks.

missing_data For when an ExternalTask is not complete, and this caused the worker to give up. As an alternative to fiddling with this, see the [worker] keep_alive option.

not_run For when a task is not granted run permission by the scheduler. Typically because of lack of resources, because the task has been already run by another worker or because the attempted task is in DISABLED state. Connectivity issues with the central scheduler might also cause this. This does not include the cases for which a run is not allowed due to missing dependencies (missing_data) or due to the fact that another worker is currently running the task (already_running).

task_failed For signaling that there were last known to have failed. Typically because some exception have been raised.

scheduling_error For when a task’s complete() or requires() method fails with an exception, or when the limit number of tasks is reached.

unhandled_exception For internal Luigi errors. Defaults to 4, since this type of error probably will not recover over time.

If you customize return codes, prefer to set them in range 128 to 255 to avoid conflicts. Return codes in range 0 to 127 are reserved for possible future use by Luigi contributors.

8.9.18 [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.

8.9.19 [scheduler]

Parameters controlling scheduler behavior

batch_emails Whether to send batch e-mails for failures and disables rather than sending immediate disable e-mails and just relying on workers to send immediate batch e-mails. Defaults to false.

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 retry_count or disable_window_seconds settings.
**retry_count**  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 **retry_count** 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.

**pause_enabled**  If false, disables pause/unpause operations and hides the pause toggle from the visualiser.

**send_messages**  When true, the scheduler is allowed to send messages to running tasks and the central scheduler provides a simple prompt per task to send messages. Defaults to true.

**metrics_collector**  Optional setting allowing Luigi to use a contribution to collect metrics about the pipeline to a third-party. By default this uses the default metric collector that acts as a shell and does nothing. The currently available options are “datadog” and “prometheus”.

### 8.9.20 [sendgrid]

These parameters control sending error e-mails through SendGrid.

**apikey**  API key of the SendGrid account.

### 8.9.21 [smtp]

These parameters control the smtp server setup.

**host**  Hostname for sending mail through smtp. Defaults to localhost.

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

**no_tls**  If true, connects to smtp without TLS. Defaults to false.

**password**  Password to log in to your smtp server. Must be specified for username to have an effect.

**port**  Port number for smtp on smtp_host. Defaults to 0.
ssl  If true, connects to smtp through SSL. Defaults to false.

timeout  Sets the number of seconds after which smtp attempts should time out. Defaults to 10.

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

8.9.22  [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

packages  Comma-separated list of packages to link to on the driver and executors

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):

8.9.23 [task_history]

Parameters controlling storage of task history in a database

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

8.9.24 [execution_summary]

Parameters controlling execution summary of a worker

summary_length Maximum number of tasks to show in an execution summary. If the value is 0, then all tasks will be displayed. Default value is 5.

8.9.25 [webhdfs]

port The port to use for webhdfs. The normal namenode port is probably on a different port from this one.

user Perform file system operations as the specified user instead of $USER. Since this parameter is not honored by any of the other hdfs clients, you should think twice before setting this parameter.

client_type The type of client to use. Default is the “insecure” client that requires no authentication. The other option is the “kerberos” client that uses kerberos authentication.

8.9.26 [datadog]

api_key The api key found in the account settings of Datadog under the API sections.

app_key The application key found in the account settings of Datadog under the API sections.

default_tags Optional settings that adds the tag to all the metrics and events sent to Datadog. Default value is “application:luigi”.

environment Allows you to tweak multiple environment to differentiate between production, staging or development metrics within Datadog. Default value is “development”.

statsd_host The host that has the statsd instance to allow Datadog to send statsd metric. Default value is “localhost”.

statsd_port The port on the host that allows connection to the statsd host. Defaults value is 8125.

metric_namespace Optional prefix to add to the beginning of every metric sent to Datadog. Default value is “luigi”.

8.9.27 Per Task Retry-Policy

Luigi also supports defining retry_policy per task.

class GenerateWordsFromHdfs(luigi.Task):
    retry_count = 2

(continues on next page)
class GenerateWordsFromRDBM(luigi.Task):
    retry_count = 5
    ...

class CountLetters(luigi.Task):
    def requires(self):
        return [GenerateWordsFromHdfs()]
    def run():
        yield GenerateWordsFromRDBM()
    ...

If none of retry-policy fields is defined per task, the field value will be default value which is defined in luigi config file.

To make luigi sticks to the given retry-policy, be sure you run luigi worker with keep_alive config. Please check keep_alive config in [worker] section.

8.9.28 Retry-Policy Fields

The fields below are in retry-policy and they can be defined per task.

- retry_count
- disable_hard_timeout
- disable_window_seconds

8.10 Configure logging

8.10.1 Config options:

Some config options for config [core] section

log_level The default log level to use when no logging_conf_file is set. Must be a valid name of a Python log level. Default is DEBUG.

logging_conf_file Location of the logging configuration file.

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

8.10.2 Config section

If you’re use TOML for configuration file, you can configure logging via logging section in this file. See example for more details.
8.10.3 Luigid CLI options:

`--background` Run daemon in background mode. Disable logging setup and set up log level to INFO for root logger.

`--logdir` set logging with INFO level and output in `$logdir/luigi-server.log` file

8.10.4 Worker CLI options:

`--logging-conf-file` Configuration file for logging.

`--log-level` Default log level. Available values: NOTSET, DEBUG, INFO, WARNING, ERROR, CRITICAL. Default DEBUG. See Python documentation for information about levels difference.

8.10.5 Configuration options resolution order:

1. no_configure_logging option
2. `--background`
3. `--logdir`
4. `--logging-conf-file`
5. logging_conf_file option
6. logging section
7. `--log-level`
8. log_level option

8.11 Design and limitations

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 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 are 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.
CHAPTER 9

API Reference

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>luigi</code></td>
<td>Package containing core luigi functionality.</td>
</tr>
<tr>
<td><code>luigi.contrib</code></td>
<td>Package containing optional and-on functionality.</td>
</tr>
<tr>
<td><code>luigi.tools</code></td>
<td>Sort of a standard library for doing stuff with Tasks at a somewhat abstract level.</td>
</tr>
<tr>
<td><code>luigi.local_target</code></td>
<td>LocalTarget provides a concrete implementation of a Target class that uses files on the local file system</td>
</tr>
</tbody>
</table>

9.1 luigi package

9.1.1 Subpackages

`luigi.configuration` package

Submodules

`luigi.configuration.base_parser` module

```python
class luigi.configuration.base_parser.BaseParser

   @classmethod
   @instance(*args, **kwargs)
   Singleton getter

   @classmethod
   def add_config_path(path)

   @classmethod
   def reload()
```
**luigi.configuration.cfg_parser 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 luigi.cfg (or 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.

**exception** luigi.configuration.cfg_parser.InterpolationMissingEnvvarError(
  option,
  section,
  value,
  envvar)

Bases: ConfigParser.InterpolationError

Raised when option value refers to a nonexisting environment variable.

**class** luigi.configuration.cfg_parser.EnvironmentInterpolation

Bases: object

Custom interpolation which allows values to refer to environment variables using the ${ENVVAR} syntax.

**before_get** (parser, section, option, value, defaults)

**class** luigi.configuration.cfg_parser.CombinedInterpolation (interpolations)

Bases: object

Custom interpolation which applies multiple interpolations in series.

**Parameters** interpolations – a sequence of configparser.Interpolation objects.

**before_get** (parser, section, option, value, defaults)

**before_read** (parser, section, option, value)

**before_set** (parser, section, option, value)

**before_write** (parser, section, option, value)

**class** luigi.configuration.cfg_parser.LuigiConfigParser (defaults=None,
  dict_type=collection.OrderedDict,
  allow_no_value=False)

Bases: luigi.configuration.base_parser.BaseParser, ConfigParser.ConfigParser

NO_DEFAULT = <object object>

enabled = True

**classmethod** reload()

**has_option** (section, option)

modified has_option Check for the existence of a given option in a given section. If the specified ’section’ is None or an empty string, DEFAULT is assumed. If the specified ’section’ does not exist, returns False.

**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.core module

luigi.configuration.core.get_config (parser='cfg')
Get configs singleton for parser
luigi.configuration.core.add_config_path (path)
Select config parser by file extension and add path into parser.

luigi.configuration.toml_parser module

class luigi.configuration.toml_parser.LuigiTomlParser
    Bases: luigi.configuration.base_parser.BaseParser
    NO_DEFAULT = <object object>
    enabled = False
data = {}
read (config_paths)
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)
has_option (section, option)

Module contents

luigi.configuration.add_config_path (path)
Select config parser by file extension and add path into parser.
luigi.configuration.get_config (parser='cfg')
Get configs singleton for parser
class luigi.configuration.LuigiConfigParser (defaults=None,
dict_type=<class 'collections.OrderedDict'>,
allow_no_value=False)
Bases: luigi.configuration.base_parser.BaseParser, ConfigParser.ConfigParser
NO_DEFAULT = <object object>
enabled = True
classmethod reload()
**has_option** *(section, option)*

Modified has_option Check for the existence of a given option in a given section. If the specified ‘section’ is None or an empty string, DEFAULT is assumed. If the specified ‘section’ does not exist, returns False.

**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.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.

In hdfs land, “mv” is often called rename. So we add an alias for move() called rename(). This is also to keep backward compatibility since move() became standardized in luigi’s filesystem interface.
rename_dont_move (path, dest)
   Override this method with an implementation that uses rename2, which is a rename operation that never moves.


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

chmod (path, permissions, recursive=False)

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

count (path)
   Count contents in a directory

copy (path, destination)
   Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only single file copying but S3Client copies either a file or a directory as required.

put (local_path, destination)

get (path, local_destination)

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.

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

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(client_cache=<thread_local object>)

Creates the client as specified in the luigi.cfg configuration.

luigi.contrib.hdfs.clients.exists(*args, **kwargs)
luigi.contrib.hdfs.clients.rename(*args, **kwargs)
luigi.contrib.hdfs.clients.remove(*args, **kwargs)
luigi.contrib.hdfs.clients.mkdir(*args, **kwargs)
luigi.contrib.hdfs.clients.listdir(*args, **kwargs)

**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
    
    client_version = IntParameter (defaults to None)
    effective_user = OptionalParameter (defaults to None): Optionally specifies the effective user for snakebite.
    snakebite_autoconfig = BoolParameter (defaults to False)
    namenode_host = OptionalParameter (defaults to None)
    namenode_port = IntParameter (defaults to None)
    client = Parameter (defaults to hadoopcli)
    tmp_dir = OptionalParameter (defaults to None)

class luigi.contrib.hdfs.config.hadoopcli(*args, **kwargs)
    Bases: luigi.task.Config
    
    command = Parameter (defaults to hadoop): The hadoop command, will run split() on it.
    version = Parameter (defaults to cdh4): Can also be cdh3 or apache1

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()
    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 @type include_unix_username: 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.

```python
exception luigi.contrib.hdfs.error.HDFSCliError(command, returncode, stdout, stderr)
    Bases: exceptions.Exception
```

**luigi.contrib.hdfs.format module**

```python
exception luigi.contrib.hdfs.format.HdfsAtomicWriteError
    Bases: exceptions.IOError

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)
    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.
    move(path, dest)
        Move a file, as one would expect.
    remove(path, recursive=True, skip_trash=False)
        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.

    chmod(path, permissions, recursive=False)
    chown(path, owner, group, recursive=False)
    count(path)
        Count contents in a directory
    copy(path, destination)
        Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only
        single file copying but S3Client copies either a file or a directory as required.
    put(local_path, destination)
    get(path, local_destination)
    getmerge(path, local_destination, new_line=False)
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.

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

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.

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, parents=True, raise_if_exists=False)

No explicit -p switch, this version of Hadoop always creates parent directories.

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

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.

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)

Use hadoop fs -stat to check file existence.

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.

```python
static list_path(path)
```

**get_bite()**

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

```python
exists(path)
```

Use snakebite.test to check file existence.

**Parameters**

- `path` *(string)* – path to test

**Returns** boolean, True if path exists in HDFS

```python
move(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

```python
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

```python
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

```python
chmod(path, permissions, recursive=False)
```

Use snakebite.chmod, if available.

**Parameters**

- `path` *(either a string or sequence of strings)* – update-able file(s) or directory(ies)
- `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

**get_merge** *(path, local_destination)*

Using snakebite getmerge to implement this. :

```
:param path: HDFS directory
:param local_destination: path on the system running Luigi
:return: merge of the directory
```

**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')*
  
  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. Using b is not supported; initialize with format=Nop instead.

- **remove** *(skip_trash=False)*
  
  Remove the resource at the path specified by this FileSystemTarget.

  This method is implemented by using `fs`.

- **rename** *(path, raise_if_exists=False)*
  
  Does not change self.path.

  Unlike `move_dir()`, `rename()` might cause nested directories. See spotify/luigi#522

- **move** *(path, raise_if_exists=False)*
  
  Alias for `rename()`

- **move_dir** *(path)*
  
  Move using `rename_dont_move`

  New since after luigi v2.1: Does not change self.path

  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
**copy**(*dst_dir*)
Copy to destination directory.

**is_writable**()
Currently only works with hadoopcli

```python
class luigi.contrib.hdfs.target.HdfsFlagTarget(path, format=None, client=None, flag='_SUCCESS')

Bases: luigi.contrib.hdfs.target.HdfsTarget

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 HdfsTarget)
• 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.

Initializes a HdfsFlagTarget.

**Parameters**

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

**exists**()
Returns True if the path for this FileSystemTarget exists; False otherwise.
This method is implemented by using fs.

**luigi.contrib.hdfs.webhdfs_client module**

A luigi file system client that wraps around the hdfs-library (a webhdfs client)
This is a sensible fast alternative to snakebite. In particular for python3 users, where snakebite is not supported at the
time of writing (dec 2015).

Note. This wrapper client is not feature complete yet. As with most software the authors only implement the features
they need. If you need to wrap more of the file system operations, please do and contribute back.

```python
class luigi.contrib.hdfs.webhdfs_client.webhdfs(*args, **kwargs)

Bases: luigi.task.Config

port = IntParameter (defaults to 50070): Port for webhdfs
user = Parameter (defaults to ): Defaults to $USER envvar
client_type = ChoiceParameter (defaults to insecure): Type of hdfs client to use. Choices: {insecure, kerberos}

class luigi.contrib.hdfs.webhdfs_client.WebHdfsClient(host=None, port=None, user=None, client_type=None)

Bases: luigi.contrib.hdfs.abstract_client.HdfsFileSystem

A webhdfs that tries to confirm to luigis interface for file existence.
The library is using this api.

**url**

**client**
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=True, skip_trash=False)
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.

read (hdfs_path, offset=0, length=None, buffer_size=None, chunk_size=1024, buffer_char=None)

move (path, dest)
Move a file, as one would expect.

mkdir (path, parents=True, mode=493, raise_if_exists=False)
Has no returnvalue (just like WebHDFS)

chmod (path, permissions, recursive=False)
Raise a NotImplementedError exception.

chown (path, owner, group, recursive=False)
Raise a NotImplementedError exception.

count (path)
Raise a NotImplementedError exception.

copy (path, destination)
Raise a NotImplementedError exception.

put (local_path, destination)
Restricted version of upload

get (path, local_destination)
Restricted version of download

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

touchz (path)
To touchz using the web hdfs “write” cmd.

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.
Since the hdfs functionality is quite big in luigi, it’s split into smaller files under `luigi/contrib/hdfs/*.py`. But for the sake of convenience and API stability, everything is reexported under `luigi.contrib.hdfs`.

**Submodules**

- **luigi.contrib.azureblob module**
- **luigi.contrib.batch module**

   AWS Batch wrapper for Luigi

   From the AWS website:

   AWS Batch enables you to run batch computing workloads on the AWS Cloud.

   Batch computing is a common way for developers, scientists, and engineers to access large amounts of compute resources, and AWS Batch removes the undifferentiated heavy lifting of configuring and managing the required infrastructure. AWS Batch is similar to traditional batch computing software. This service can efficiently provision resources in response to jobs submitted in order to eliminate capacity constraints, reduce compute costs, and deliver results quickly.

   See [AWS Batch User Guide](https://docs.aws.amazon.com/batch/latest/userguide/) for more details.

   To use AWS Batch, you create a `jobDefinition` JSON that defines a `docker run` command, and then submit this JSON to the API to queue up the task. Behind the scenes, AWS Batch auto-scales a fleet of EC2 Container Service instances, monitors the load on these instances, and schedules the jobs.

   This boto3-powered wrapper allows you to create Luigi Tasks to submit Batch `jobDefinition`’s. You can either pass a dict (mapping directly to the `jobDefinition` JSON) OR an Amazon Resource Name (arn) for a previously registered `jobDefinition`.

   Requires:

   - boto3 package
   - Amazon AWS credentials discoverable by boto3 (e.g., by using `aws configure` from awscli)
   - An enabled AWS Batch job queue configured to run on a compute environment.

Written and maintained by Jake Feala (@jfeala) for Outlier Bio (@outlierbio)

**exception** luigi.contrib.batch.BatchJobException

**class** luigi.contrib.batch.BatchClient (`poll_time`=10)

   Bases: object

   `get_active_queue()`  
   Get name of first active job queue

   `get_job_id_from_name`(`job_name`)  
   Retrieve the first job ID matching the given name

   `get_job_status`(`job_id`)  
   Retrieve task statuses from ECS API

   **Parameters** (str) `job_id` – AWS Batch job uuid

   Returns one of `{SUBMITTED,PENDING,RUNNABLE,STARTING,RUNNING,SUCCEEDED,FAILED}`

   `get_logs`(`log_stream_name`, `get_last`=50)  
   Retrieve log stream from CloudWatch

9.1. Luigi package
submit_job (job_definition, parameters, job_name=None, queue=None)

Wrap submit_job with useful defaults

wait_on_job (job_id)

Poll task status until STOPPED

register_job_definition (json_fpath)

Register a job definition with AWS Batch, using a JSON

class luigi.contrib.batch.BatchTask(*args, **kwargs)

Bases: luigi.task.Task

Base class for an Amazon Batch job

Amazon Batch requires you to register “job definitions”, which are JSON descriptions for how to issue the
docker run command. This Luigi Task requires a pre-registered Batch jobDefinition name passed as a

Parameters

- (str) (job_definition) – name of pre-registered jobDefinition
- job_name – name of specific job, for tracking in the queue and logs.
- job_queue – name of job queue where job is going to be submitted.

job_definition = Parameter
job_name = OptionalParameter (defaults to None)
job_queue = OptionalParameter (defaults to None)
poll_time = IntParameter (defaults to 10)
run()

The task run method, to be overridden in a subclass.

See Task.run

parameters

Override to return a dict of parameters for the Batch Task

luigi.contrib.beam_dataflow module

class luigi.contrib.beam_dataflow.DataflowParamKeys

Bases: object

Defines the naming conventions for Dataflow execution params. For example, the Java API expects param
names in lower camel case, whereas the Python implementation expects snake case.

runner
project
zone
region
staging_location
temp_location
gcp_temp_location
num_workers
Luigi wrapper for a Dataflow job. Must be overridden for each Beam SDK with that SDK’s `dataflow_executable()`.

For more documentation, see: https://cloud.google.com/dataflow/docs/guides/specifying-exec-params

The following required Dataflow properties must be set:

- `project` # GCP project ID
- `temp_location` # Cloud storage path for temporary files

The following optional Dataflow properties can be set:

- `network` # Network in GCE to be used for launching workers
- `subnetwork` # Subnetwork in GCE to be used for launching workers
- `disk_size_gb` # Remote worker disk size. Minimum value is 30GB
- `worker_machine_type` # Machine type to create Dataflow worker VMs
- `job_name` # Custom job name, must be unique across project’s active jobs
- `worker_disk_type` # Specify SSD for local disk or defaults to hard disk as a full URL of disk type resource
- `service_account` # Service account of Dataflow VMs/workers
- `region` # Region to deploy Dataflow job to
- `zone` # Availability zone for launching workers
- `staging_location` # Cloud Storage bucket for Dataflow to stage binary files
- `gcp_temp_location` # Cloud Storage path for Dataflow to stage temporary files
labels # Custom GCP labels attached to the Dataflow job  Default: nothing

project = None
runner = None
temp_location = None
staging_location = None
gcp_temp_location = None
num_workers = None
autoscaling_algorithm = None
max_num_workers = None
network = None
subnetwork = None
disk_size_gb = None
worker_machine_type = None
job_name = None
worker_disk_type = None
service_account = None
zone = None
region = None
labels = {}

# Command alias

cmd_line_runner
    alias of _CmdLineRunner

dataflow_params = None

dataflow_executable()
    Command representing the Dataflow executable to be run. For example:
    return ['java', 'com.spotify.luigi.MyClass', '-Xmx256m']

args()
    Extra String arguments that will be passed to your Dataflow job. For example:
    return ['-setup_file=setup.py']

before_run()
    Hook that gets called right before the Dataflow job is launched. Can be used to setup any temporary
    files/tabsles, validate input, etc.

on_successful_run()
    Callback that gets called right after the Dataflow job has finished successfully but before validate_output
    is run.

validate_output()
    Callback that can be used to validate your output before it is moved to its final location. Returning false
    here will cause the job to fail, and output to be removed instead of published.

file_pattern()
    If one/some of the input target files are not in the pattern of part-, we can add the key of the required target

and the correct file pattern that should be appended in the command line here. If the input target key is not
found in this dict, the file pattern will be assumed to be part-* for that target.

:return A dictionary of overridden file pattern that is not part-* for the inputs

on_successful_output_validation()
    Callback that gets called after the Dataflow job has finished successfully if validate_output returns True.

cleanup_on_error(error)
    Callback that gets called after the Dataflow job has finished unsuccessfully, or validate_output returns
    False.

run()
    The task run method, to be overridden in a subclass.

    See Task.run

static get_target_path(target)
    Given a luigi Target, determine a stringly typed path to pass as a Dataflow job argument.

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

        AVRO = 'AVRO'
        CSV = 'CSV'

        DATASTORE_BACKUP = 'DATASTORE_BACKUP'
        NEWLINE_DELIMITED_JSON = 'NEWLINE_DELIMITED_JSON'

class luigi.contrib.bigquery.FieldDelimiter
    Bases: object

    The separator for fields in a CSV file. The separator can be any ISO-8859-1 single-byte character. To use
    a character in the range 128-255, you must encode the character as UTF8. BigQuery converts the string to
    ISO-8859-1 encoding, and then uses the first byte of the encoded string to split the data in its raw, binary state.
    BigQuery also supports the escape sequence "\" to specify a tab separator. The default value is a comma (',').
https://cloud.google.com/bigquery/docs/reference/v2/jobs#configuration.load

class luigi.contrib.bigquery.PrintHeader
    Bases: object
    TRUE = True
    FALSE = False
class luigi.contrib.bigquery.DestinationFormat
    Bases: object
    AVRO = 'AVRO'
    CSV = 'CSV'
    NEWLINE_DELIMITED_JSON = 'NEWLINE_DELIMITED_JSON'
class luigi.contrib.bigquery.Compression
    Bases: object
    GZIP = 'GZIP'
    NONE = 'NONE'
class luigi.contrib.bigquery.Encoding
    Bases: object
    [Optional] The character encoding of the data. The supported values are UTF-8 or ISO-8859-1. The default value is UTF-8.
    UTF_8 = 'UTF-8'
    ISO_8859_1 = 'ISO-8859-1'
class luigi.contrib.bigquery.BQDataset(project_id, dataset_id, location)
    Bases: tuple
    Create new instance of BQDataset(project_id, dataset_id, location)
    dataset_id
        Alias for field number 1
    location
        Alias for field number 2
    project_id
        Alias for field number 0
class luigi.contrib.bigquery.BQTable
    Bases: luigi.contrib.bigquery.BQTable
    Create new instance of BQTable(project_id, dataset_id, table_id, location)
    dataset
    uri
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. If regional location is specified for the dataset, that is also checked to be compatible with the remote dataset, otherwise an exception is thrown.

    param dataset
    type dataset  BQDataset

table_exists (table)

Returns whether the given table exists.

    Parameters table (BQTable) –

make_dataset (dataset, raise_if_exists=False, body=None)

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) –

get_view (table)

Returns the SQL query for a view, or None if it doesn’t exist or is not a view.

    Parameters table (BQTable) – The table containing the view.

update_view (table, view)

Updates the SQL query for a view.
If the output table exists, it is replaced with the supplied view query. Otherwise a new table is created with this view.

**Parameters**

- **table** (BQTable) – The table to contain the view.
- **view** (str) – The SQL query for the view.

**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, location=None)*

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

A constructor that takes a BQTable.

**exists** *

Returns True if the Target exists and False otherwise.

**class** luigi.contrib.bigquery.MixinBigQueryBulkComplete

**Bases:** object

Allows to efficiently check if a range of BigQueryTargets are complete. This enables scheduling tasks with luigi range tools.

If you implement a custom Luigi task with a BigQueryTarget output, make sure to also inherit from this mixin to enable range support.

**classmethod** bulk_complete *(parameter_tuples)*

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

**Bases:** luigi.contrib.bigquery.MixinBigQueryBulkComplete, luigi.task.Task

Load data into BigQuery from GCS.

**source_format**

The source format to use (see SourceFormat).
encoding
   The encoding of the data that is going to be loaded (see Encoding).

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.

max_bad_records
   The maximum number of bad records that BigQuery can ignore when reading data.
   If the number of bad records exceeds this value, an invalid error is returned in the job result.

field_delimiter
   The separator for fields in a CSV file. The separator can be any ISO-8859-1 single-byte character.

source_uris()
   The fully-qualified URIs that point to your data in Google Cloud Storage.
   Each URI can contain one ‘*’ wildcard character and it must come after the ‘bucket’ name.

skip_leading_rows
   The number of rows at the top of a CSV file that BigQuery will skip when loading the data.
   The default value is 0. This property is useful if you have header rows in the file that should be skipped.

allow_jagged_rows
   Accept rows that are missing trailing optional columns. The missing values are treated as nulls.
   If false, records with missing trailing columns are treated as bad records, and if there are too many bad records,
   an invalid error is returned in the job result. The default value is false. Only applicable to CSV, ignored for other formats.

ignore_unknown_values
   Indicates if BigQuery should allow extra values that are not represented in the table schema.
   If true, the extra values are ignored. If false, records with extra columns are treated as bad records,
   and if there are too many bad records, an invalid error is returned in the job result. The default value is false.
   The sourceFormat property determines what BigQuery treats as an extra value:
   CSV: Trailing columns JSON: Named values that don’t match any column names

allow_quoted_new_lines
   Indicates if BigQuery should allow quoted data sections that contain newline characters in a CSV file. The default value is false.

run()
   The task run method, to be overridden in a subclass.
   See Task.run

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

**flatten_results**
Flattens all nested and repeated fields in the query results. allowLargeResults must be true if this is set to False.

**query**
The query, in text form.

**query_mode**
The query mode. See *QueryMode*.

**udf_resource_uris**
Iterator of code resource to load from a Google Cloud Storage URI (gs://bucket/path).

**use Legacy_sql**
Whether to use legacy SQL

**run()**
The task run method, to be overridden in a subclass.

See *Task.run*

```python
class luigi.contrib.bigquery.BigQueryCreateViewTask(*args, **kwargs)
Bases: luigi.task.Task

Creates (or updates) a view in BigQuery.

The output of this task needs to be a BigQueryTarget. Instances of this class should specify the view SQL in the view property.

If a view already exist in BigQuery at output(), it will be updated.

**view**
The SQL query for the view, in text form.

**complete()**
If the task has any outputs, return True if all outputs exist. Otherwise, return False.

However, you may freely override this method with custom logic.

**run()**
The task run method, to be overridden in a subclass.

See *Task.run*
```

```python
class luigi.contrib.bigquery.ExternalBigQueryTask(*args, **kwargs)
Bases: luigi.contrib.bigquery.MixinBigQueryBulkComplete, luigi.task.ExternalTask

An external task for a BigQuery target.
```

```python
class luigi.contrib.bigquery.BigQueryExtractTask(*args, **kwargs)
Bases: luigi.task.Task

Extracts (unloads) a table from BigQuery to GCS.

This tasks requires the input to be exactly one BigQueryTarget while the output should be one or more GCSTargets from luigi.contrib.gcs depending on the use of destinationUris property.
```
destination_uris
   The fully-qualified URIs that point to your data in Google Cloud Storage. Each URI can contain one ‘*’ wildcard character and it must come after the ‘bucket’ name.

   Wildcarded destinationUris in GCSQueryTarget might not be resolved correctly and result in incomplete data. If a GCSQueryTarget is used to pass wildcarded destinationUris be sure to overwrite this property to suppress the warning.

print_header
   Whether to print the header or not.

field_delimiter
   The separator for fields in a CSV file. The separator can be any ISO-8859-1 single-byte character.

destination_format
   The destination format to use (see DestinationFormat).

compression
   Whether to use compression.

run()
   The task run method, to be overridden in a subclass.

   See Task.run

luigi.contrib.bigquery.BigqueryClient
   alias of luigi.contrib.bigquery.BigQueryClient

luigi.contrib.bigquery.BigqueryTarget
   alias of luigi.contrib.bigquery.BigQueryTarget

luigi.contrib.bigquery.MixinBigqueryBulkComplete
   alias of luigi.contrib.bigquery.MixinBigQueryBulkComplete

luigi.contrib.bigquery.BigqueryLoadTask
   alias of luigi.contrib.bigquery.BigQueryLoadTask

luigi.contrib.bigquery.BigqueryRunQueryTask
   alias of luigi.contrib.bigquery.BigQueryRunQueryTask

luigi.contrib.bigquery.BigqueryCreateViewTask
   alias of luigi.contrib.bigquery.BigQueryCreateViewTask

luigi.contrib.bigquery.ExternalBigqueryTask
   alias of luigi.contrib.bigquery.ExternalBigQueryTask

luigi.contrib.bigquery_avro module

   Specialized tasks for handling Avro data in BigQuery from GCS.

class luigi.contrib.bigquery_avro.BigQueryLoadAvro(*args, **kwargs)
   Bases: luigi.contrib.bigquery.BigQueryLoadTask

   A helper for loading specifically Avro data into BigQuery from GCS.

   Copies table level description from Avro schema doc, BigQuery internally will copy field-level descriptions to the table.

   Suitable for use via subclassing: override requires() to return Task(s) that output to GCS Targets; their paths are expected to be URIs of .avro files or URI prefixes (GCS “directories”) containing one or many .avro files.

   Override output() to return a BigQueryTarget representing the destination table.
source_format = 'AVRO'

source_uris()
    The fully-qualified URIs that point to your data in Google Cloud Storage.
    Each URI can contain one '*' wildcard character and it must come after the 'bucket' name.

run()
    The task run method, to be overridden in a subclass.
    See Task.run

luigi.contrib.datadog_metric module

class luigi.contrib.datadog_metric.datadog(*args, **kwargs)
    Bases: luigi.task.Config
    api_key = Parameter (defaults to dummy_api_key): API key provided by Datadog
    app_key = Parameter (defaults to dummy_app_key): APP key provided by Datadog
    default_tags = Parameter (defaults to application:luigi): Default tags for every event
    environment = Parameter (defaults to development): Environment of which the pipeline is ran from
    metric_namespace = Parameter (defaults to luigi): Default namespace for events and metrics
    statsd_host = Parameter (defaults to localhost): StatsD host implementing the Datadog service
    statsd_port = IntParameter (defaults to 8125): StatsD port implementing the Datadog service

class luigi.contrib.datadog_metric.DatadogMetricsCollector(*args, **kwargs)
    Bases: luigi.metrics.MetricsCollector
    handle_task_started(task)
    handle_task_failed(task)
    handle_task_disabled(task, config)
    handle_task_done(task)
    default_tags

luigi.contrib.dataproc module

luigi bindings for Google Dataproc on Google Cloud

luigi.contrib.dataproc.get_dataproc_client()

luigi.contrib.dataproc.set_dataproc_client(client)

class luigi.contrib.dataproc.DataprocBaseTask(*args, **kwargs)
    Bases: luigi.contrib.dataproc._DataprocBaseTask
    Base task for running jobs in Dataproc. It is recommended to use one of the tasks specific to your job type.
    Extend this class if you need fine grained control over what kind of job gets submitted to your Dataproc cluster.
    submit_job(job_config)
    submit_spark_job(jars, main_class, job_args=None)
    submit_pyspark_job(job_file, extra_files=[], job_args=None)
wait_for_job()

class luigi.contrib.dataproc.DataprocSparkTask(*args, **kwargs):
    Bases: luigi.contrib.dataproc.DataprocBaseTask
    Runs a spark jobs on your Dataproc cluster
    main_class = Parameter
    jars = Parameter (defaults to )
    job_args = Parameter (defaults to )
    run()
        The task run method, to be overridden in a subclass.
        See Task.run

class luigi.contrib.dataproc.DataprocPysparkTask(*args, **kwargs):
    Bases: luigi.contrib.dataproc.DataprocBaseTask
    Runs a pyspark jobs on your Dataproc cluster
    job_file = Parameter
    extra_files = Parameter (defaults to )
    job_args = Parameter (defaults to )
    run()
        The task run method, to be overridden in a subclass.
        See Task.run

class luigi.contrib.dataproc.CreateDataprocClusterTask(*args, **kwargs):
    Bases: luigi.contrib.dataproc._DataprocBaseTask
    Task for creating a Dataproc cluster.
    gcloud_zone = Parameter (defaults to europe-west1-c)
    gcloud_network = Parameter (defaults to default)
    master_node_type = Parameter (defaults to n1-standard-2)
    master_disk_size = Parameter (defaults to 100)
    worker_node_type = Parameter (defaults to n1-standard-2)
    worker_disk_size = Parameter (defaults to 100)
    worker_normal_count = Parameter (defaults to 2)
    worker_preemptible_count = Parameter (defaults to 0)
    image_version = Parameter (defaults to )
    complete()
        If the task has any outputs, return True if all outputs exist. Otherwise, return False.
        However, you may freely override this method with custom logic.
    run()
        The task run method, to be overridden in a subclass.
        See Task.run
class luigi.contrib.dataproc.DeleteDataprocClusterTask(*args, **kwargs)

Bases: luigi.contrib.dataproc._DataprocBaseTask

Task for deleting a Dataproc cluster. One of the uses for this class is to extend it and have it require a Dataproc task that does a calculation and have that task extend the cluster creation task. This allows you to create chains where you create a cluster, run your job and remove the cluster right away. (Store your input and output files in gs://... instead of hdfs://... if you do this).

complete()

If the task has any outputs, return True if all outputs exist. Otherwise, return False.

However, you may freely override this method with custom logic.

run()

The task run method, to be overridden in a subclass.

See Task.run

luigicontrib.docker_runner module

Docker container wrapper for Luigi.

Enables running a docker container as a task in luigi. This wrapper uses the Docker Python SDK to communicate directly with the Docker API avoiding the common pattern to invoke the docker client from the command line. Using the SDK it is possible to detect and properly handle errors occurring when pulling, starting or running the containers. On top of this, it is possible to mount a single file in the container and a temporary directory is created on the host and mounted allowing the handling of files bigger than the container limit.

Requires:

• docker: pip install docker

Written and maintained by Andrea Pierleoni (@apierleoni). Contributions by Eliseo Papa (@elipapa).

class luigi.contrib.docker_runner.DockerTask(*args, **kwargs)

Bases: luigi.task.Task

When a new instance of the DockerTask class gets created: - call the parent class __init__ method - start the logger - init an instance of the docker client - create a tmp dir - add the temp dir to the volume binds specified in the task

image

command

name

container_options

environment

container_tmp_dir

binds

Override this to mount local volumes, in addition to the /tmp/luigi which gets defined by default. This should return a list of strings. e.g. ['/hostpath1:/containerpath1', '/hostpath2:/containerpath2']

network_mode
docker_url
auto_remove
force_pull
mount_tmp

run()

The task run method, to be overridden in a subclass.

See Task.run

luigi.contrib.dropbox module

luigi.contrib.dropbox.accept_trailing_slash_in_existing_dirpaths(func)
luigi.contrib.dropbox.accept_trailing_slash(func)

class luigi.contrib.dropbox.DropboxClient (token, user_agent='Luigi')
Bases: luigi.target.FileSystem

Dropbox client for authentication, designed to be used by the DropboxTarget class.

Parameters
token (str) – Dropbox Oauth2 Token. See DropboxTarget for more information about generating a token

exists(path, *args, **kwargs)

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, *args, **kwargs)

Remove file or directory at location path

Parameters
• path (str) – a path within the FileSystem to remove.

mkdir(path, *args, **kwargs)

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.

isdir(path, *args, **kwargs)

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, *args, **kwargs)

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.

**move** *(path, *args, **kwargs)*
Move a file, as one would expect.

**copy** *(path, *args, **kwargs)*
Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only single file copying but S3Client copies either a file or a directory as required.

**download_as_bytes** *(path)*

**upload** *(tmp_path, dest_path)*

**class luigi.contrib.dropbox.ReadableDropboxFile** *(path, client)*
Bases: object
Represents a file inside the Dropbox cloud which will be read

**Parameters**

- **path** *(str)* – Dropbox path of the file to be read (always starting with /)
- **client** *(DropboxClient)* – a DropboxClient object (initialized with a valid token)

**read**()
**close**()
**readable**()
**writable**()
**seekable**()

**class luigi.contrib.dropbox.AtomicWritableDropboxFile** *(path, client)*
Bases: luigi.target.AtomicLocalFile
Represents a file that will be created inside the Dropbox cloud

**Parameters**

- **path** *(str)* – Destination path inside Dropbox
- **client** *(DropboxClient)* – a DropboxClient object (initialized with a valid token, for the desired account)

**move_to_final_destination**()
After editing the file locally, this function uploads it to the Dropbox cloud

**class luigi.contrib.dropbox.DropboxTarget** *(path, token, format=None, user_agent='Luigi')*
Bases: luigi.target.FileSystemTarget
A Dropbox filesystem target.
Create an Dropbox Target for storing data in a dropbox.com account

**About the path parameter**
The path must start with ‘/’ and should not end with ‘/’ (even if it is a directory). The path must not contain adjacent slashes (’/files/img.jpg’ is an invalid path)

If the app has ‘App folder’ access, then / will refer to this app folder (which mean that there is no need to prepend the name of the app to the path) Otherwise, if the app has ‘full access’, then / will refer to the root of the Dropbox folder

**About the token parameter:**
The Dropbox target requires a valid OAuth2 token as a parameter (which means that a Dropbox API app must be created. This app can have ‘App folder’ access or ‘Full Dropbox’, as desired).

Information about generating the token can be read here:

- https://blogs.dropbox.com/developers/2014/05/generate-an-access-token-for-your-own-account/

**Parameters**

- **path** *(str)* – Remote path in Dropbox (starting with ‘/’).
- **token** *(str)* – a valid OAuth2 Dropbox token.
- **format** *(luigi.Format)* – the luigi format to use (e.g. `luigi.format.Nop`)

**fs**

**temporary_path**(**kwds**)

A context manager that enables a reasonably short, general and magic-less way to solve the **Atomic Writes Problem**.

- On **entering**, it will create the parent directories so the temporary_path is writeable right away. This step uses `FileSystem.mkdir()`.
- On **exiting**, it will move the temporary file if there was no exception thrown. This step uses `FileSystem.rename_dont_move()`

The file system operations will be carried out by calling them on `fs`.

The typical use case looks like this:

```python
class MyTask(luigi.Task):
    def output(self):
        return MyFileSystemTarget(...)

    def run(self):
        with self.output().temporary_path() as self.temp_output_path:
            run_some_external_command(output_path=self.temp_output_path)
```

**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. Using *b* is not supported; initialize with `format=Nop` instead.

**luigi.contrib.ecs module**

EC2 Container Service wrapper for Luigi

From the AWS website:

Amazon EC2 Container Service (ECS) is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances.
To use ECS, you create a taskDefinition JSON that defines the docker run command for one or more containers in a task or service, and then submit this JSON to the API to run the task.

This boto3-powered wrapper allows you to create Luigi Tasks to submit ECS taskDefinition s. You can either pass a dict (mapping directly to the taskDefinition JSON) OR an Amazon Resource Name (arn) for a previously registered taskDefinition.

Requires:

• boto3 package
• Amazon AWS credentials discoverable by boto3 (e.g., by using aws configure from awscli)
• A running ECS cluster (see ECS Get Started)

Written and maintained by Jake Feala (@jfeala) for Outlier Bio (@outlierbio)

```python
class luigi.contrib.ecs.ECSTask(*args, **kwargs):
    Bases: luigi.task.Task
    
    Base class for an Amazon EC2 Container Service Task
    
    Amazon ECS requires you to register “tasks”, which are JSON descriptions for how to issue the docker run command. This Luigi Task can either run a pre-registered ECS taskDefinition, OR register the task on the fly from a Python dict.

    Parameters

    • task_def_arn – pre-registered task definition ARN (Amazon Resource Name), of the form:

      arn:aws:ecs:<region>:<user_id>:task-definition/<family>:<tag>

    • task_def – dict describing task in taskDefinition JSON format, for example:

      ```python
      task_def = {
          'family': 'hello-world',
          'volumes': [],
          'containerDefinitions': [
              {  
                  'memory': 1,
                  'essential': True,
                  'name': 'hello-world',
                  'image': 'ubuntu',
                  'command': ['/bin/echo', 'hello world']
              }
          ]
      }
      ```

    • cluster – str defining the ECS cluster to use. When this is not defined it will use the default one.

      ```python
      task_def_arn = OptionalParameter (defaults to None)
      task_def = OptionalParameter (defaults to None)
      cluster = Parameter (defaults to default)
      ```

      Expose the ECS task ID

      Command passed to the containers
```
Override to return list of dicts with keys ‘name’ and ‘command’, describing the container names and commands to pass to the container. Directly corresponds to the `overrides` parameter of runTask API. For example:

```
[
    {
        'name': 'myContainer',
        'command': ['/bin/sleep', '60']
    }
]
```

`run()`

The task run method, to be overridden in a subclass.

See `Task.run`

**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):

```python
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:

```python
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:
Luigi Documentation, Release 2.8.13

- 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 luigi.cfg configuration options:

```yaml
[elasticsearch]
marker-index = update_log
marker-doc-type = entry
```

```python
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=None)

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 sufficient but, for documentation, we index the parameters `update_id`, `target_index`, `target_doc_type` and `date` as well.

```python
exists()
```

Test, if this task has been run.

```python
create_marker_index()
```

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

```python
ensure_hist_size()
```

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

```python
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.

**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.

**luigi.contrib.external_daily_snapshot module**

**class luigi.contrib.external_daily_snapshot.ExternalDailySnapshot(*args,**

**Bases:** luigi.task.ExternalTask
Abstract class containing a helper method to fetch the latest snapshot.

Example:

```python
class MyTask(luigi.Task):
    def requires(self):
        return PlaylistContent.latest()
```

All tasks subclassing ExternalDailySnapshot must have a luigi.DateParameter named date.
You can also provide additional parameters to the class and also configure lookback size.

Example:
date = DateParameter

classmethod latest (*args, **kwargs)
    This is cached so that requires() is deterministic.

**luigi.contrib.external_program module**

Template tasks for running external programs as luigi tasks.

This module is primarily intended for when you need to call a single external program or shell script, and it’s enough to specify program arguments and environment variables.

If you need to run multiple commands, chain them together or pipe output from one command to the next, you’re probably better off using something like `plumbum`, and wrapping plumbum commands in normal luigi `Task` s.

```python
class luigi.contrib.external_program.ExternalProgramTask (*args, **kwargs)
    Bases: luigi.task.Task

    Template task for running an external program in a subprocess
    The program is run using `subprocess.Popen`, with `args` passed as a list, generated by `program_args()` (where the first element should be the executable). See `subprocess.Popen` for details.

    Your must override `program_args()` to specify the arguments you want, and you can optionally override `program_environment()` if you want to control the environment variables (see `ExternalPythonProgramTask` for an example).

    By default, the output (stdout and stderr) of the run external program is being captured and displayed after the execution has ended. This behaviour can be overridden by passing `--capture-output False`

    `capture_output = Insignificant BoolParameter (defaults to True)`

    `stream_for_searching_tracking_url = Insignificant ChoiceParameter (defaults to none):`
        Used for defining which stream should be tracked for URL, may be set to ‘stdout’, ‘stderr’ or ‘none’.

        Default value is ‘none’, so URL tracking is not performed.

    `tracking_url_pattern = Insignificant OptionalParameter (defaults to None):`
        Regex pattern used for searching URL in the logs of the external program.

        If a log line matches the regex, the first group in the matching is set as the tracking URL for the job in the web UI. Example: ‘Job UI is here: (https?://.*)’.

        Default value is None, so URL tracking is not performed.

    `program_args()`
        Override this method to map your task parameters to the program arguments

        Returns list to pass as `args` to `subprocess.Popen`

    `program_environment()`
        Override this method to control environment variables for the program

        Returns dict mapping environment variable names to values

    `always_log_stderr`
        When True, stderr will be logged even if program execution succeeded

        Override to False to log stderr only when program execution fails.
```
Luigi Documentation, Release 2.8.13

**build_tracking_url** *(logs_output)*
This method is intended for transforming pattern match in logs to an URL:
:code:`param logs_output: Found match of self.tracking_url_pattern`
:code:`return: a tracking URL for the task`

**run()**
The task run method, to be overridden in a subclass.

See Task.run

```python
class luigi.contrib.external_program.ExternalProgramRunContext (proc)
Bases: object
```

```python
kill_job (captured_signal=None, stack_frame=None)
```

```python
exception luigi.contrib.external_program.ExternalProgramRunError (message,
args, env=None, std-out=None, stderr=None)
Bases: exceptions.RuntimeError
```

```python
class luigi.contrib.external_program.ExternalPythonProgramTask (*args,
**kwargs)
Bases: luigi.contrib.external_program.ExternalProgramTask
```

Template task for running an external Python program in a subprocess.

Simple extension of ExternalProgramTask, adding two luigi.parameter.Parameter
for setting a virtualenv and for extending the PYTHONPATH.

```python
virtualenv = Parameter (defaults to None): path to the virtualenv directory to use. It should
extra_pythonpath = Parameter (defaults to None): extend the search path for modules by
```

```python
program_environment ()
```

Override this method to control environment variables for the program.

*Returns* dict mapping environment variable names to values

**luigi.contrib.ftp module**

This library is a wrapper of ftplib or pysftp. It is convenient to move data from/to (S)FTP servers.

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 does not provide secure communication.

```python
class luigi.contrib.ftp.RemoteFileSystem (host, username=None, password=None,
port=None, tls=False, timeout=60, sftp=False, pysftp_conn_kwargs=None)
Bases: luigi.target.FileSystem
```

```python
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.

```python
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, atomic=True)`
Put file from local filesystem to (s)FTP.

`get (path, local_path)`
Download file from (s)FTP to local filesystem.

`listdir (path= '.')`
Gets a list of the contents of path in (s)FTP.

**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 ()`

**class** **luigi.contrib.ftp.RemoteTarget (path, host, format=None, username=None, password=None, port=None, mtime=None, tls=False, timeout=60, sftp=False, pysftp_conn_kwargs=None)**

Bases: **luigi.target.FileSystemTarget**

Target used for reading from remote files. The target is implemented using intermediate files on the local system. On Python2, these files may not be cleaned up.

`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.

`put (local_path, atomic=True)`

`get (local_path)`

**luigi.contrib.gcp module**

Common code for GCP (google cloud services) integration
luigi.contrib.gcp.get_authenticate_kwargs(oauth_credentials=None, http_=None)
Returns a dictionary with keyword arguments for use with discovery
Prioritizes oauth_credentials or a http client provided by the user. If none provided, falls back to default credentials provided by google's command line utilities. If that also fails, tries using httpclient.Http()
Used by gcs.GCSClient and bigquery.BigQueryClient to initiate the API Client

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, chunksize=10485760, **discovery_build_kwargs)
    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 google-auth credentials object. e.g. to use a service account:
    credentials = google.auth.jwt.Credentials.from_service_account_info(
        '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=credentials)
    The chunksize parameter specifies how much data to transfer when downloading
    or uploading files.

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

Parameters path (str) – a path within the FileSystem to check for existence.

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.
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** *(filename, dest_path, mimetype=None, chunksize=None)*

**put_multiple** *(filepaths, remote_directory, mimetype=None, chunksize=None, num_process=1)*

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

**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.

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

Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only single file copying but S3Client copies either a file or a directory as required.

**rename** *(*args, **kwargs)*

Alias for move()

**move** *(source_path, destination_path)*

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

**listdir** *(path)*

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

**list_wildcard** *(wildcard_path)*

Yields full object URIs matching the given wildcard.

 Currently only the '*' wildcard after the last path delimiter is supported.

(If we need “full” wildcard functionality we should bring in gsutil dependency with its https://github.com/GoogleCloudPlatform/gsutil/blob/master/gslib/wildcard_iterator.py…)

**download** *(path, chunksize=None, chunk_callback=<function <lambda>>)*

Downloads the object contents to local file system.

 Optionally stops after the first chunk for which chunk_callback returns True.

**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')*

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. Using *b* is not supported; initialize with *format=Nop* instead.

```python
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 GCSFlagTarget.

**Parameters**

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

```python
fs = None
exists()
Returns True if the path for this FileSystemTarget exists; False otherwise.
```

This method is implemented by using *fs*.

---

**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.

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

Bases: *luigi.task.Config*

- **pool** = OptionalParameter (defaults to None): Hadoop pool so use for Hadoop tasks. To specify pools per tasks, see *BaseHadoopJobTask.pool*

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

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

```python
luigi.contrib.hadoop.dereference(f)
```

```python
luigi.contrib.hadoop.get_extra_files(extra_files)
```

```python
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.flatten(sequence)
A simple generator which flattens a sequence.
Only one level is flattened.

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

class luigi.contrib.hadoop.HadoopRunContext
    Bases: object
    kill_job(captured_signal=None, stack_frame=None)

exception luigi.contrib.hadoop.HadoopJobError(message, out=None, err=None)
    Bases: exceptions.RuntimeError

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.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

class luigi.contrib.hadoop.JobRunner
    Bases: object
    run_job = NotImplemented

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

run_job(job, tracking_url_callback=None)
    finish()

class luigi.contrib.hadoop.DefaultHadoopJobRunner
    Bases: luigi.contrib.hadoop.HadoopJobRunner
The default job runner just reads from config and sets stuff.
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

    sample(input_stream, n, output)
    group(input_stream)
    run_job(job)

class luigi.contrib.hadoop.BaseHadoopJobTask(*args, **kwargs)
    Bases: luigi.task.Task

    pool = Insignificant OptionalParameter (defaults to None)
    batch_counter_default = 1
    final_mapper = NotImplemented
    final_combiner = NotImplemented
    final_reducer = NotImplemented
    mr_priority = NotImplemented
    package_binary = None
    task_id = None
    job_runner()
    jobconf()
    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.
    init_hadoop()
    data_interchange_format = 'python'
    run()
    The task run method, to be overridden in a subclass.
    See Task.run
    requires_local()
    Default impl - override this method if you need any local input to be accessible in init().
    requires_hadoop()
    input_local()
    input_hadoop()
    deps()
    Internal method used by the scheduler.
    Returns the flattened list of requires.
**on_failure** *(exception)*

Override for custom error handling.

This method gets called if an exception is raised in *run*. The returned 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.

```python
class luigi.contrib.hadoop.JobTask(*args, **kwargs)
Bases: luigi.contrib.hadoop.BaseHadoopJobTask

jobconf_truncate = 20000
n_reduce_tasks = 25
reducer = NotImplemented
jobconfs()
init_mapper()
init_combiner()
init.reducer()

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).

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.

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.

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.

combiner = NotImplemented
incr_counter(*args, **kwargs)*
Increments a Hadoop counter.

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

extra_modules()
extra_files()
Can be overridden 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_streaming_arguments()
    Extra arguments to Hadoop command line. Return here a list of (parameter, value) tuples.
extra_archives()
    List of paths to archives
add_link(src, dst)
dump(directory="")
    Dump instance to file.
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.
run_combiner(stdin=<open file '<stdin>', mode 'r'>, stdout=<open file '<stdout>', mode 'w'>)
internal_reader(input_stream)
    Reader which uses python eval on each part of a tab separated string. Yields a tuple of python objects.
internal_writer(outputs, stdout)
    Writer which outputs the python repr for each item.

**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, tracking_url_callback=None)

**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.
    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>).

---

### 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.get_hive_warehouse_location()

luigi.contrib.hive.get_ignored_file_masks()

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)
        Returns location of db.table (or db.table.partition). partition is a dict of partition key to value.
    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.
    table_schema (table, database='default')
        Returns list of [(name, type)] for each column in database.table.
    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')
Returns list of [(name, type)] for each column in database.table.

class luigi.contrib.hive.MetastoreClient
Bases: luigi.contrib.hive.HiveClient

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_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.

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

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

class luigi.contrib.hive.HiveThriftContext
Bases: object

Context manager for hive metastore client.

class luigi.contrib.hive.WarehouseHiveClient (hdfs_client=None, warehouse_location=None)
Bases: luigi.contrib.hive.HiveClient

Client for managed tables that makes decision based on presence of directory in hdfs

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

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_exists (table, database='default', partition=None)
The table/partition is considered existing if corresponding path in hdfs exists and contains file except those
which match pattern set in ignored_file_masks

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

luigi.contrib.hive.get_default_client ()

class luigi.contrib.hive.HiveQueryTask (*args, **kwargs)
Bases: luigi.contrib.hadoop.BaseHadoopJobTask

Task to run a hive query.

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 luigi.cfg, will default to the value there otherwise returns None.

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

**hivevars()**
Returns a dict of key=value settings to be passed along to the hive command line via –hivevar. This option can be used as a separated namespace for script local variables. See [https://cwiki.apache.org/confluence/display/Hive/LanguageManual+VariableSubstitution](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+VariableSubstitution)

**hiveconsfs()**
Returns a dict of key=value settings to be passed along to the hive command line via –hiveconf. By default, sets `mapred.reduce.tasks (n_reduce_tasks)`
- `mapred.fairscheduler.pool (pool)` or `mapred.job.queue.name (pool)`
- `hive.execreducers.bytes.per.reducer (bytes_per_reducer)`
- `hive.execreducers.max (reducers_max)`

**job_runner()**

```python
class luigi.contrib.hive.HiveQueryRunner
    Bases: luigi.contrib.hadoop.JobRunner

    Runs a HiveQueryTask by shell 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.

**get_arglist(f_name, job)**

**run_job(job, tracking_url_callback=None)**

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

    Bases: luigi.target.Target

    Target representing Hive table or Hive partition
```

@param table: Table name @type table: str @param partition: partition specificaton in form of dict of {“partition_column_1”: “partition_value_1”, “partition_column_2”: “partition_value_2”, . . . } If `partition` is `None` or `[]` then target is Hive nonpartitioned table @param database: Database name @param fail_missing_table: flag to ignore errors raised due to table nonexistence @param client: `HiveCommandClient` instance. Default if `client` is `None`

**exists()**
returns `True` if the partition/table exists

**path**
Returns the path for this HiveTablePartitionTarget’s data.

```python
class luigi.contrib.hive.HiveTableTarget
    table, database='default', client=None)

    Bases: luigi.contrib.hive.HivePartitionTarget

    Target representing non-partitioned table
```

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

Bases: `luigi.task.ExternalTask`

External task that depends on a Hive table/partition.
database = Parameter (defaults to default)

table = Parameter

partition = DictParameter (defaults to {}): Python dictionary specifying the target partition e.g. "{"date": "2013-01-25"}

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

luigi.contrib.kubernetes module

Kubernetes Job wrapper for Luigi.

From the Kubernetes website:

   Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

   For more information about Kubernetes Jobs: http://kubernetes.io/docs/user-guide/jobs/

   Requires:

   • pykube: pip install pykube-ng

   Written and maintained by Marco Capuccini (@mcapuccini).

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

   auth_method = Parameter (defaults to kubeconfig): Authorization method to access the cluster

   kubeconfig_path = Parameter (defaults to ~/.kube/config): Path to kubeconfig file for cluster authentication

   max_retrials = IntParameter (defaults to 0): Max retrials in event of job failure

   kubernetes_namespace = OptionalParameter (defaults to None): K8s namespace in which the job will run

class luigi.contrib.kubernetes.KubernetesJobTask (*args, **kwargs)
   Bases: luigi.task.Task

   auth_method
      This can be set to kubeconfig or service-account. It defaults to kubeconfig.

      For more details, please refer to:

      • kubeconfig: http://kubernetes.io/docs/user-guide/kubeconfig-file

      • service-account: http://kubernetes.io/docs/user-guide/service-accounts

   kubeconfig_path
      Path to kubeconfig file used for cluster authentication. It defaults to "~/.kube/config", which is the default location when using minikube (http://kubernetes.io/docs/getting-started-guides/minikube). When auth_method is service-account this property is ignored.

      WARNING: For Python versions < 3.5 kubeconfig must point to a Kubernetes API hostname, and NOT to an IP address.
For more details, please refer to: http://kubernetes.io/docs/user-guide/kubeconfig-file

**kubernetes_namespace**
Namespace in Kubernetes where the job will run. It defaults to the default namespace in Kubernetes
For more details, please refer to: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

**name**
A name for this job. This task will automatically append a UUID to the name before to submit to Kubernetes.

**labels**
Return custom labels for kubernetes job. example::
```python
datetime.date.today().strftime('%F')
```

**spec_schema**
Kubernetes Job spec schema in JSON format, an example follows.
```json
{
    "containers": [{
        "name": "pi",
        "image": "perl",
        "command": ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
    }],
    "restartPolicy": "Never"
}
```

**restartPolicy**
- If restartPolicy is not defined, it will be set to “Never” by default.
- **Warning**: restartPolicy=OnFailure will bypass max_retrials, and restart the container until success, with the risk of blocking the Luigi task.

For more informations please refer to: http://kubernetes.io/docs/user-guide/pods/multi-container/#the-spec-schema

**max_retrials**
Maximum number of retrials in case of failure.

**backoff_limit**
Maximum number of retries before considering the job as failed. See: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#pod-backoff-failure-policy

**delete_on_success**
Delete the Kubernetes workload if the job has ended successfully.

**print_pod_logs_on_exit**
Fetch and print the pod logs once the job is completed.

**active_deadline_seconds**
Time allowed to successfully schedule pods. See: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/#job-termination-and-cleanup

**kubernetes_config**

**poll_interval**
How often to poll Kubernetes for job status, in seconds.

**pod_creation_wait_interval**
Delay for initial pod creation for just submitted job in seconds
signal_complete()  
Signal job completion for scheduler and dependent tasks.

Touching a system file is an easy way to signal completion. example::  
```python
with self.output().open('w') as output_file: output_file.write('')
```

run()  
The task run method, to be overridden in a subclass.

See Task.run

output()  
An output target is necessary for checking job completion unless an alternative complete method is defined.

Example:
```python
return luigi.LocalTarget(os.path.join('/tmp', 'example'))
```

**luigi.contrib.lsf module**

luigi.contrib.lsf.track_job(job_id)  
Tracking is done by requesting each job and then searching for whether the job has one of the following states:  
- "RUN", - "PEND", - "SSUSP", - "EXIT" based on the LSF documentation

luigi.contrib.lsf.kill_job(job_id)  
Kill a running LSF job

class luigi.contrib.lsf.LSFJobTask(*args, **kwargs)  
Bases: luigi.task.Task

Takes care of uploading and executing an LSF job

n_cpu_flag = Insignificant IntParameter (defaults to 2)
shared_tmp_dir = Insignificant Parameter (defaults to /tmp)
resource_flag = Insignificant Parameter (defaults to mem=8192)
memory_flag = Insignificant Parameter (defaults to 8192)
queue_flag = Insignificant Parameter (defaults to queue_name)
runtime_flag = IntParameter (defaults to 60)
job_name_flag = Parameter (defaults to )
poll_time = Insignificant FloatParameter (defaults to 5): specify the wait time to poll
save_job_info = BoolParameter (defaults to False)
output = Parameter (defaults to )
extra_bsub_args = Parameter (defaults to )
job_status = None

fetch_task_failures()  
Read in the error file from bsub

fetch_task_output()  
Read in the output file
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 compute nodes.

run()
The procedure: - Pickle the class - Tarball the dependencies - Construct a bsub argument that runs a generic runner function with the path to the pickled class - Runner function loads the class from pickle - Runner class untars the dependencies - Runner function hits the button on the class’s work() method

work()
Subclass this for where you’re doing your actual work.
Why not run(), like other tasks? Because we need run to always be something that the Worker can call, and that’s the real logical place to do LSF scheduling. So, the work will happen in work().

class luigi.contrib.lsf.LocalLSFJobTask(*args, **kwargs)
Bases: luigi.contrib.lsf.LSFJobTask
A local version of JobTask, for easier debugging.

run()
The procedure: - Pickle the class - Tarball the dependencies - Construct a bsub argument that runs a generic runner function with the path to the pickled class - Runner function loads the class from pickle - Runner class untars the dependencies - Runner function hits the button on the class’s work() method

luigi.contrib.lsf_runner module

luigi.contrib.lsf_runner.do_work_on_compute_node(work_dir)
luigi.contrib.lsf_runner.extract_packages_archive(work_dir)
luigi.contrib.lsf_runner.main(args=['/home/docs/checkouts/readthedocs.org/user_builds/luigi/envs/latest/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '': '_build/latex'])
Run the work() method from the class instance in the file “job-instance.pickle”.

luigi.contrib.mongodb module

class luigi.contrib.mongodb.MongoTarget(mongo_client, index, collection)
Bases: luigi.target.Target
Target for a resource in MongoDB

Parameters

• mongo_client (MongoClient) – MongoDB instance
• index (str) – database index
• collection (str) – index collection

get_collection()
Return targeted mongo collection to query on

get_index()
Return targeted mongo index to query on
class luigi.contrib.mongodb.MongoCellTarget(mongo_client, index, collection, document_id, path)
Bases: luigi.contrib.mongodb.MongoTarget

Target for a resource in a specific field from a MongoDB document

Parameters

• **document_id** (str) – targeted mongo document
• **path** (str) – full path to the targeted field in the mongo document

exists()
Test if target has been run Target is considered run if the targeted field exists

read()
Read the target value Use $project aggregate operator in order to support nested objects

write(value)
Write value to the target

class luigi.contrib.mongodb.MongoRangeTarget(mongo_client, index, collection, document_ids, field)
Bases: luigi.contrib.mongodb.MongoTarget

Target for a level 0 field in a range of documents

Parameters

• **document_ids** – targeted mongo documents
• **field** (str) – targeted field in documents

exists()
Test if target has been run Target is considered run if the targeted field exists in ALL documents

read()
Read the targets value

write(values)
Write values to the targeted documents Values need to be a dict as: {document_id: value}

get_empty_ids()
Get documents id with missing targeted field

class luigi.contrib.mongodb.MongoCollectionTarget(mongo_client, index, collection)
Bases: luigi.contrib.mongodb.MongoTarget

Target for existing collection

exists()
Test if target has been run Target is considered run if the targeted collection exists in the database

read()
Return if the target collection exists in the database

class luigi.contrib.mongodb.MongoCountTarget(mongo_client, index, collection, target_count)
Bases: luigi.contrib.mongodb.MongoTarget

Target for documents count

Parameters **target_count** – Value of the desired item count in the target
exists()
    Test if the target has been run. Target is considered run if the number of items in the target matches the value
    of self._target_count

read()
    Using the aggregate method to avoid inaccurate count if using a sharded cluster. See
    https://docs.mongodb.com/manual/reference/method/db.collection.count/#behavior

luigi.contrib.mrrunner module

Since after Luigi 2.5.0, this is a private module to Luigi. Luigi users should not rely on importing this module
works. Furthermore, “luigi mr streaming” have been greatly superseeded by technoligies like Spark, Hive, etc.

The hadoop runner.

This module contains the main() method which will be used to run the mapper, combiner, or reducer on the Hadoop
nodes.

class luigi.contrib.mrrunner.Runner(job=None)
    Bases: object
    Run the mapper, combiner, or reducer on hadoop nodes.
    
    run(kind, stdin=<open file '<stdin>' mode 'r'>, stdout=<open file '<stdout>' mode 'w'>)
    extract_packages_archive()
luigi.contrib.mrrunner.print_exception(exc)

luigi.contrib.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, combiner, or reducer from the class instance in the file “job-instance.pickle”.
    
    Arguments:
        kind – is either map, combiner, or reduce

luigi.contrib.mssqldb module

class luigi.contrib.mssqldb.MSSqlTarget(host, database, user, password, table, update_id)
    Bases: luigi.target.Target
    Target for a resource in Microsoft SQL Server. This module is primarily derived from mysqldb.py. Much of
    MSSqlTarget, MySqlTarget and PostgresTarget are similar enough to potentially add a RDBMSTarget abstract
    base class to rdbms.py that these classes could be derived from.

    Initializes a MsSqlTarget instance.

    Parameters
        • host(str) – MsSql 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)
Returns True if the Target exists and False otherwise.

connect ()
Create a SQL Server connection and return a connection object

create_marker_table ()
Create marker table if it doesn’t exist. Use a separate connection since the transaction might have to be reset.

luigi.contrib.mysqldb module

class luigi.contrib.mysqldb.MySqlTarget (host, database, user, password, table, update_id, **cnx_kwargs)
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)
Returns True if the Target exists and False otherwise.

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.

class luigi.contrib.mysqldb.CopyToTable (*args, **kwargs)
Bases: luigi.contrib.rdbms.CopyToTable
Template task for inserting a data set into MySQL.
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`.

`rows()`
Return/yield tuples or lists corresponding to each row to be inserted.

`output()`
Returns a MySqlTarget representing the inserted dataset.
Normally you don’t override this.

`copy(cursor, file=None)`

`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.

`bulk_size`

**luigi.contrib.opener module**

OpenerTarget support, allows easier testing and configuration by abstracting out the LocalTarget, S3Target, and MockTarget types.

Example:

```python
from luigi.contrib.opener import OpenerTarget
OpenerTarget('/local/path.txt')
OpenerTarget('s3://zefr/remote/path.txt')
```

**exception luigi.contrib.opener.OpenerError**
Bases: luigi.target.FileSystemException

The base exception thrown by openers

**exception luigi.contrib.opener.NoOpenerError**
Bases: luigi.contrib.opener.OpenerError

Thrown when there is no opener for the given protocol

**exception luigi.contrib.opener.InvalidQuery**
Bases: luigi.contrib.opener.OpenerError

Thrown when an opener is passed unexpected arguments

**class luigi.contrib.opener.OpenerRegistry(openers=None)**
Bases: object

An opener registry that stores a number of opener objects used to parse Target URIs

Parameters `openers` (list) – A list of objects inherited from the Opener class.

`get_opener(name)`
Retrieve an opener for the given protocol

Parameters `name` (string) – name of the opener to open

Raises `NoOpenerError` – if no opener has been registered of that name
**add** *(opener)*

Adds an opener to the registry

**Parameters**

- **opener** *(Opener inherited object)* - Opener object

**open** *(target_uri, **kwargs)*

Open target uri.

**Parameters**

- **target_uri** *(string)* - Uri to open

**Returns**

Target object

class luigi.contrib.opener.Opener

Bases: object

Base class for Opener objects.

- **allowed_kwarg** = {}

- **filter_kwarg** = True

**classmethod conform_query** *(query)*

Converts the query string from a target uri, uses cls.allowed_kwarg, and cls.filter_kwarg to drive logic.

**Parameters**

- **query** *(urllib.parse.unsplit(uri)query)* - Unparsed query string

**Returns**

Dictionary of parsed values, everything in cls.allowed_kwarg with values set to True will be parsed as json strings.

**classmethod get_target** *(scheme, path, fragment, username, password, hostname, port, query, **kwargs)*

Override this method to use values from the parsed uri to initialize the expected target.

class luigi.contrib.opener.MockOpener

Bases: luigi.contrib.opener.Opener

Mock target opener, works like LocalTarget but files are all in memory.

example: * mock://foo/bar.txt

- **names** = ['mock']

- **allowed_kwarg** = {'format': False, 'is_tmp': True, 'mirror_on_stderr': True}

**classmethod get_target** *(scheme, path, fragment, username, password, hostname, port, query, **kwargs)*

Override this method to use values from the parsed uri to initialize the expected target.

class luigi.contrib.opener.LocalOpener

Bases: luigi.contrib.opener.Opener

Local filesystem opener, works with any valid system path. This is the default opener and will be used if you don’t indicate which opener.

examples: * file://relative/foo/bar/baz.txt (opens a relative file) * file:///home/user (opens a directory from an absolute path) * foo/bar.baz (file:// is the default opener)

- **names** = ['file']

- **allowed_kwarg** = {'format': False, 'is_tmp': True}

**classmethod get_target** *(scheme, path, fragment, username, password, hostname, port, query, **kwargs)*

Override this method to use values from the parsed uri to initialize the expected target.
class luigi.contrib.opener.S3Opener

Bases: luigi.contrib.opener.Opener

Opens a target stored on Amazon S3 storage

examples: * s3://bucket/foo/bar.txt * s3://bucket/foo/bar.txt?aws_access_key_id=xxx&aws_secret_access_key=yyy

names = ['s3', 's3n']

allowed_kwargs = {'client': True, 'format': False}

filter_kwargs = False

classmethod get_target(scheme, path, fragment, username, password, hostname, port, query, **kwargs)

Override this method to use values from the parsed uri to initialize the expected target.

luigi.contrib.pai module

MicroSoft OpenPAI Job wrapper for Luigi.

“OpenPAI is an open source platform that provides complete AI model training and resource management capabilities, it is easy to extend and supports on-premise, cloud and hybrid environments in various scale.”

For more information about OpenPAI: https://github.com/Microsoft/pai/, this task is tested against OpenPAI 0.7.1

Requires:

• requests: pip install requests

Written and maintained by Liu, Dongqing (@liudongqing).

luigi.contrib.pai.slot_to_dict(o)

class luigi.contrib.pai.PaiJob(jobName, image, tasks)

Bases: object

The Open PAI job definition. Refer to here https://github.com/Microsoft/pai/blob/master/docs/job_tutorial.md

```python
{
    "jobName": String,
    "image": String,
    "authFile": String,
    "dataDir": String,
    "outputDir": String,
    "codeDir": String,
    "virtualCluster": String,
    "taskRoles": [
        {
            "name": String,
            "taskNumber": Integer,
            "cpuNumber": Integer,
            "memoryMB": Integer,
            "shmMB": Integer,
            "gpuNumber": Integer,
            "portList": [
                {
                    "label": String,
                    "beginAt": Integer,
                    "portNumber": Integer
                }
            ]
        }
    ]
}
```

(continues on next page)
Initialize a Job with required fields.

**Parameters**

- **jobName** – Name for the job, need to be unique
- **image** – URL pointing to the Docker image for all tasks in the job
- **tasks** – List of taskRole, one task role at least

```python

```

```python

```
- **name** – Name for the task role, need to be unique with other roles, required
- **command** – Executable command for tasks in the task role, can not be empty, required
- **taskNumber** – Number of tasks for the task role, no less than 1, required
- **cpuNumber** – CPU number for one task in the task role, no less than 1, required
- **shmMB** – Shared memory for one task in the task role, no more than memory size, required
- **memoryMB** – Memory for one task in the task role, no less than 100, required
- **gpuNumber** – GPU number for one task in the task role, no less than 0, required
- **portList** – List of portType to use, optional

```python
def class luigi.contrib.pai.OpenPai(*args, **kwargs)
    Bases: luigi.task.Config
    pai_url = Parameter (defaults to http://127.0.0.1:9186): rest server url, default is http://127.0.0.1:9186
    username = Parameter (defaults to admin): your username
    password = Parameter (defaults to None): your password
    expiration = IntParameter (defaults to 3600): expiration time in seconds

def class luigi.contrib.pai.PaiTask(*args, **kwargs)
    Bases: luigi.task.Task
    Parameters
    - **pai_url** – The rest server url of PAI clusters, default is http://127.0.0.1:9186.
    - **token** – The token used to auth the rest server of PAI.
```

- **name**
  - Name for the job, need to be unique, required

- **image**
  - URL pointing to the Docker image for all tasks in the job, required

- **tasks**
  - List of taskRole, one task role at least, required

- **auth_file_path**
  - Docker registry authentication file existing on HDFS, optional
data_dir
   Data directory existing on HDFS, optional

code_dir
   Code directory existing on HDFS, should not contain any data and should be less than 200MB, optional

output_dir
   Output directory on HDFS, $PAI_DEFAULT_FS_URI/$jobName/output will be used if not specified, optional

virtual_cluster
   The virtual cluster job runs on. If omitted, the job will run on default virtual cluster, optional

gpu_type
   Specify the GPU type to be used in the tasks. If omitted, the job will run on any gpu type, optional

retry_count
   Job retry count, no less than 0, optional

run()
   The task run method, to be overridden in a subclass.
   See Task.run

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

complete()
   If the task has any outputs, return True if all outputs exist. Otherwise, return False.
   However, you may freely override this method with custom logic.

luigi.contrib.pig module

Apache Pig support. Example configuration section in luigi.cfg:

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

class luigi.contrib.pig.PigJobTask(*args, **kwargs)
   Bases: luigi.task.Task

   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’ }

124 Chapter 9. API Reference
**pig_properties()**
Dictionary of properties that should be set when running Pig.

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

**pig_parameters()**
Dictionary of parameters that should be set for the Pig job.

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

**pig_options()**
List of options that will be appended to the Pig command.

Example:
```python
return ['-x', 'local']
```

**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 if 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`

**pig_script_path()**
Return the path to the Pig script to be run.

**run()**
The task run method, to be overridden in a subclass.

See `Task.run`

**track_and_progress(cmd)**

**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.postgres module**

Implements a subclass of `Target` that writes data to Postgres. Also provides a helper task to copy data into a Postgres table.

**class** `luigi.contrib.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:

```python
>>> 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.contrib.postgres.PostgresTarget (host, database, user, password, table, update_id, port=None)

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
- port (int): Postgres server port.

**marker_table** = 'table_updates'

**DEFAULT_DB_PORT** = 5432

**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)

Returns True if the Target exists and False otherwise.

**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.contrib.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`.

```python
rows()
```

Return/yield tuples or lists corresponding to each row to be inserted.

```python
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`.

```python
output()
```

Returns a PostgresTarget representing the inserted dataset.

Normally you don’t override this.

```python
copy(cursor, file)
```

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.

```python
class luigi.contrib.postgres.PostgresQuery(*args, **kwargs)
   Bases: luigi.contrib.rdbms.Query
```

Template task for querying a Postgres compatible database

Usage: Subclass and override the required `host`, `database`, `user`, `password`, `table`, and `query` attributes. Optionally one can override the `autocommit` attribute to put the connection for the query in autocommit mode.

Override the `run` method if your use case requires some action with the query result.

Task instances require a dynamic `update_id`, e.g. via parameter(s), otherwise the query will only execute once.

To customize the query signature as recorded in the database marker table, override the `update_id` property.

```python
run()
```

The task run method, to be overridden in a subclass.

See `Task.run`

```python
output()
```

Returns a PostgresTarget representing the executed query.

Normally you don’t override this.

---

**luigi.contrib.presto module**

```python
class luigi.contrib.presto.presto(*args, **kwargs)
   Bases: luigi.task.Config
```

- `host` = Parameter (defaults to localhost): Presto host
- `port` = IntParameter (defaults to 8090): Presto port
- `user` = Parameter (defaults to anonymous): Presto user
- `catalog` = Parameter (defaults to hive): Default catalog
- `password` = Parameter (defaults to None): User password
- `protocol` = Parameter (defaults to https): Presto connection protocol
poll_interval = FloatParameter (defaults to 1.0): how often to ask the Presto REST interface

class luigi.contrib.presto.PrestoClient (connection, sleep_time=1)
    Helper class wrapping pyhive.presto.Connection for executing presto queries and tracking progress

    percentage_progress
        Returns percentage of query overall progress

    info_uri
        Returns query UI link

    execute(query, parameters=None, mode=None)
        Parameters
            • query – query to run
            • parameters – parameters should be injected in the query
            • mode – “fetch” - yields rows, “watch” - yields log entries
        Returns

class luigi.contrib.presto.WithPrestoClient
    Bases: luigi.task_register.Register

    A metaclass for injecting PrestoClient as a _client field into a new instance of class T Presto connection options
    are taken from T-instance fields Fields should have the same names as in pyhive.presto.Cursor

class luigi.contrib.presto.PrestoTarget (client, catalog, database, table, partition=None)
    Bases: luigi.target.Target

    Target for presto-accessible tables

    count()

    exists()
        Returns True if given table exists and there are any rows in a given partition False if no rows in
        the partition exists or table is absent

class luigi.contrib.presto.PrestoTask (*args, **kwargs)
    Bases: luigi.contrib.rdbms.Query

    Task for executing presto queries During its executions tracking url and percentage progress are set

    host
    port
    user
    username
    schema
    password
    catalog
    poll_interval
    source
    partition
    protocol
session_props
requests_session
requests_kwargs
query = None
run()
The task run method, to be overridden in a subclass.
See Task.run
output()
Override with an RDBMS Target (e.g. PostgresTarget or RedshiftTarget) to record execution in a marker table

luigi.contrib.prometheus_metric module

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.SparkContextEntryPoint (conf)
   Bases: luigi.contrib.pyspark_runner._SparkEntryPoint
      sc = None
class luigi.contrib.pyspark_runner.SparkSessionEntryPoint (conf)
   Bases: luigi.contrib.pyspark_runner._SparkEntryPoint
      spark = None
class luigi.contrib.pyspark_runner.AbstractPySparkRunner (job, *args)
   Bases: object
      run()
class luigi.contrib.pyspark_runner.PySparkRunner (job, *args)
   Bases: luigi.contrib.pyspark_runner.AbstractPySparkRunner
class luigi.contrib.pyspark_runner.PySparkSessionRunner (job, *args)
   Bases: luigi.contrib.pyspark_runner.AbstractPySparkRunner

luigi.contrib.rdbms module

A common module for postgres like databases, such as postgres or redshift
class luigi.contrib.rdbms.CopyToTable (*args, **kwargs)
   Bases:       luigi.task.MixinNaiveBulkComplete,       luigi.contrib.rdbms.
               _MetadataColumnsMixin, 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
• port

host
database
user
password
table
port
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()
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

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.

post_copy (connection)
Override to perform custom queries.

Any code here will be formed in the same transaction as the main copy, just after copying data. Example
use cases include cleansing data in temp table prior to insertion into real table.
```
copy (cursor, file)

class luigi.contrib.rdbms.Query(*args, **kwargs)
    Bases: luigi.task.MixinNaiveBulkComplete, luigi.task.Task

An abstract task for executing an RDBMS query.

Usage:

Subclass and override the following attributes:

• `host`
• `database`
• `user`
• `password`
• `table`
• `query`

Optionally override:

• `port`
• `autocommit`
• `update_id`

Subclass and override the following methods:

• `run`
• `output`

`host`
Host of the RDBMS. Implementation should support `hostname:port` to encode port.

`port`
Override to specify port separately from host.

`database`

`user`

`password`

`table`

`query`

`autocommit`

`update_id`
Override to create a custom marker table ‘update_id’ signature for Query subclass task instances

`run()`
The task run method, to be overridden in a subclass.

    See Task.run

`output()`
Override with an RDBMS Target (e.g. PostgresTarget or RedshiftTarget) to record execution in a marker table
```
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 = Parameter (defaults to luigi)

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, port=None)

Bases: luigi.contrib.postgres.PostgresTarget

Target for a resource in Redshift.

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

Args:

marker_table = 'table_updates'

DEFAULT_DB_PORT = 5439

use_db_timestamps = False

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

Bases: luigi.contrib.rdbms.CopyToTable, luigi.contrib.redshift._CredentialsMixin

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

Usage:
• Subclass and override the required attributes:
  – host,
  – database,
  – user,
  – password,
  – table,
  – columns,
  – s3_load_path.

• You can also override the attributes provided by the CredentialsMixin if they are not supplied by your configuration or environment variables.

`s3_load_path()`
Override to return the load path.

copy_options
Add extra copy options, for example:
  • TIMEFORMAT 'auto'
  • IGNOREHEADER 1
  • TRUNCATECOLUMNS
  • IGNOREBLANKLINES
  • DELIMITER ' ' 

prune_table
Override to set equal to the name of the table which is to be pruned. Intended to be used in conjunction with prune_column and prune_date i.e. copy to temp table, prune production table to prune_column with a date greater than prune_date, then insert into production table from temp table

prune_column
Override to set equal to the column of the prune_table which is to be compared. Intended to be used in conjunction with prune_table and prune_date i.e. copy to temp table, prune production table to prune_column with a date greater than prune_date, then insert into production table from temp table

prune_date
Override to set equal to the date by which prune_column is to be compared. Intended to be used in conjunction with prune_table and prune_column i.e. copy to temp table, prune production table to prune_column with a date greater than prune_date, then insert into production table from temp table

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

table_constraints
Add extra table constraints, for example:
  PRIMARY KEY (MY_FIELD, MY_FIELD_2) UNIQUE KEY (MY_FIELD_3)

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

do_prune()
Return True if prune_table, prune_column, and prune_date are implemented. If only a subset of prune
variables are override, an exception is raised to remind the user to implement all or none. Prune (data newer than prune_date deleted) before copying new data in.

**table_type**
Return table type (i.e. ‘temp’).

**queries**
Override to return a list of queries to be executed in order.

**truncate_table** *(connection)*

**prune** *(connection)*

**create_schema** *(connection)*
Will create the schema in the database

**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.

  If both key-based and role-based credentials are provided, role-based will be used.

**output** ()
Returns a RedshiftTarget representing the inserted dataset.

  Normally you don’t override this.

**does_schema_exist** *(connection)*
Determine whether the schema already exists.

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

**init_copy** *(connection)*
Perform pre-copy sql - such as creating table, truncating, or removing data older than x.

**post_copy** *(cursor)*
Performs post-copy sql - such as cleansing data, inserting into production table (if copied to temp table), etc.

**post_copy_metacolumns** *(cursor)*
Performs post-copy to fill metadata columns.

**class** `luigi.contrib.redshift.S3CopyJSONToTable(*args, **kwargs)`

  `Bases: luigi.contrib.redshift.S3CopyToTable, luigi.contrib.redshift._CredentialsMixin`

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

Usage:
- Subclass and override the required attributes:
  - `host`,
  - `database`,
You can also override the attributes provided by the CredentialsMixin if they are not supplied by your configuration or environment variables.

**jsonpah**
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.

```python
class luigi.contrib.redshift.RedshiftManifestTask(*args, **kwargs)
```

Bases: `luigi.contrib.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.


**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

```python
folder_paths = Parameter
text_target = True
```

```python
run ()
```

The task run method, to be overridden in a subclass.

See `Task.run`

```python
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.

```python
class luigi.contrib.redshift.RedshiftQuery(*args, **kwargs)
Bases: luigi.contrib.postgres.PostgresQuery
```
Template task for querying an Amazon Redshift database

Usage: Subclass and override the required `host`, `database`, `user`, `password`, `table`, and `query` attributes.

Override the `run` method if your use case requires some action with the query result.

Task instances require a dynamic `update_id`, e.g. via parameter(s), otherwise the query will only execute once

To customize the query signature as recorded in the database marker table, override the `update_id` property.

```python
class luigi.contrib.redshift.RedshiftUnloadTask(*args, **kwargs)
Bases: luigi.contrib.postgres.PostgresQuery, luigi.contrib.redshift._CredentialsMixin
```
Template task for running UNLOAD on an Amazon Redshift database

Usage: Subclass and override the required `host`, `database`, `user`, `password`, `table`, and `query` attributes. Optionally, override the `autocommit` attribute to run the query in autocommit mode - this is necessary to run VACUUM for example. Override the `run` method if your use case requires some action with the query result.

Task instances require a dynamic `update_id`, e.g. via parameter(s), otherwise the query will only execute once

To customize the query signature as recorded in the database marker table, override the `update_id` property. You can also override the attributes provided by the CredentialsMixin if they are not supplied by your configuration or environment variables.

```python
def s3_unload_path
    Override to return the load path.
```

```python
def unload_options
    Add extra or override default unload options:
```

```python
def unload_query
    Default UNLOAD command
```
run()
The task run method, to be overridden in a subclass.

See Task.run

output()
Returns a RedshiftTarget representing the executed query.
Normally you don’t override this.

**luigi.contrib.s3 module**

Implementation of Simple Storage Service support. `S3Target` is a subclass of the Target class to support S3 file system operations. The `boto3` library is required to use S3 targets.

---

**exception** luigi.contrib.s3.InvalidDeleteException
**Bases:** luigi.target.FileSystemException

**exception** luigi.contrib.s3.FileNotFoundException
**Bases:** luigi.target.FileSystemException

**exception** luigi.contrib.s3.DeprecatedBotoClientException
**Bases:** exceptions.Exception

**class** luigi.contrib.s3.S3Client(aws_access_key_id=None, aws_secret_access_key=None, aws_session_token=None, **kwargs)
**Bases:** luigi.target.FileSystem

* boto3-powered S3 client.

**DEFAULT_PART_SIZE** = 8388608

**DEFAULT_THREADS** = 100

s3

exists(path)
Does provided path exist on S3?

remove(path, recursive=True)
Remove a file or directory from S3. :param path: File or directory to remove :param recursive: Boolean indicator to remove object and children :return: Boolean indicator denoting success of the removal of 1 or more files

move(source_path, destination_path, **kwargs)
Rename/move an object from one S3 location to another. :param source_path: The s3:// path of the directory or key to copy from :param destination_path: The s3:// path of the directory or key to copy to :param kwargs: Keyword arguments are passed to the boto3 function copy

get_key(path)
Returns the object summary at the path

put(local_path, destination_s3_path, **kwargs)
Put an object stored locally to an S3 path. :param local_path: Path to source local file :param destination_s3_path: URL for target S3 location :param kwargs: Keyword arguments are passed to the boto3 function put_object

put_string(content, destination_s3_path, **kwargs)
Put a string to an S3 path. :param content: Data str :param destination_s3_path: URL for target S3 location :param kwargs: Keyword arguments are passed to the boto3 function put_object
**put_multipart** *(local_path, destination_s3_path, part_size=8388608, **kwargs)*
Put an object stored locally to an S3 path using S3 multi-part upload (for files > 8Mb).
- **local_path**: Path to source local file
- **destination_s3_path**: URL for target S3 location
- **part_size**: Part size in bytes. Default: 8388608 (8MB)
- **kwargs**: Keyword arguments are passed to the boto function `upload_fileobj` as ExtraArgs

**copy** *(source_path, destination_path, threads=100, start_time=None, end_time=None, part_size=8388608, **kwargs)*
Copy object(s) from one S3 location to another. Works for individual keys or entire directories. When files are larger than `part_size`, multipart uploading will be used.
- **source_path**: The s3:// path of the directory or key to copy from
- **destination_path**: The s3:// path of the directory or key to copy to
- **threads**: Optional argument to define the number of threads to use when copying (min: 3 threads)
- **start_time**: Optional argument to copy files with modified dates after start_time
- **end_time**: Optional argument to copy files with modified dates before end_time
- **part_size**: Part size in bytes
- **kwargs**: Keyword arguments are passed to the boto function `copy` as ExtraArgs

**get** *(s3_path, destination_local_path)*
Get an object stored in S3 and write it to a local path.

**get_as_bytes** *(s3_path)*
Get the contents of an object stored in S3 as bytes

**Parameters**
- **s3_path**: URL for target S3 location

**Returns**
File contents as pure bytes

**get_as_string** *(s3_path, encoding='utf-8')*
Get the contents of an object stored in S3 as string.

**Parameters**
- **s3_path**: URL for target S3 location
- **encoding**: Encoding to decode bytes to string

**Returns**
File contents as a string

**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)*
Create directory at location `path`

**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.

**listdir** *(path, start_time=None, end_time=None, return_key=False)*
Get an iterable with S3 folder contents. Iterable contains paths relative to queried path.
- **path**: URL for target S3 location
- **start_time**: Optional argument to list files with modified (offset aware) datetime after start_time
- **end_time**: Optional argument to list files with modified (offset aware)
datetime before end_time :param return_key: Optional argument, when set to True will return boto3’s ObjectSummary (instead of the filename)

```
list (path, start_time=None, end_time=None, return_key=False)
```

class luigi.contrib.s3.AtomicS3File(path, s3_client, **kwargs)
Bases: luigi.target.AtomicLocalFile

An S3 file that writes to a temp file and puts to S3 on close.

**Parameters**

kwargs – Keyword arguments are passed to the boto function `initialize_multipart_upload`

move_to_final_destination()

class luigi.contrib.s3.ReadableS3File(s3_key)
Bases: object

read (size=None)

close()

readable()

writable()

seekable()

class luigi.contrib.s3.S3Target(path, format=None, client=None, **kwargs)
Bases: luigi.target.FileSystemTarget

Target S3 file object

**Parameters**

kwargs – Keyword arguments are passed to the boto function `initialize_multipart_upload`

```
fs = None
```

open (mode='r')

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. Using b is not supported; initialize with format=Nop instead.

class luigi.contrib.s3.S3FlagTarget(path, format=None, client=None, flag='_SUCCESS')
Bases: luigi.contrib.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)* –

```python
fs = None
```

**exists()**

Returns `True` if the path for this FileSystemTarget exists; `False` otherwise.

This method is implemented by using `fs`.

```python
class Luigi.contrib.s3.S3EmrTarget(*args, **kwargs)
Bases: Luigi.contrib.s3.S3FlagTarget
```

Deprecated. Use `S3FlagTarget`

```python
class Luigi.contrib.s3.S3PathTask(*args, **kwargs)
Bases: Luigi.task.ExternalTask
```

A external task that to require existence of a path in S3.

```python
path = Parameter
```

**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`

```python
class Luigi.contrib.s3.S3EmrTask(*args, **kwargs)
Bases: Luigi.task.ExternalTask
```

An external task that requires the existence of EMR output in S3.

```python
path = Parameter
```

**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`

```python
class Luigi.contrib.s3.S3FlagTask(*args, **kwargs)
Bases: Luigi.task.ExternalTask
```

An external task that requires the existence of EMR output in S3.

```python
path = Parameter
flag = OptionalParameter (defaults to None)
```
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

**luigi.contrib.salesforce module**

luigi.contrib.salesforce.get_soql_fields(soql)

Gets queried columns names.

luigi.contrib.salesforce.ensure_utf(value)

luigi.contrib.salesforce.parse_results(fields, data)

Traverses ordered dictionary, calls _traverse_results() to recursively read into the dictionary depth of data

class luigi.contrib.salesforce.salesforce(*args, **kwargs)

Bases: luigi.task.Config

Config system to get config vars from 'salesforce' section in configuration file.

Did not include sandbox_name here, as the user may have multiple sandboxes.

username = Parameter (defaults to )

password = Parameter (defaults to )

security_token = Parameter (defaults to )

sb_security_token = Parameter (defaults to )

class luigi.contrib.salesforce.QuerySalesforce(*args, **kwargs)

Bases: luigi.task.Task

object_name

Override to return the SF object we are querying. Must have the SF “__c” suffix if it is a customer object.

use_sandbox

Override to specify use of SF sandbox. True iff we should be uploading to a sandbox environment instead of the production organization.

sandbox_name

Override to specify the sandbox name if it is intended to be used.

soql

Override to return the raw string SOQL or the path to it.

is_soql_file

Override to True if soql property is a file path.

content_type

Override to use a different content type. Salesforce allows XML, CSV, ZIP_CSV, or ZIP_XML. Defaults to CSV.

run()

The task run method, to be overridden in a subclass.
See Task.run

merge_batch_results(result_ids)
Merges the resulting files of a multi-result batch bulk query.

class luigi.contrib.salesforce.SalesforceAPI(username, password, security_token,
sb_token=None, sandbox_name=None)

Bases: object

Class used to interact with the SalesforceAPI. Currently provides only the methods necessary for performing a bulk upload operation.

API_VERSION = 34.0
SOAP_NS = '{urn:partner.soap.sforce.com}'

start_session()
Starts a Salesforce session and determines which SF instance to use for future requests.

has_active_session()

query(query, **kwargs)
Return the result of a Salesforce SOQL query as a dict decoded from the Salesforce response JSON payload.

Parameters
query – the SOQL query to send to Salesforce, e.g. “SELECT id from Lead WHERE email = ‘a@b.com’”

query_more(next_records_identifier, identifier_is_url=False, **kwargs)
Retrieves more results from a query that returned more results than the batch maximum. Returns a dict decoded from the Salesforce response JSON payload.

Parameters
• next_records_identifier – either the Id of the next Salesforce object in the result, or a URL to the next record in the result.

• identifier_is_url – True if next_records_identifier should be treated as a URL, False if next_records_identifier should be treated as an Id.

query_all(query, **kwargs)
Returns the full set of results for the query. This is a convenience wrapper around query(...) and query_more(...). The returned dict is the decoded JSON payload from the final call to Salesforce, but with the totalSize field representing the full number of results retrieved and the records list representing the full list of records retrieved.

Parameters
query – the SOQL query to send to Salesforce, e.g. SELECT Id FROM Lead WHERE Email = “waldo@somewhere.com”

restful(path, params)
Allows you to make a direct REST call if you know the path Arguments: :param path: The path of the request. Example: sobjects/User/ABC123/password :param params: dict of parameters to pass to the path

create_operation_job(operation, obj, external_id_field_name=None, content_type=None)
Creates a new SF job that for doing any operation (insert, upsert, update, delete, query)

Parameters
• operation – delete, insert, query, upsert, update, hardDelete. Must be lowercase.

• obj – Parent SF object

• external_id_field_name – Optional.
**get_job_details** *(job_id)*

Gets all details for existing job

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>job_id as returned by ‘create_operation_job(...)’</td>
</tr>
</tbody>
</table>

**Returns**

job info as xml

**abort_job** *(job_id)*

Abort an existing job. When a job is aborted, no more records are processed. Changes to data may already have been committed and aren’t rolled back.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>job_id as returned by ‘create_operation_job(...)’</td>
</tr>
</tbody>
</table>

**Returns**

abort response as xml

**close_job** *(job_id)*

Closes job

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>job_id as returned by ‘create_operation_job(...)’</td>
</tr>
</tbody>
</table>

**Returns**

close response as xml

**create_batch** *(job_id, data, file_type)*

Creates a batch with either a string of data or a file containing data.

If a file is provided, this will pull the contents of the file_target into memory when running. That shouldn’t be a problem for any files that meet the Salesforce single batch upload size limit (10MB) and is done to ensure compressed files can be uploaded properly.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>job_id as returned by ‘create_operation_job(...)’</td>
</tr>
<tr>
<td>data</td>
<td></td>
</tr>
</tbody>
</table>

**Returns**

Returns batch_id

**block_on_batch** *(job_id, batch_id, sleep_time_seconds=5, max_wait_time_seconds=-1)*

Blocks until @batch_id is completed or failed. 

**get_batch_results** *(job_id, batch_id)*

DEPRECATED: Use get_batch_result_ids

**get_batch_result_ids** *(job_id, batch_id)*

Get result IDs of a batch that has completed processing.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_id</td>
<td>job_id as returned by ‘create_operation_job(...)’</td>
</tr>
<tr>
<td>batch_id</td>
<td>batch_id as returned by ‘create_batch(...)’</td>
</tr>
</tbody>
</table>

**Returns**

list of batch result IDs to be used in ‘get_batch_result(...)’

**get_batch_result** *(job_id, batch_id, result_id)*

Gets result back from Salesforce as whatever type was originally sent in create_batch (xml, or csv).

**luigi.contrib.scalding module**

**logger**

Scalding support for Luigi.
Example configuration section in luigi.cfg:

```
[scalding]
# scala home directory, which should include a lib subdir with scala jars.
scale-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 required 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
```

```python
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, tracking_url_callback=None)
```

```python
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
    ```

    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()
    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 subclass 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

job_args()
    Extra arguments to pass to the Scalding job.

args()
    Returns an array of args to pass to the job.

**luigi.contrib.sge module**

SGE batch system Tasks.

Adapted by Jake Feala (@jfeala) from LSF extension by Alex Wiltschko (@alexbw) Maintained by Jake Feala
(@jfeala)

SunGrid Engine is a job scheduler used to allocate compute resources on a shared cluster. Jobs are submitted using
the qsub command and monitored using qstat. To get started, install luigi on all nodes.

To run luigi workflows on an SGE cluster, subclass *luigi.contrib.sge.SGEJobTask* as you would any
luigi.Task, but override the **work()** method, instead of **run()**, to define the job code. Then, run your Luigi
workflow from the master node, assigning > 1 **workers** in order to distribute the tasks in parallel across the cluster.

The following is an example usage (and can also be found in sge_tests.py)

```python
import logging
import luigi
import os
from luigi.contrib.sge import SGEJobTask

logger = logging.getLogger('luigi-interface')
```

(continues on next page)
class TestJobTask(SGEJobTask):
    i = luigi.Parameter()

    def work(self):
        logger.info('Running test job...')
        with open(self.output().path, 'w') as f:
            f.write('this is a test')

    def output(self):
        return luigi.LocalTarget(os.path.join('/home', 'testfile_' + str(self.i)))

if __name__ == '__main__':
    tasks = [TestJobTask(i=str(i), n_cpu=i+1) for i in range(3)]
    luigi.build(tasks, local_scheduler=True, workers=3)

The n-cpu parameter allows you to define different compute resource requirements (or slots, in SGE terms) for each task. In this example, the third Task asks for 3 CPU slots. If your cluster only contains nodes with 2 CPUs, this task will hang indefinitely in the queue. See the docs for luigi.contrib.sge.SGEJobTask for other SGE parameters. As for any task, you can also set these in your luigi configuration file as shown below. The default values below were matched to the values used by MIT StarCluster, an open-source SGE cluster manager for use with Amazon EC2:

```
[SGEJobTask]
shared-tmp-dir = /home
parallel-env = orte
n-cpu = 2
```

class luigi.contrib.sge.SGEJobTask(*args, **kwargs)
    Bases: luigi.task.Task

    Base class for executing a job on SunGrid Engine

    Override work() (rather than run()) with your job code.

    Parameters:

    - n_cpu: Number of CPUs (or “slots”) to allocate for the Task. This value is passed as qsub -pe {pe} {n_cpu}

    - parallel_env: SGE parallel environment name. The default is “orte”, the parallel environment installed with MIT StarCluster. If you are using a different cluster environment, check with your sysadmin for the right pe to use. This value is passed as {pe} to the qsub command above.

    - shared_tmp_dir: Shared drive accessible from all nodes in the cluster. Task classes and dependencies are pickled to a temporary folder on this drive. The default is /home, the NFS share location setup by StarCluster.

    - job_name_format: String that can be passed in to customize the job name string passed to qsub; e.g. “Task123_{task_family}_{n_cpu}...”.

    - job_name: Exact job name to pass to qsub.

    - run_locally: Run locally instead of on the cluster.

    - poll_time: the length of time to wait in order to poll qstat

    - dont_remove_tmp_dir: Instead of deleting the temporary directory, keep it.
Luigi Documentation, Release 2.8.13

- **no_tarball**: Don’t create a tarball of the luigi project directory. Can be useful to reduce I/O requirements when the luigi directory is accessible from cluster nodes already.

  ```
  n_cpu = Insignificant IntParameter (defaults to 2)
  shared_tmp_dir = Insignificant Parameter (defaults to /home)
  parallel_env = Insignificant Parameter (defaults to orte)
  job_name_format = Insignificant Parameter (defaults to None): A string that can be formatted with class variables to name the job with qsub.
  run_locally = Insignificant BoolParameter (defaults to False): run locally instead of on the cluster
  poll_time = Insignificant IntParameter (defaults to 5): specify the wait time to poll qstat for the job status
  dont_remove_tmp_dir = Insignificant BoolParameter (defaults to False): don't delete the temporary directory used (for debugging)
  no_tarball = Insignificant BoolParameter (defaults to False): don't tarball (and extract) the luigi project files
  job_name = Insignificant Parameter (defaults to None): Explicit job name given via qsub.
  ```

- The task run method, to be overridden in a subclass.
  - See `Task.run`

- **work()**
  - Override this method, rather than `run()`, for your actual work.

```python
class Luigi.contrib.sge.LocalSGEJobTask(*args, **kwargs):
    Bases: Luigi.contrib.sge.SGEJobTask

    A local version of SGEJobTask, for easier debugging.
    This version skips the `qsub` steps and simply runs `work()` on the local node, so you don’t need to be on an SGE cluster to use your Task in a test workflow.

    run()
    - The task run method, to be overridden in a subclass.
    - See `Task.run`
```

### luigi.contrib.sge_runner module

The SunGrid Engine runner

The main() function of this module will be executed on the compute node by the submitted job. It accepts as a single argument the shared temp folder containing the package archive and pickled task to run, and carries out these steps:

- extract tarfile of package dependencies and place on the path
- unpickle SGETask instance created on the master node
- run SGETask.work()

On completion, SGETask on the master node will detect that the job has left the queue, delete the temporary folder, and return from SGETask.run()

```python
luigi.contrib.sge_runner.main(args=['/home/docs/checkouts/readthedocs.org/user_builds/luigi/envs/latest/bin/sphinx-build', '-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '': '_build/latex'])
```

Run the work() method from the class instance in the file “job-instance.pickle”.

9.1. Luigi package 147
luigi.contrib.simulate module

A module containing classes used to simulate certain behaviors

```python
class luigi.contrib.simulate.RunAnywayTarget(task_obj):
    Bases: luigi.target.Target

    A target used to make a task run every time it is called.

    Usage:
    Pass `self` as the first argument in your task’s `output`:
    And then mark it as `done` in your task’s `run`:
    ```
    temp_dir = '/tmp/luigi-simulate'
    temp_time = 86400
    unique = <Synchronized wrapper for c_int(0)>
    get_path()
    Returns a temporary file path based on a MD5 hash generated with the task’s name and its arguments
    exists()
    Checks if the file exists
    done()
    Creates temporary file to mark the task as `done`
```
```
luigi.contrib.spark module

```python
class luigi.contrib.spark.SparkSubmitTask(*args, **kwargs):
    Bases: luigi.contrib.external_program.ExternalProgramTask

    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
    ```
    name = None
    entry_class = None
    app = None
    always_log_stderr = False
    stream_for_searching_tracking_url = 'stderr'
    tracking_url_pattern
    app_options()
    Returns a subclass this method to map your task parameters to the app’s arguments
    ```
    pyspark_python
    pyspark_driver_python
    hadoop_user_name
    spark_version
    spark_submit
```
master
deploy_mode
jars
packages
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()

program_environment()  
Override this method to control environment variables for the program

Returns  dict mapping environment variable names to values

program_args()  
Override this method to map your task parameters to the program arguments

Returns  list to pass as args to subprocess.Popen

spark_command()

app_command()

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)

app = '/home/docs/checkouts/readthedocs.org/user_builds/luigi/envs/latest/lib/python2.7/luigi/contrib/pyspark_runner.py'
name
py_packages
files
setup \(\text{conf}\)
  Called by the pyspark_runner with a SparkConf instance that will be used to instantiate the SparkContext
  
  Parameters
  \text{conf} – SparkConf
setup_remote \(\text{sc}\)
main \(\text{sc}, \star\text{args}\)
  Called by the pyspark_runner with a SparkContext and any arguments returned by \text{app\_options}()
  
  Parameters
  \text{• sc} – SparkContext
  \text{• args} – arguments list
app\_command()
run()
  The task run method, to be overridden in a subclass.
  
  See \text{Task\_run}

\text{luigi.contrib.sparkey module}

class \text{luigi.contrib.sparkey.SparkeyExportTask} \(\star\text{args}, \star\star\text{kwargs}\)
  Bases: \text{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 \text{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.
  
  \quad \text{separator} = \text{'}\backslash t\text{'}
  
  \quad \text{run}()
  
  The task run method, to be overridden in a subclass.
  
  See \text{Task.run}

\text{luigi.contrib.sqla module}

Support for SQLAlchemy. 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):
```

(continues on next page)
# 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:

```python
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. Alternatively, if the data comes from another task, the modified example would look as shown below:

```python
from sqlalchemy import String
import luigi
from luigi.contrib import sqla
from luigi.mock import MockTarget

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

(continues on next page)
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

    Create new instance of Connection(engine, pid)

    engine
        Alias for field number 0

    pid
        Alias for field number 1

    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.

touch()
    Mark this update as complete.

exists()
    Returns True if the Target exists and False otherwise.

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.contrib.sq1a.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.
        • optionally override the schema attribute to use a different schema for the target table.

echo = False
connect_args = {}
connection_string
table
columns = []
schema = ''
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.

9.1. luigi package
**update_id()**

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

**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

**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()**

The task run method, to be overridden in a subclass.

See Task.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.

---

**luigi.contrib.ssh module**

Light-weight remote execution library and utilities.

There are some examples in the unittest but I added another that is 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).

**exception luigi.contrib.ssh.RemoteCalledProcessError**(returncode, command, host, output=None)

Bases: subprocess.CalledProcessError

**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** *(**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)*  
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.

**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)*  
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.

**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')
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. Using b is not supported; initialize with format=Nop instead.

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', 'listdir', 'getmerge', 'isdir', 'rename_dont_move', 'touchz']

luigi.contrib.webhdfs module

Provides a WebHdfsTarget using the Python hdfs
This module is DEPRECATED and does not play well with rest of luigi’s hdfs contrib module. You can consider migrating to luigi.contrib.hdfs.webhdfs_client.WebHdfsClient

class luigi.contrib.webhdfs.WebHdfsTarget (path, client=None, format=None)
Bases: luigi.target.FileSystemTarget

fs = None
open (mode='r')
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. Using b is not supported; initialize with format=Nop instead.

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() 

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-family

       family = OptionalParameter (defaults to 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.get_task_output_description**(task_output)

   Returns a task’s output as a string

**luigi.tools.deps.main()**

**luigi.tools.deps_tree module**

This module parses commands exactly the same as the luigi task runner. You must specify the module, the task and task paramters. Instead of executing a task, this module prints the significant paramters and state of the task and its dependencies in a tree format. Use this to visualize the execution plan in the terminal.

```
$ luigi-deps-tree --module foo_complex examples.Foo
...
|--[Foo-{} (PENDING)]
   |--[Bar-{'num': '0'} (PENDING)]
   |   |--[Bar-{'num': '4'} (PENDING)]
   |       |--[Bar-{'num': '5'} (PENDING)]
   |   |--[Bar-{'num': '1'} (PENDING)]
   |       |--[Bar-{'num': '6'} (PENDING)]
   |   |   [--[Bar-{'num': '7'} (PENDING)]
   |   |       |--[Bar-{'num': '9'} (PENDING)]
   |   |       |--[Bar-{'num': '10'} (PENDING)]
   |   |       |--[Bar-{'num': '11'} (PENDING)]
   |   |   (continues on next page)
```
class luigi.tools.deps_tree.bcolors
    colored output for task status
    OKBLUE = '\x1b[94m'
    OKGREEN = '\x1b[92m'
    ENDC = '\x1b[0m'
luigi.tools.deps_tree.print_tree(task, indent='', last=True)
    Return a string representation of the tasks, their statuses/parameters in a dependency tree format
luigi.tools.deps_tree.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.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 classes, e.g.
    DateParameter, DateHourParameter, ...., and different parameter naming, e.g. days_back/forward,
    hours_back/forward, ...., as well as different documentation wording, to improve user experience.)

    Subclasses will need to use the of parameter when overriding methods.

    of = TaskParameter:  task name to be completed. The task must take a single datetime parameter
    of_params = DictParameter (defaults to {}):  Arguments to be provided to the 'of' class
    start = Parameter
    stop = Parameter
    reverse = BoolParameter (defaults to False):  specifies the preferred order for catching up
    task_limit = IntParameter (defaults to 50):  how many of 'of' tasks to require. Guards
    now = IntParameter (defaults to None):  set to override current time. In seconds since
    param_name = Parameter (defaults to None):  parameter name used to pass in parameterized

    of_cls
        DONT USE. Will be deleted soon. Use self.of!

    datetime_to_parameter(dt)

    parameter_to_datetime(p)

    datetime_to_parameters(dt)

    parameters_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()

    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 subclass 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

missing_datetimes (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.

class luigi.tools.range.RangeDailyBase(*args, **kwargs)
Bases: luigi.tools.range.RangeBase

Produces a contiguous completed range of a daily recurring task.

start = DateParameter (defaults to None): beginning date, inclusive. Default: None – work backward forever (requires reverse=True)
stop = DateParameter (defaults to None): ending date, exclusive. Default: None – work forward forever
days_back = IntParameter (defaults to 100): extent to which contiguousness is to be assured into past, in days from now to prevent the oldest outputs flapping. Increase freely if you intend to process old dates - worker’s memory is the limit
days_forward = IntParameter (defaults to 0): extent to which contiguousness is to be assured into future, in days from current time. Prevents infinite loop when stop is None
datetime_to_parameter (dt)
parameter_to_datetime (p)
datetime_to_parameters (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)
Simply returns the points in time that correspond to turn of day.

class luigi.tools.range.RangeHourlyBase(*args, **kwargs)
Bases: luigi.tools.range.RangeBase

Produces a contiguous completed range of an hourly recurring task.

start = DateHourParameter (defaults to None): beginning datehour, inclusive. Default: None – work backward forever (requires reverse=True)
stop = DateHourParameter (defaults to None): ending datehour, exclusive. Default: None – work forward forever
hours_back = IntParameter (defaults to 2400): extent to which contiguousness is to be assured into past, in hours from now to prevent the oldest outputs flapping. Increase freely if you intend to process old dates - worker’s memory is the limit
hours_forward = IntParameter (defaults to 0): extent to which contiguousness is to be assured into future, in hours from current time. Prevents infinite loop when stop is None
datetime_to_parameter (dt)
parameter_to_datetime (p)
**datetime_to_parameters** (*dt*)

Given a date-time, will produce a dictionary of of-params combined with the ranged task parameter.

**parameters_to_datetime** (*p*)

Given a dictionary of parameters, will extract the ranged task parameter value.

**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*)

Simply returns the points in time that correspond to whole hours.

```python
class Luigi.tools.range.RangeByMinutesBase(*args, **kwargs)
Bases: Luigi.tools.range.RangeBase

Produces a contiguous completed range of an recurring tasks separated a specified number of minutes.

- **start** = DateMinuteParameter (defaults to None): beginning date-hour-minute, inclusive.
- **stop** = DateMinuteParameter (defaults to None): ending date-hour-minute, exclusive.
- **minutes_back** = IntParameter (defaults to 1440): extent to which contiguousness is to be ensured into the past, in minutes. Increase freely if you intend to process old dates - worker's memory is the limit.
- **minutes_forward** = IntParameter (defaults to 0): extent to which contiguousness is to be ensured into the future, in minutes from current time. Prevents infinite loop when stop is None.
- **minutes_interval** = IntParameter (defaults to 1): separation between events in minutes.

**datetime_to_parameters** (*dt*)

**parameter_to_datetime** (*p*)

**datetime_to_parameters** (*dt*)

Given a date-time, will produce a dictionary of of-params combined with the ranged task parameter.

**parameters_to_datetime** (*p*)

Given a dictionary of parameters, will extract the ranged task parameter value.

**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*)

Simply returns the points in time that correspond to whole hours.

**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.)

```python
class luigi.tools.range.RangeMonthly(*args, **kwargs)
    Bases: luigi.tools.range.RangeBase

    Produces a contiguous completed range of a monthly recurring task.

    Unlike the Range* classes with shorter intervals, this class does not perform bulk optimisation. It is assumed that the number of months is low enough not to motivate the increased complexity. Hence, there is no class RangeMonthlyBase.

    start = MonthParameter (defaults to None): beginning month, inclusive. Default: None - work backward forever (requires reverse=True)

    stop = MonthParameter (defaults to None): ending month, exclusive. Default: None - work forward forever

    months_back = IntParameter (defaults to 13): extent to which contiguousness is to be assured into past, in months from current time, to prevent the oldest outputs flapping. Increase freely if you intend to process old dates - worker's memory is the limit

    months_forward = IntParameter (defaults to 0): extent to which contiguousness is to be assured into future, in months from current time. Prevents infinite loop when stop is none

    datetime_to_parameter(dt)
    parameter_to_datetime(p)
    datetime_to_parameters(dt)
    Given a date-time, will produce a dictionary of of-params combined with the ranged task parameter

    parameters_to_datetime(p)
    Given a dictionary of parameters, will extract the ranged task parameter value

    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)
    Simply returns the points in time that correspond to turn of month.

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
    ```

    missing_datetimes(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.

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
```

`missing_datetimes` *(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.

```python
class luigi.tools.range.RangeByMinutes(*args, **kwargs)
```

Bases: `luigi.tools.range.RangeByMinutesBase`

Efficiently produces a contiguous completed range of an recurring task every interval minutes that takes a single `DateMinuteParameter`.

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 RangeByMinutes --of YourActualTask --start 2014-01-01T0123
```

`missing_datetimes` *(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.

### 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.

#### 9.1.2 Submodules

**luigi.batch_notifier module**

Library for sending batch notifications from the Luigi scheduler. This module is internal to Luigi and not designed for use in other contexts.

```python
class luigi.batch_notifier.batch_email(*args, **kwargs)
```

Bases: `luigi.task.Config`

```
email_interval = IntParameter (defaults to 60): Number of minutes between e-mail sends
```
batch_mode = ChoiceParameter (defaults to unbatched_params): Method used for batching failures in email. If "family" all tasks with the same name will be batched. If "all", tasks will only be batched if they have identical names. Choices: {all, unbatched_params, family}

error_lines = IntParameter (defaults to 20): Number of lines to show from each error message.

error_messages = IntParameter (defaults to 1): Number of error messages to show for each group.

group_by_error_messages = BoolParameter (defaults to True): Group items with the same error messages together.

class luigi.batch_notifier.ExplQueue(num_items)
    Bases: collections.OrderedDict
    enqueue(item)

class luigi.batch_notifier.BatchNotifier(**kwargs)
    Bases: object
    add_failure(task_name, family, unbatched_args, expl, owners)
    add_disable(task_name, family, unbatched_args, owners)
    add_scheduling_fail(task_name, family, unbatched_args, expl, owners)
    send_email()
    update()

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.cmdline_parser module

This module contains luigi internal parsing logic. Things exposed here should be considered internal to luigi.

class luigi.cmdline_parser.CmdlineParser(cmdline_args)
    Bases: object
    Helper for parsing command line arguments and used as part of the context when instantiating task objects.
    Initialize cmd line args
    classmethod get_instance()
        Singleton getter
    classmethod global_instance(**kwds)
        Meant to be used as a context manager.
    get_task_obj()
        Get the task object

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:

```python
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.

```python
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:

```python
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)`.

```python
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)**
    - Abstract class method.
For instance, `Month.from_date(datetime.date(2012, 6, 6))` returns a `Month(2012, 6)`.

```python
classmethod parse(s)
    Abstract class method.
    For instance, `Year.parse("2014")` returns a `Year(2014)`.
```

```python
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 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)
        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)`.
```

```python
class luigi.date_interval.Month(y, m)
    Bases: `luigi.date_interval.DateInterval`
    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)`.
```

```python
class luigi.date_interval.Year(y)
    Bases: `luigi.date_interval.DateInterval`
    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)`.
```

```python
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.

```python
to_string()
classmethod parse(s)
    Abstract class method.
    For instance, Year.parse("2014") returns a Year(2014).
```

**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.

```python
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.
    CURRENT_SOURCE_VERSION = 1
task_scheduled(task)
task_finished(task, successful)
task_started(task, 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.
```

```python
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
```
```python
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
```

```python
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
    task_id
    name
    host
    parameters
    events
```

**luigi.event module**

Definitions needed for events. See *Events and callbacks* for info on how to use it.

```python
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'
    PROGRESS = 'event.core.progress'
    This event can be fired by the task itself while running. The purpose is for the task to report progress,
    metadata or any generic info so that event handler listening for this can keep track of the progress of
    running task.
```
FAILURE = 'event.core.failure'
SUCCESS = 'event.core.success'
PROCESSING_TIME = 'event.core.processing_time'
TIMEOUT = 'event.core.timeout'
PROCESS_FAILURE = 'event.core.process_failure'

luigi.execution_summary module

This module provides the function summary() that is used for printing an execution summary at the end of luigi invocations.

class luigi.execution_summary.execution_summary(*args, **kwargs)
    Bases: luigi.task.Config
    summary_length = IntParameter (defaults to 5)

class luigi.execution_summary.LuigiStatusCode
    Bases: enum.Enum

All possible status codes for the attribute status in LuigiRunResult when the argument detailed_summary=True in luigi.run() / luigi.build. Here are the codes and what they mean:

<table>
<thead>
<tr>
<th>Status Code Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>There were no failed tasks or missing dependencies</td>
</tr>
<tr>
<td>SUCCESS_WITH_RETRY</td>
<td>There were failed tasks but they all succeeded in a retry</td>
</tr>
<tr>
<td>FAILED</td>
<td>There were failed tasks</td>
</tr>
<tr>
<td>FAILED_AND_SCHEDULING_FAILED</td>
<td>There were failed tasks and tasks whose scheduling failed</td>
</tr>
<tr>
<td>SCHEDULING_FAILED</td>
<td>There were tasks whose scheduling failed</td>
</tr>
<tr>
<td>NOT_RUN</td>
<td>There were tasks that were not granted run permission by the scheduler</td>
</tr>
<tr>
<td>MISSING_EXT</td>
<td>There were missing external dependencies</td>
</tr>
</tbody>
</table>

SUCCESS = (':)', 'there were no failed tasks or missing dependencies')
SUCCESS_WITH_RETRY = (':)', 'there were failed tasks but they all succeeded in a retry')
FAILED = (':(', 'there were failed tasks')
FAILED_AND_SCHEDULING_FAILED = (':(', 'there were failed tasks and tasks whose scheduling failed')
SCHEDULING_FAILED = (':(', 'there were tasks whose scheduling failed')
NOT_RUN = (':|', 'there were tasks that were not granted run permission by the scheduler')
MISSING_EXT = (':|', 'there were missing external dependencies')

class luigi.execution_summary.LuigiRunResult(worker, worker_add_run_status=True)
    Bases: object

The result of a call to build/run when passing the detailed_summary=True argument.

Attributes:
- one_line_summary (str): One line summary of the progress.
- summary_text (str): Detailed summary of the progress.
- status (LuigiStatusCode): Luigi Status Code. See LuigiStatusCode for what these codes mean.
• worker (luigi.worker.worker): Worker object. See worker.

• scheduling_succeeded (bool): Boolean which is True if all the tasks were scheduled without errors.

```python
luigi.execution_summary.summary(worker)
```
Given a worker, return a human readable summary of what the worker have done.

### luigi.format module

```python
class luigi.format.FileWrapper(file_object)
```
Wrap file in a “real” so stuff can be added to it after creation.

```python
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.

```python
create_subprocess(command)
```

```python
close()
readable()
writeable()
seekable()
```

```python
class luigi.format.OutputPipeProcessWrapper(command, output_pipe=None)
```
Bases: object

**WRITES_BEFORE_FLUSH = 10000**

```python
write(*args, **kwargs)
writeLine(line)
```

```python
close()
abort()
readable()
writeable()
seekable()
```

```python
class luigi.format.BaseWrapper(stream, *args, **kwargs)
```
Bases: object

```python
class luigi.format.NewlineWrapper(stream, newline=None)
```
Bases: luigi.format.BaseWrapper

```python
read(n=-1)
writelines(lines)
write(b)
```

```python
class luigi.format.MixedUnicodeBytesWrapper(stream, encoding=None)
```
Bases: luigi.format.BaseWrapper
write\((b)\)
write\(lines\)

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\(*args, \**kwargs\)
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
```python
input = 'bytes'
output = 'bytes'
pipe_reader (input_pipe)
pipe_writer (output_pipe)
```

```python
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.freezing module

Internal-only module with immutable data structures.

Please, do not use it outside of Luigi codebase itself.

```python
class luigi.freezing.FrozenOrderedDict (*args, **kwargs)
    Bases: _abcoll.Mapping
    It is an immutable wrapper around ordered dictionaries that implements the complete collections. Mapping interface. It can be used as a drop-in replacement for dictionaries where immutability and ordering are desired.
    get_wrapped()
```

`luigi.freezing.recursively_freeze`(`value`)

Recursively walks Mapping`s and `list`s and converts them to `FrozenOrderedDict and `tuples, respectively.

### luigi.interface module

This module contains the bindings for command line integration and dynamic loading of tasks

If you don’t want to run luigi from the command line. You may use the methods defined in this module to programatically run luigi.

```python
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.
```

`use_cmdline_section = False`

`local_scheduler = BoolParameter (defaults to False): Use an in-memory central scheduler`

`scheduler_host = Parameter (defaults to localhost): Hostname of machine running remote scheduler`

`scheduler_port = IntParameter (defaults to 8082): Port of remote scheduler api process`

`scheduler_url = Parameter (defaults to ): Full path to remote scheduler`
lock_size = IntParameter (defaults to 1): Maximum number of workers running the same command
no_lock = BoolParameter (defaults to False): Ignore if similar process is already running
lock_pid_dir = Parameter (defaults to /tmp/luigi): Directory to store the pid file
take_lock = BoolParameter (defaults to False): Signal other processes to stop getting work
workers = IntParameter (defaults to 1): Maximum number of parallel tasks to run
logging_conf_file = Parameter (defaults to ) : Configuration file for logging
log_level = ChoiceParameter (defaults to DEBUG): Default log level to use when logging_conf_file is not set
module = Parameter (defaults to ) : Used for dynamic loading of modules
parallel_scheduling = BoolParameter (defaults to False): Use multiprocessing to do scheduling in parallel
parallel_scheduling_processes = IntParameter (defaults to 0): The number of processes to use for scheduling in parallel
assistant = BoolParameter (defaults to False): Run any task from the scheduler.
help = BoolParameter (defaults to False): Show most common flags and all task-specific flags
help_all = BoolParameter (defaults to False): Show all command line flags

exception Luigi.interface.PidLockAlreadyTakenExit
Bases: exceptions.SystemExit

The exception thrown by luigi.run(), when the lock file is inaccessible

luigi.interface.run(*args, **kwargs)
Please dont use. Instead use luigi binary.
Run from cmdline using argparse.

Parameters

use_dynamic_argparse – Deprecated and ignored

luigi.interface.build(tasks, worker_scheduler_factory=None, detailed_summary=False, **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 True if there were no scheduling errors, even if tasks may fail.

luigi.local_target module

LocalTarget provides a concrete implementation of a Target class that uses files on the local file system

class Luigi.local_target.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

```python
move_to_final_destination()
generate_tmp_path(path)
```

class luigi.local_target.LocalFileSystem
Bases: luigi.target.FileSystem
Wrapper for access to file system operations.
Work in progress - add things as needed.
copy(old_path, new_path, raise_if_exists=False)
Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only single file copying but S3Client copies either a file or a directory as required.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
move(old_path, new_path, raise_if_exists=False)
Move file atomically. If source and destination are located on different filesystems, atomicity is approximated but cannot be guaranteed.
```
rename_dont_move(path, dest)
    Rename path to dest, but don’t move it into the dest folder (if it is a folder). This method is just a wrapper around the move method of LocalTarget.

class luigi.local_target.LocalTarget (path=None, format=None, is_tmp=False)
    Bases: luigi.target.FileSystemTarget

    fs = <luigi.local_target.LocalFileSystem object>
makedirs()
    Create all parent folders if they do not exist.

    open(mode='r')
    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. Using b is not supported; initialize with format=Nop instead.

    move(new_path, raise_if_exists=False)

    move_dir(new_path)

    remove()
    Remove the resource at the path specified by this FileSystemTarget.
    This method is implemented by using fs.

    copy(new_path, raise_if_exists=False)

    fn

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, kill_signal=None)
    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.metrics module

class luigi.metrics.MetricsCollectors
    Bases: enum.Enum

    default = 1
    none = 1
datadog = 2
prometheus = 3
get = <bound method EnumMeta.get of <enum 'MetricsCollectors'>>

class Luigi.metrics.MetricsCollector
Bases: object
Abstractable MetricsCollector base class that can be replace by tool specific implementation.
handle_task_started(task)
handle_task_failed(task)
handle_task_disabled(task, config)
handle_task_done(task)
generate_latest()
configure_http_handler(http_handler)

class Luigi.metrics.NoMetricsCollector
Bases: Luigi.metrics.MetricsCollector
Empty MetricsCollector when no collector is being used
handle_task_started(task)
handle_task_failed(task)
handle_task_disabled(task, config)
handle_task_done(task)

luigi.mock module

This module 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.
copy(path, dest, raise_if_exists=False)
   Copies the contents of a single file path to dest
get_all_data()
get_data(fn)
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)
   Removes the given mockfile. skip_trash doesn't have any meaning.
move(path, dest, raise_if_exists=False)
   Moves a single file from path to dest
.listdir(path)
   listdir does a prefix match of self.get_all_data(), but doesn't yet support globs.
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.

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()
        Returns True if the path for this FileSystemTarget exists; False otherwise.
        This method is implemented by using fs.

    move(path, raise_if_exists=False)
        Call MockFileSystem’s move command

    rename(*args, **kwargs)
        Call move to rename self

    open(mode='r')
        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. Using b is not supported; initialize with format=Nop instead.

**luigi.notifications module**

Supports sending emails when tasks fail. This needs some more documentation. See Configuration for configuration options. In particular using the config receiver should set up Luigi so that it will send emails when tasks fail.

```python
[email]
    receiver=foo@bar.baz
```

class luigi.notifications.TestNotificationsTask(*args, **kwargs)
    Bases: luigi.task.Task

    You may invoke this task to quickly check if you correctly have setup your notifications Configuration. You can run:

    ```
    $ luigi TestNotificationsTask --local-scheduler --email-force-send
    ```

    And then check your email inbox to see if you got an error email or any other kind of notifications that you expected.

    raise_in_complete = BoolParameter (defaults to False): If true, fail in complete() instead.
run()

The task run method, to be overridden in a subclass.

See Task.run

complete()

If the task has any outputs, return True if all outputs exist. Otherwise, return False.

However, you may freely override this method with custom logic.

class luigi.notifications.email(*args, **kwargs)
    Bases: luigi.task.Config

    force_send = BoolParameter (defaults to False): Send e-mail even from a tty
    format = ChoiceParameter (defaults to plain): Format type for sent e-mails Choices: {plain, html}
    method = ChoiceParameter (defaults to smtp): Method for sending e-mail Choices: {ses, sendgrid, smtp, sns}
    prefix = Parameter (defaults to ): Prefix for subject lines of all e-mails
    receiver = Parameter (defaults to ): Address to send error e-mails to
    sender = Parameter (defaults to luigi-client@build-11151103-project-12134-luigi): Address to send e-mails from

class luigi.notifications.smtp(*args, **kwargs)
    Bases: luigi.task.Config

    host = Parameter (defaults to localhost): Hostname of smtp server
    local_hostname = Parameter (defaults to None): If specified, local_hostname is used as the FQDN of the local host in the HELO/EHLO command
    no_tls = BoolParameter (defaults to False): Do not use TLS in SMTP connections
    password = Parameter (defaults to None): Password for the SMTP server login
    port = IntParameter (defaults to 0): Port number for smtp server
    ssl = BoolParameter (defaults to False): Use SSL for the SMTP connection.
    timeout = FloatParameter (defaults to 10.0): Number of seconds before timing out the smtp connection
    username = Parameter (defaults to None): Username used to log in to the SMTP host

class luigi.notifications.sendgrid(*args, **kwargs)
    Bases: luigi.task.Config

    apikey = Parameter: API key for SendGrid login

luigi.notifications.generate_email(sender, subject, message, recipients, image_png)

luigi.notifications.wrap_traceback(traceback)

For internal use only (until further notice)

luigi.notifications.send_email_smtp(sender, subject, message, recipients, image_png)

Sends notification through AWS SES.

    Does not handle access keys. Use either 1/ configuration file 2/ EC2 instance profile


luigi.notifications.send_email_sendgrid(sender, subject, message, recipients, image_png)

Sends notification through AWS SES.

    Does not handle access keys. Use either 1/ configuration file 2/ EC2 instance profile

luigi.notifications.send_email_ses(sender, subject, message, recipients, image_png)

Sends notification through AWS SES.

    Does not handle access keys. Use either 1/ configuration file 2/ EC2 instance profile

luigi.notifications.send_email_sns(sender, subject, message, topic_ARN, image_png)

Sends notification through AWS SNS. Takes Topic ARN from recipients.

    Does not handle access keys. Use either 1/ configuration file 2/ EC2 instance profile

```python
luigi.notifications.send_email(subject, message, sender, recipients, image_png=None)
```

Decides whether to send notification. Notification is cancelled if there are no recipients or if stdout is onto tty or if in debug mode.

 Dispatches on config value email.method. Default is ‘smtp’.

```python
luigi.notifications.send_error_email(subject, message, additional_recipients=None)
```

Sends an email to the configured error email, if it’s configured.

```python
luigi.notifications.format_task_error(headline, task, command, formatted_exception=None)
```

Format a message body for an error email related to a luigi.task.Task

**Parameters**

- **headline** – Summary line for the message
- **task** – luigi.task.Task instance where this error occurred
- **formatted_exception** – optional string showing traceback

**Returns** message body

**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.

```python
class luigi.parameter.ParameterVisibility
    Bases: enum.IntEnum

    Possible values for the parameter visibility option. Public is the default. See Parameters for more info.
    PUBLIC = 0
    HIDDEN = 1
    PRIVATE = 2

    has_value = <bound method EnumMeta.has_value of <enum 'ParameterVisibility'>>
    serialize()
```

**exception luigi.parameter.ParameterException**

**exception luigi.parameter.MissingParameterException**

**exception luigi.parameter.UnknownParameterException**

**exception luigi.parameter.DuplicateParameterException**

9.1. luigi package
class Luigi Documentation, Release 2.8.13

class luigi.parameter.Parameter (default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)

Bases: object

Parameter whose value is a str, and a base class for other parameter types.

Parameters are objects set on the Task class level to make it possible to parameterize tasks. For instance:

```python
class MyTask (luigi.Task):
    foo = luigi.Parameter ()

class RequiringTask (luigi.Task):
    def requires (self):
        return MyTask (foo="hello")

    def run (self):
        print (self.requires ().foo)  # prints "hello"
```

This makes it possible to instantiate multiple tasks, eg MyTask (foo='bar') and MyTask (foo='baz'). The task will then have the foo attribute set appropriately.

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. When the value is not provided, the value will be resolved in this order of falling priority:

- Any value provided on the command line:
  - To the root task (eg. --param xyz)
  - Then to the class, using the qualified task name syntax (eg. --TaskA-param xyz).
- With [TASK_NAME]>PARAM_NAME: <serialized value> syntax. See Parameters from config Ingestion
- Any default value set using the default flag.

Parameter objects may be reused, but you must then set the positional=False flag.

Parameters

- `default` – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.
- `significant` (bool) – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.
- `description` (str) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.
- `config_path` (dict) – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.
- `positional` (bool) – If true, you can set the argument as a positional argument. It’s true by default but we recommend positional=False for abstract base classes and similar cases.
• **always_in_help** *(bool)* – For the \-help option in the command line parsing. Set true to always show in \-help.

• **batch_method** *(function(iterable[A])\rightarrow A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a *ParameterVisibility*. Default value is *ParameterVisibility.PUBLIC*

**has_task_value** *(task_name, param_name)*

**task_value** *(task_name, param_name)*

**parse**(x)

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

**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.

**normalize**(x)

Given a parsed parameter value, normalizes it.

The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

**next_in_enumeration** *(value)*

If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the *luigi.execution_summary* module for pretty printing purposes. Enabling it to pretty print tasks like MyTask(num=1), MyTask(num=2), MyTask(num=3) to MyTask(num=1..3).

**Parameters**

value – The value

**Returns**

The next value, like “value + 1”. Or None if there’s no enumerable ordering.

**class** *luigi.parameter.OptionalParameter*(default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)

**Bases**: *luigi.parameter.Parameter*

A Parameter that treats empty string as None

**Parameters**

• **default** – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.
• **significant** *(bool)* – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

• **description** *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the *help* string shown to users. Default: None.

• **config_path** *(dict)* – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.

• **positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

• **always_in_help** *(bool)* – For the –help option in the command line parsing. Set true to always show in –help.

• **batch_method** *(function(iterable[A]–>A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`

`serialize(x)`

Opposite of `parse()`.

Converts the value `x` to a string.

Parameters `x` – the value to serialize.

`parse(x)`

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

Parameters `x` *(str)* – the value to parse.

Returns the parsed value.

class `luigi.parameter.DateParameter`(interval=1, start=None, **kwargs)

Bases: `luigi.parameter._DateParameterBase`

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. DateParameters are 90% of the time used to be interpolated into file system paths or the like. Here is a gentle reminder of how to interpolate date parameters into strings:

```python
class MyTask(luigi.Task):
    date = luigi.DateParameter()

    def run(self):
        templated_path = "/my/path/to/my/dataset/{date:%Y/%m/%d}/"
        instantiated_path = templated_path.format(date=self.date)
        # print(instantiated_path) --> /my/path/to/my/dataset/2016/06/09/
        # ... use instantiated_path ...
```

182 Chapter 9. API Reference
To set this parameter to default to the current day. You can write code like this:

```python
import datetime

class MyTask(luigi.Task):
    date = luigi.DateParameter(default=datetime.date.today())

date_format = '%Y-%m-%d'

next_in_enumeration(value)
    If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the `luigi.execution_summary` module for pretty printing purposes. Enabling it to pretty print tasks like `MyTask(num=1), MyTask(num=2), MyTask(num=3)` to `MyTask(num=1..3)`.

    Parameters
        value -- The value

    Returns
        The next value, like “value + 1”. Or None if there’s no enumerable ordering.

normalize(value)
    Given a parsed parameter value, normalizes it.

    The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

    This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

class luigi.parameter.MonthParameter(interval=1, start=None, **kwargs)
    Bases: luigi.parameter.DateParameter

    Parameter whose value is a date, specified to the month (day of date is “rounded” to first of the month).

    A MonthParameter is a Date string formatted YYYY-MM. For example, 2013-07 specifies July of 2013. Task objects constructed from code accept date (ignoring the day value) or Month.

date_format = '%Y-%m'

next_in_enumeration(value)
    If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the `luigi.execution_summary` module for pretty printing purposes. Enabling it to pretty print tasks like `MyTask(num=1), MyTask(num=2), MyTask(num=3)` to `MyTask(num=1..3)`.

    Parameters
        value -- The value

    Returns
        The next value, like “value + 1”. Or None if there’s no enumerable ordering.

normalize(value)
    Given a parsed parameter value, normalizes it.

    The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

    This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

class luigi.parameter.YearParameter(interval=1, start=None, **kwargs)
    Bases: luigi.parameter.DateParameter
Parameter whose value is a date, specified to the year (day and month of date is “rounded” to first day of the year).

A YearParameter is a Date string formatted YYYY. Task objects constructed from code accept date (ignoring the month and day values) or Year.

date_format = '%Y'

next_in_enumeration(value)
If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the luigi.execution_summary module for pretty printing purposes. Enabling it to pretty print tasks like MyTask(num=1), MyTask(num=2), MyTask(num=3) to MyTask(num=1..3).

Parameters
value – The value

Returns
The next value, like “value + 1”. Or None if there’s no enumerable ordering.

normalize(value)
Given a parsed parameter value, normalizes it.

The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as x // 2 * 2.

class luigi.parameter.DateHourParameter(interval=1, start=None, **kwargs)
Bases: luigi.parameter._DatetimeParameterBase

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'

class luigi.parameter.DateMinuteParameter(interval=1, start=None, **kwargs)
Bases: luigi.parameter._DatetimeParameterBase

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, 2013-07-10T1907 specifies July 10, 2013 at 19:07.

The interval parameter can be used to clamp this parameter to every N minutes, instead of every minute.

date_format = '%Y-%m-%dT%H%M'

deprecated_date_format = '%Y-%m-%dT%HH%M'

parse(s)
 Parses a string to a datetime.

class luigi.parameter.DateSecondParameter(interval=1, start=None, **kwargs)
Bases: luigi.parameter._DatetimeParameterBase

Parameter whose value is a datetime specified to the second.

A DateSecondParameter is a ISO 8601 formatted date and time specified to the second. For example, 2013-07-10T190738 specifies July 10, 2013 at 19:07:38.

The interval parameter can be used to clamp this parameter to every N seconds, instead of every second.

date_format = '%Y-%m-%dT%H%M%S'
class Luigi::Parameter::IntParameter

Parameter whose value is an int.

Parameters

- `default` – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime::date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- `significant` (bool) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- `description` (str) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: `None`.

- `config_path` (dict) – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- `positional` (bool) – If true, you can set the argument as a positional argument. It’s true by default but we recommend positional=False for abstract base classes and similar cases.

- `always_in_help` (bool) – For the -help option in the command line parsing. Set true to always show in -help.

- `batch_method` (function(iterable[A]->A)) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See Batch multiple parameter values into a single run

- `visibility` – A Parameter whose value is a ParameterVisibility. Default value is ParameterVisibility::PUBLIC

parse(s)

Parses an int from the string using `int()`.

next_in_enumeration(value)

If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the luigi::execution::summary module for pretty printing purposes. Enabling it to pretty print tasks like `MyTask(num=1), MyTask(num=2), MyTask(num=3)` to `MyTask(num=1..3)`.

Parameters value – The value

Returns The next value, like “value + 1”. Or None if there’s no enumerable ordering.

class Luigi::Parameter::FloatParameter

Bases: Luigi::Parameter::Parameter
Parameter whose value is a float.

Parameters

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- **significant** (`bool`) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- **description** (`str`) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.

- **config_path** (`dict`) – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- **positional** (`bool`) – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- **always_in_help** (`bool`) – For the `-help` option in the command line parsing. Set true to always show in `-help`.

- **batch_method** (`function(iterable[A]) -> A`) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See `Batching multiple parameter values into a single run`.

- **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

```
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 Parameter has an implicit default value of `False`. For the command line interface this means that the value is `False` unless you add `"--the-bool-parameter"` to your command without giving a parameter value. This is considered `implicit` parsing (the default). However, in some situations one might want to give the explicit bool value ("--the-bool-parameter true|false"), e.g. when you configure the default value to be `True`. This is called `explicit` parsing. When omitting the parameter value, it is still considered `True` but to avoid ambiguities during argument parsing, make sure to always place bool parameters behind the task family on the command line when using explicit parsing.

You can toggle between the two parsing modes on a per-parameter base via

```
class MyTask(luigi.Task):
```

or globally by

```
```
for all bool parameters instantiated after this line.

```python
IMPLICIT_PARSING = 'implicit'
EXPLICIT_PARSING = 'explicit'
parsing = 'implicit'

parse(val)
    Parses a bool from the string, matching ‘true’ or ‘false’ ignoring case.

normalize(value)
    Given a parsed parameter value, normalizes it.

    The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

    This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

class luigi.parameter.DateIntervalParameter (default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
```

Bases: luigi.parameter.Parameter

A Parameter whose value is a DateInterval.

Date Intervals are specified using the ISO 8601 date notation for dates (eg. “2015-11-04”), months (eg. “2015-05”), years (eg. “2015”), or weeks (eg. “2015-W35”). In addition, it also supports arbitrary date intervals provided as two dates separated with a dash (eg. “2015-11-04-2015-12-04”).

Parameters

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for DateParameter or `int` for IntParameter. By default, no default is stored and the value must be specified at runtime.

- **significant** (`bool`) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- **description** (`str`) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.

- **config_path** (`dict`) – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- **positional** (`bool`) – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- **always_in_help** (`bool`) – For the –help option in the command line parsing. Set true to always show in –help.
• **batch_method** (*function(iterable[A]) -> A*) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a *ParameterVisibility*. Default value is *ParameterVisibility.PUBLIC*

```python
parse(s)
```

Parses a *DateInterval* from the input.

*see* [luigi.date_interval](#) for details on the parsing of DateIntervals.

class luigi.parameter.TimeDeltaParameter

*default=<$object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>*

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](#)

**Parameters**

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- **significant** (*bool*) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- **description** (*str*) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the *help* string shown to users. Default: `None`.

- **config_path** (*dict*) – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- **positional** (*bool*) – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- **always_in_help** (*bool*) – For the --help option in the command line parsing. Set true to always show in --help.

- **batch_method** (*function(iterable[A]) -> A*) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

- **visibility** – A Parameter whose value is a *ParameterVisibility*. Default value is *ParameterVisibility.PUBLIC*
parse \textbf{(input)}

Parses a time delta from the input.

See \textit{TimeDeltaParameter} for details on supported formats.

\textbf{serialize \textbf{(x)}}

Converts \texttt{datetime.timedelta} to a string

\textbf{Parameters} \textbf{x} – the value to serialize.

\textbf{class} \textit{luigi.parameter.TaskParameter}(default=<object object>, \textit{is\_global}=False, \textit{significant}=True, \textit{description}=None, \textit{config\_path}=None, \textit{positional}=True, \textit{always\_in\_help}=False, \textit{batch\_method}=None, \textit{visibility}=\textless ParameterVisibility\text{.PUBLIC}>)

\textbf{Bases:} \textit{luigi.parameter.Parameter}

A parameter that takes another luigi task class.

When used programatically, the parameter should be specified directly with the \textit{luigi.task.Task} (sub) class. Like \texttt{MyMetaTask(my\_task\_param=my\_tasks.MyTask)}. On the command line, you specify the \texttt{luigi.task.Task.get\_task\_family()} method. Like

\texttt{$\$\text{ luigi --module my\_tasks MyMetaTask --my\_task\_param my\_namespace.MyTask}$}

Where \texttt{my\_namespace.MyTask} is defined in the \texttt{my\_tasks} python module.

When the \textit{luigi.task.Task} class is instantiated to an object. The value will always be a task class (and not a string).

\textbf{Parameters}

- \textbf{default} – the default value for this parameter. This should match the type of the Parameter, i.e. \texttt{datetime.date} for \texttt{DateParameter} or \texttt{int} for \texttt{IntParameter}. By default, no default is stored and the value must be specified at runtime.

- \textbf{significant} \textbf{(bool)} – specify \texttt{False} if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: \texttt{True}.

- \textbf{description} \textbf{(str)} – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the \textit{help} string shown to users. Default: \texttt{None}.

- \textbf{config\_path} \textbf{(dict)} – a dictionary with entries \texttt{section} and \texttt{name} specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: \texttt{None}.

- \textbf{positional} \textbf{(bool)} – If true, you can set the argument as a positional argument. It’s true by default but we recommend \texttt{positional=False} for abstract base classes and similar cases.

- \textbf{always\_in\_help} \textbf{(bool)} – For the \texttt{-help} option in the command line parsing. Set true to always show in \texttt{-help}.

- \textbf{batch\_method} \textbf{(function(iterable[A])\rightarrow A)} – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See \textit{Batching multiple parameter values into a single run}

- \textbf{visibility} – A Parameter whose value is a \texttt{ParameterVisibility}. Default value is \texttt{ParameterVisibility\text{.PUBLIC}}
**parse**\((input)\)

Parse a task family using the `Register`.

**serialize**\((cls)\)

Converts the `luigi.task.Task` (sub) class to its family name.

```python
class luigi.parameter.EnumParameter(*args, **kwargs)
    Bases: luigi.parameter.Parameter

A parameter whose value is an `Enum`.

In the task definition, use

```python
class Model(enum.Enum):
    Honda = 1
    Volvo = 2
class MyTask(luigi.Task):
    my_param = luigi.EnumParameter(enum=Model)
```

At the command line, use,

```bash
$ luigi --module my_tasks MyTask --my-param Honda
```

**parse**\((s)\)

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

Parameters

- **x** (`str`) – the value to parse.

Returns

the parsed value.

**serialize**\((e)\)

Converts the value `x` to a string.

Parameters

- **x** – the value to serialize.

```python
class luigi.parameter.EnumListParameter(*args, **kwargs)
    Bases: luigi.parameter.Parameter

A parameter whose value is a comma-separated list of `Enum`. Values should come from the same enum. Values are taken to be a list, i.e. order is preserved, duplicates may occur, and empty list is possible.

In the task definition, use

```python
class Model(enum.Enum):
    Honda = 1
    Volvo = 2
class MyTask(luigi.Task):
    my_param = luigi.EnumListParameter(enum=Model)
```

At the command line, use,

```bash
$ luigi --module my_tasks MyTask --my-param Honda,Volvo
```
The default implementation is the identity function, but subclasses should override this method for specialized parsing.

**Parameters**

\[ x \text{ (} \text{str}\text{)} \] – the value to parse.

**Returns**

the parsed value.

### serialize

\[ \text{enum\_values} \]

Opposite of \[ \text{parse()} \].

Converts the value \( x \) to a string.

**Parameters**

\[ x \] – the value to serialize.

---

**class** luigi.parameter.DictParameter

Parameter whose value is a dict.

In the task definition, use

```python
class MyTask(luigi.Task):
    tags = luigi.DictParameter()

    def run(self):
        logging.info("Find server with role: \$s", self.tags['role'])
        server = aws.ec2.find_my_resource(self.tags)
```

At the command line, use

```
$ luigi --module my_tasks MyTask --tags <JSON string>
```

Simple example with two tags:

```
$ luigi --module my_tasks MyTask --tags \"{""role"": "web", "env": "staging"}\"
```

It can be used to define dynamic parameters, when you do not know the exact list of your parameters (e.g. list of tags, that are dynamically constructed outside Luigi), or you have a complex parameter containing logically related values (like a database connection config).

**Parameters**

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.

- **significant** (bool) – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.

- **description** (str) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.

- **config_path** (dict) – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.
• **positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

• **always_in_help** *(bool)* – For the `--help` option in the command line parsing. Set true to always show in `--help`.

• **batch_method** *(function(iterable[A]→A)*) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`

`normalize(value)`

Ensure that dictionary parameter is converted to a FrozenOrderedDict so it can be hashed.

`parse(source)`

Parses an immutable and ordered `dict` from a JSON string using standard JSON library.

We need to use an immutable dictionary, to create a hashable parameter and also preserve the internal structure of parsing. The traversal order of standard `dict` is undefined, which can result various string representations of this parameter, and therefore a different task id for the task containing this parameter. This is because task id contains the hash of parameters’ JSON representation.

**Parameters**

- `s` – String to be parsed

`serialize(x)`

Opposite of `parse()`.

Converts the value `x` to a string.

**Parameters**

- `x` – the value to serialize.

---

**class** `luigi.parameter.ListParameter` *(default=<object object>, `is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)*)

Bases: `luigi.parameter.Parameter`

Parameter whose value is a list.

In the task definition, use

```python
class MyTask(luigi.Task):
    grades = luigi.ListParameter()

    def run(self):
        sum = 0
        for element in self.grades:
            sum += element
        avg = sum / len(self.grades)
```

At the command line, use

```bash
$ luigi --module my_tasks MyTask --grades <JSON string>
```

Simple example with two grades:

```bash
$ luigi --module my_tasks MyTask --grades '[100,70]'
```
Parameters

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- **significant** (*bool*) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- **description** (*str*) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.

- **config_path** (*dict*) – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- **positional** (*bool*) – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- **always_in_help** (*bool*) – For the `-help` option in the command line parsing. Set true to always show in `-help`.

- **batch_method** (*function(iterable[A])->A*) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*.

- **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

`normalize(x)`

Ensure that struct is recursively converted to a tuple so it can be hashed.

Parameters

- **x** (*str*) – the value to parse.

Returns the normalized (hashable/immutable) value.

`parse(x)`

Parse an individual value from the input.

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.

Class *luigi.parameter.TupleParameter*

Parameter whose value is a `tuple` or `tuple` of tuples.

In the task definition, use
```python
class MyTask(luigi.Task):
    book_locations = luigi.TupleParameter()

    def run(self):
        for location in self.book_locations:
            print("Go to page \$d, line \$d" % (location[0], location[1]))
```

At the command line, use:

```
$ luigi --module my_tasks MyTask --book_locations <JSON string>
```

Simple example with two grades:

```
$ luigi --module my_tasks MyTask --book_locations '((12,3),(4,15),(52,1))'
```

### Parameters

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.
- **significant** (`bool`) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.
- **description** (`str`) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.
- **config_path** (`dict`) – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.
- **positional** (`bool`) – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.
- **always_in_help** (`bool`) – For the –help option in the command line parsing. Set true to always show in –help.
- **batch_method** (`function(iterable[A]-->A)`) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See Batching multiple parameter values into a single run.
- **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

```python
parse(x)
```

Parse an individual value from the input.

**Parameters** `x` (str) – the value to parse.

**Returns** the parsed value.

```python
class luigi.parameter.NumericalParameter(left_op=<built-in function le>, right_op=<built-in function lt>, *args, **kwargs)
```

Bases: `luigi.parameter.Parameter`  

Parameter whose value is a number of the specified type, e.g. `int` or `float` and in the range specified.
In the task definition, use

```python
class MyTask(luigi.Task):
    my_param_1 = luigi.NumericalParameter(
        var_type=int, min_value=-3, max_value=7)  # -3 <= my_param_1 < 7
    my_param_2 = luigi.NumericalParameter(
        var_type=int, min_value=-3, max_value=7, left_op=operator.lt, right_op=operator.le)  # -3 < my_param_2 <= 7
```

At the command line, use

```
$ luigi --module my_tasks MyTask --my-param-1 -3 --my-param-2 -2
```

**Parameters**

- **var_type** *(function)* – The type of the input variable, e.g., int or float.
- **min_value** – The minimum value permissible in the accepted values range. May be inclusive or exclusive based on left_op parameter. This should be the same type as var_type.
- **max_value** – The maximum value permissible in the accepted values range. May be inclusive or exclusive based on right_op parameter. This should be the same type as var_type.
- **left_op** *(function)* – The comparison operator for the left-most comparison in the expression `min_value left_op value right_op value`. This operator should generally be either `operator.lt` or `operator.le`. Default: `operator.le`.
- **right_op** *(function)* – The comparison operator for the right-most comparison in the expression `min_value left_op value right_op value`. This operator should generally be either `operator.lt` or `operator.le`. Default: `operator.lt`.

**parse(s)**

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

Parameters `x (str)` – the value to parse.

Returns the parsed value.

```python
class luigi.parameter.ChoiceParameter(var_type=<type 'str'>, *args, **kwargs)
Bases: luigi.parameter.Parameter

A parameter which takes two values:

1. an instance of Iterable and
2. the class of the variables to convert to.
```

In the task definition, use

```python
class MyTask(luigi.Task):
    my_param = luigi.ChoiceParameter(Choices=[0.1, 0.2, 0.3], var_type=float)
```

At the command line, use

```
$ luigi --module my_tasks MyTask --my-param 0.1
```

Consider using *EnumParameter* for a typed, structured alternative. This class can perform the same role when all choices are the same type and transparency of parameter value on the command line is desired.

---

9.1. luigi package
Parameters

- **var_type** *(function)* – The type of the input variable, e.g. str, int, float, etc. Default: str
- **choices** – An iterable, all of whose elements are of var_type to restrict parameter choices to.

**parse** *(s)*

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

**Parameters**

- **x** *(str)* – the value to parse.

**Returns**

the parsed value.

**normalize** *(var)*

Given a parsed parameter value, normalizes it.

The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

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, unix_socket=None)*

luigi.retcodes module

Module containing the logic for exit codes for the luigi binary. It’s useful when you in a programmatic way need to know if luigi actually finished the given task, and if not why.

**class** luigi.retcodes.retcode *(args, **kwargs)*

**Bases:** luigi.task.Config

See the `return codes configuration section`.

- **unhandled_exception** = IntParameter *(defaults to 4)*: For internal luigi errors.
- **missing_data** = IntParameter *(defaults to 0)*: For when there are incomplete ExternalTask dependencies.
- **task_failed** = IntParameter *(defaults to 0)*: For when a task’s run() method fails.
- **already_running** = IntParameter *(defaults to 0)*: For both local --lock and luigid "lock files".
- **scheduling_error** = IntParameter *(defaults to 0)*: For when a task’s complete() or requires() fails, or task-limit reached.
- **not_run** = IntParameter *(defaults to 0)*: For when a task is not granted run permission.
luigi.retcodes.run_with_retcodes(argv)
Run luigi with command line parsing, but raise SystemExit with the configured exit code.

Note: Usually you use the luigi binary directly and don’t call this function yourself.

Parameters argv – Should (conceptually) be sys.argv[1:]

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
Bases: exceptions.Exception

class luigi.rpc.URLLibFetcher
Bases: object
raises = (('<class 'urllib2.URLError'>), '<class 'socket.timeout'>)
fetch(full_url, body, timeout)

class luigi.rpc.RequestsFetcher(session)
Bases: object
check_pid()
fetch(full_url, body, timeout)

class luigi.rpc.RemoteScheduler(url='http://localhost:8082/', connect_timeout=None)
Bases: object
Scheduler proxy object. Talks to a RemoteSchedulerResponder.

add_scheduler_message_response(*args, **kwargs)
add_task(*args, **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_task_batcher(*args, **kwargs)
add_worker(*args, **kwargs)
announce_scheduling_failure(*args, **kwargs)
count_pending(*args, **kwargs)
decrease_running_task_resources(*args, **kwargs)
dep_graph(*args, **kwargs)
disable_worker(*args, **kwargs)
fetch_error(*args, **kwargs)
forgive_failures(*args, **kwargs)
get_running_task_resources(*args, **kwargs)
get_scheduler_message_response (*args, **kwargs)
get_task_progress_percentage (*args, **kwargs)
get_task_status_message (*args, **kwargs)
get_work (*args, **kwargs)
graph (*args, **kwargs)
has_task_history (*args, **kwargs)
inverse_dep_graph (*args, **kwargs)
is_pause_enabled (*args, **kwargs)
is_paused (*args, **kwargs)
mark_as_done (*args, **kwargs)
pause (*args, **kwargs)
ping (*args, **kwargs)
prune (*args, **kwargs)
re_enable_task (*args, **kwargs)
resource_list (*args, **kwargs)

Resources usage info and their consumers (tasks).
send_scheduler_message (*args, **kwargs)
set_task_progress_percentage (*args, **kwargs)
set_task_status_message (*args, **kwargs)
set_worker_processes (*args, **kwargs)
task_list (*args, **kwargs)
Query for a subset of tasks by status.
task_search (*args, **kwargs)
Query for a subset of tasks by task_id.

Parameters task_str –

Returns
unpause (*args, **kwargs)
update_metrics_task_started (*args, **kwargs)
update_resource (*args, **kwargs)
update_resources (*args, **kwargs)
worker_list (*args, **kwargs)

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.

luigi.scheduler.UPSTREAM_SEVERITY_KEY() => integer – return first index of value. Raises ValueError if the value is not present.
class Luigi Documentation, Release 2.8.13

class `luigi.scheduler.RetryPolicy`(*retry_count, disable_hard_timeout, disable_window*)

Create new instance of RetryPolicy(*retry_count, disable_hard_timeout, disable_window*)

`disable_hard_timeout`
Alias for field number 1

`disable_window`
Alias for field number 2

`retry_count`
Alias for field number 0

class `luigi.scheduler.rpc_method(**request_args)`

class `luigi.scheduler.scheduler(*args, **kwargs)`

Bases: `luigi.task.Config`

retry_delay = FloatParameter (defaults to 900.0)
remove_delay = FloatParameter (defaults to 600.0)
worker_disconnect_delay = FloatParameter (defaults to 60.0)
state_path = Parameter (defaults to /var/lib/luigi-server/state.pickle)
batch_emails = BoolParameter (defaults to False): Send e-mails in batches rather than
disable_window = IntParameter (defaults to 3600)
retry_count = IntParameter (defaults to 999999999)
disable_hard_timeout = IntParameter (defaults to 999999999)
disable_persist = IntParameter (defaults to 86400)
max_shown_tasks = IntParameter (defaults to 100000)
max_graph_nodes = IntParameter (defaults to 100000)
record_task_history = BoolParameter (defaults to False)
prune_on_get_work = BoolParameter (defaults to False)
pause_enabled = BoolParameter (defaults to True)
send_messages = BoolParameter (defaults to True)
metrics_collector = EnumParameter (defaults to MetricsCollectors.default)

stable_done_cooldown_secs = IntParameter (defaults to 10): Sets cooldown period to avoid running the same task twice
Sets a cooldown period in seconds after a task was completed, during this period the same task will not
accepted by the scheduler.

class `luigi.scheduler.Failsures(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.

um_failures()
    Return the number of failures in the window.
clear()
    Clear the failure queue.

class Luigi.schedluigi.scheduler.OrderedSet (iterable=None)
    Bases: _abcoll.MutableSet

    Standard Python OrderedSet recipe found at http://code.activestate.com/recipes/576694/
    Modified to include a peek function to get the last element

    add(key)
        Add an element.

discard(key)
    Remove an element. Do not raise an exception if absent.

    peek (last=True)

    pop (last=True)
        Return the popped value. Raise KeyError if empty.

class Luigi.schedluigi.scheduler.Task (task_id, status, deps, resources=None, priority=0, family="",
    module=None, params=None, param_visibilities=None, accepts_messages=False, tracking_url=None, status_message=None, progress_percentage=None, retry_policy='notoptional')

    Bases: object

    set_params (params)

    is_batchable()

    add_failure()

    has_excessive_failures()

    pretty_id

class Luigi.schedluigi.scheduler.Worker (worker_id, last_active=None)
    Bases: object

    Structure for tracking worker activity and keeping their references.

    add_info (info)

    update (worker_reference, get_work=False)

    prune (config)

    get_tasks (state, *statuses)

    is_trivial_worker (state)
        If it's not an assistant having only tasks that are without requirements.
        We have to pass the state parameter for optimization reasons.

    assistant

    enabled

    state

    add_rpc_message (name, **kwargs)
```python
class Luigi.schedule.SimpleTaskState(state_path)

    Bases: object

    Keep track of the current state and handle persistence.
    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.

    get_state()
    set_state(state)
    dump()
    load()
    get_active_tasks()
    get_active_tasks_by_status(*statuses)
    get_active_task_count_for_status(status)
    get_batch_running_tasks(batch_id)
    set_batcher(worker_id, family, batcher_args, max_batch_size)
    get_batcher(worker_id, family)
    num_pending_tasks()
        Return how many tasks are PENDING + RUNNING. O(1).
    get_task(task_id, default=None, setdefault=None)
    has_task(task_id)
    re_enable(task, config=None)
    set_batch_running(task, batch_id, worker_id)
    set_status(task, new_status, config=None)
    fail_dead_worker_task(task, config, assistants)
    update_status(task, config)
    may_prune(task)
    inactivate_tasks(delete_tasks)
    get_active_workers(last_active_lt=None, last_get_work_gt=None)
    get_assistants(last_active_lt=None)
    get_worker_ids()
    get_worker(worker_id)
    inactivate_workers(delete_workers)
    disable_workers(worker_ids)
    update_metrics(task, config)
```

```python
class Luigi.scheduler.Scheduler(config=None, resources=None, task_history_impl=None, **kwargs)

    Bases: object

    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_impl: ignore config and use this object as the task history

- load()
- dump()
- prune()
- add_task_batcher(worker, task_family, batched_args, max_batch_size=inf)
- forgive_failures(task_id=None)
- mark_as_done(task_id=None)
- 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, param_visibilities=None, accepts_messages=False, assistant=False, tracking_url=None, worker=None, batchable=None, batch_id=None, retry_policy_dict=None, owners=None, **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

- announce_scheduling_failure(task_name, family, params, expl, owners, **kwargs)
- add_worker(worker, info, **kwargs)
- disable_worker(worker)
- set_worker_processes(worker, n)
- send_scheduler_message(worker, task, content)
- add_scheduler_message_response(task_id, message_id, response)
- get_scheduler_message_response(task_id, message_id)
- has_task_history()
- is_pause_enabled()
- is_paused()
- pause()
- unpause()
- update_resources(**resources)
- update_resource(resource, amount)
- count_pending(worker)
- get_work(host=None, assistant=False, current_tasks=None, worker=None, **kwargs)
- ping(**kwargs)
- graph(**kwargs)
**dep_graph** (task_id, include_done=True, **kwargs)

**inverse_dep_graph** (task_id, include_done=True, **kwargs)

**task_list** (status=", upstream_status=", limit=True, search=None, max_shown_tasks=None, **kwargs)

Query for a subset of tasks by status.

**worker_list** (include_running=True, **kwargs)

**resource_list** ()

Resources usage info and their consumers (tasks).

**resources** ()

get total resources and available ones

**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)

**set_task_status_message** (task_id, status_message)

**get_task_status_message** (task_id)

**set_task_progress_percentage** (task_id, progress_percentage)

**get_task_progress_percentage** (task_id)

**decrease_running_task_resources** (task_id, decrease_resources)

**get_running_task_resources** (task_id)

**task_history**

**update_metrics_task_started** (task)

---

**luigi.server module**

Simple REST server that takes commands in a JSON payload Interface to the **Scheduler** class. See *Using the Central Scheduler* for more info.

**class**  **luigi.server.cors** (*args, **kwargs)

**Bases:**  **luigi.task.Config**

    enabled = BoolParameter (defaults to False): Enables CORS support.

    allow_any_origin = BoolParameter (defaults to False): Accepts requests from any origin.

    allow_null_origin = BoolParameter (defaults to False): Allows the request to set `null` value of the `Origin` header.

    max_age = IntParameter (defaults to 86400): Content of `Access-Control-Max-Age`.

    allowed_methods = Parameter (defaults to GET, OPTIONS): Content of `Access-Control-Allow-Methods`.

    allowed_headers = Parameter (defaults to Accept, Content-Type, Origin): Content of `Access-Control-Allow-Headers`.

    exposed_headers = Parameter (defaults to ): Content of `Access-Control-Expose-Headers`.

    allow_credentials = BoolParameter (defaults to False): Indicates that the actual request can include user credentials.

---

9.1. luigi package
allowed_origins = ListParameter (defaults to []): A list of allowed origins. Used only if `allow_any_origin` is false.

class luigi.server.RPCHandler(*args, **kwargs)
    Bases: tornado.web.RequestHandler


    initialize (scheduler)
        Hook for subclass initialization. Called for each request.
        A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

        Example:

        ```python
        class ProfileHandler(RequestHandler):
            def initialize(self, database):
                self.database = database

            def get(self, username):
                ...

        app = Application([
            (r'/user/(.*)', ProfileHandler, dict(database=database)),
        ])
        ```

    options (*args)

    get (method)

    post (method)

class luigi.server.BaseTaskHistoryHandler (application, request, **kwargs)
    Bases: tornado.web.RequestHandler

    initialize (scheduler)
        Hook for subclass initialization. Called for each request.
        A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

        Example:

        ```python
        class ProfileHandler(RequestHandler):
            def initialize(self, database):
                self.database = database

            def get(self, username):
                ...

        app = Application([
            (r'/user/(.*)', ProfileHandler, dict(database=database)),
        ])
        ```

    get_template_path ()
        Override to customize template path for each handler.
        By default, we use the template_path application setting. Return None to load templates relative to the calling file.

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.RootPathHandler(application, request, **kwargs)
    Bases: Luigi.server.BaseTaskHistoryHandler

    get()
    head()
    HEAD endpoint for health checking the scheduler

class Luigi.server.MetricsHandler(application, request, **kwargs)
    Bases: tornado.web.RequestHandler

    initialize(scheduler)
    Hook for subclass initialization. Called for each request.

    A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

    Example:

    class ProfileHandler(RequestHandler):
        def initialize(self, database):
            self.database = database

        def get(self, username):
            ...

        app = Application([
            (r'/user/(.*)', ProfileHandler, dict(database=database)),
        ])

    get()

luigi.server.app(scheduler)

luigi.server.run(api_port=8082, address=None, unix_socket=None, scheduler=None)
Runs one instance of the API server.

luigi.server.stop()
luigi.setup_logging module

This module contains helper classes for configuring logging for luigid and workers via command line arguments and options from config files.

class luigi.setup_logging.BaseLogging
   Bases: object

classmethod setup(opts=<class 'luigi.setup_logging.opts'>)
   Setup logging via CLI params and config.

class luigi.setup_logging.DaemonLogging
   Bases: luigi.setup_logging.BaseLogging
   Configure logging for luigid

class luigi.setup_logging.InterfaceLogging
   Bases: luigi.setup_logging.BaseLogging
   Configure logging for worker

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.

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.

move(path, dest)
    Move a file, as one would expect.

rename_dont_move(path, dest)
    Potentially rename path to dest, but don’t move it into the dest folder (if it is a folder). This relates to Atomic Writes Problem.

    This method has a reasonable but not bullet proof default implementation. It will just do move() if the file doesn’t exists() already.
rename(*args, **kwargs)
    Alias for move()

copy(path, dest)
    Copy a file or a directory with contents. Currently, LocalFileSystem and MockFileSystem support only
    single file copying but S3Client copies either a file or a directory as required.

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.

    Usage:

    target = FileSystemTarget('~/some_file.txt')
    target = FileSystemTarget(pathlib.Path('~') / 'some_file.txt')
    target.exists()  # False

    Initializes a FileSystemTarget instance.

    Parameters path – 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. Using b is not supported; initialize with format=Nop instead.

    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.

temporary_path(**kwds)
    A context manager that enables a reasonably short, general and magic-less way to solve the Atomic Writes Problem.

    • On entering, it will create the parent directories so the temporary_path is writeable right away. This step uses FileSystem.mkdir().

    • On exiting, it will move the temporary file if there was no exception thrown. This step uses FileSystem.rename_dont_move()

    The file system operations will be carried out by calling them on fs.

    The typical use case looks like this:
class MyTask(luigi.Task):
    def output(self):
        return MyFileSystemTarget(...)

    def run(self):
        with self.output().temporary_path() as self.temp_output_path:
            run_some_external_command(output_path=self.temp_output_path)

class luigi.target.AtomicLocalFile(path):
    Bases: _io.BufferedWriter

    Abstract class to create a Target that creates a temporary file in the local filesystem before moving it to its final destination.

    This class is just for the writing part of the Target. See luigi.local_target.LocalTarget for example

    close()
    Flush and close the IO object.
    This method has no effect if the file is already closed.

    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, scope="")

Call to set namespace of tasks declared after the call.

It is often desired to call this function with the keyword argument scope=__name__.

The scope keyword makes it so that this call is only effective for task classes with a matching __module__.

The default value for scope is the empty string, which means all classes. Multiple calls with the same scope simply replace each other.

The namespace of a Task can also be changed by specifying the property task_namespace.

class Task2(luigi.Task):
    task_namespace = 'namespace2'

This explicit setting takes priority over whatever is set in the namespace() method, and it’s also inherited through normal python inheritance.

There’s no equivalent way to set the task_family.

New since Luigi 2.6.0: scope keyword argument.

See also:

The new and better scaling auto_namespace()

---

0 When there are multiple levels of matching module scopes like a.b vs a.b.c, the more specific one (a.b.c) wins.
luigi.task.auto_namespace(scope="")

Same as namespace(), but instead of a constant namespace, it will be set to the __module__ of the task class. This is desirable for these reasons:

- Two tasks with the same name will not have conflicting task families
- It’s more pythonic, as modules are Python’s recommended way to do namespaces.
- It’s traceable. When you see the full name of a task, you can immediately identify where it is defined.

We recommend calling this function from your package’s outermost __init__.py file. The file contents could look like this:

```python
import luigi
luigi.auto_namespace(scope=__name__)
```

To reset an auto_namespace() call, you can use namespace(scope='my_scope'). But this will not be needed (and is also discouraged) if you use the scope kwarg.

*New since Luigi 2.6.0.*

luigi.task.task_id_str(task_family, params)

Returns a canonical string used to identify a particular task

**Parameters**

- `task_family` – The task family (class name) of the task
- `params` – a dict mapping parameter names to their serialized values

**Returns** A unique, shortened identifier corresponding to the family and params

---

**Exception** luigi.taskBulkCompleteNotImplementedError

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 not an abstract method.

---

**Class** luigi.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.

Each Parameter of the Task should be declared as members:

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

In addition to any declared properties and methods, there are a few non-declared properties, which are created by the Register metaclass:
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 worker-timeout value in config

max_batch_size = inf
Maximum number of tasks to run together as a batch. Infinite by default

batchable
True if this instance can be run as part of a batch. By default, True if it has any batched parameters

retry_count
Override this positive integer to have different retry_count at task level Check [scheduler]

disable_hard_timeout
Override this positive integer to have different disable_hard_timeout at task level. Check [scheduler]

disable_window_seconds
Override this positive integer to have different disable_window_seconds at task level. Check [scheduler]

owner_email
Override this to send out additional error emails to task owner, in addition to the one defined in the global configuration. This should return a string or a list of strings. e.g. ‘test@example.com’ or ['test1@example.com', 'test2@example.com']

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.

accepts_messages
For configuring which scheduler messages can be received. When falsy, this tasks does not accept any message. When True, all messages are accepted.

task_module
Returns what Python module to import to get access to this class.

task_namespace = '__not_user_specified'
This value can be overridden to set the namespace that will be used. (See Namespaces, families and ids) If it’s not specified and you try to read this value anyway, it will return garbage. Please use get_task_namespace() to read the namespace.

Note that setting this value with @property will not work, because this is a class level value.

classmethod get_task_namespace()
The task family for the given class.

Note: You normally don’t want to override this.
task_family = 'Task'

classmethod get_task_family()
    The task family for the given class.
    If task_namespace is not set, then it's simply the name of the class. Otherwise, <task_namespace>. is prefixed to the class name.
    Note: You normally don’t want to override this.

classmethod get_params()
    Returns all of the Parameters for this Task.

classmethod batch_param_names()

classmethod get_param_names(include_significant=False)

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.

param_args

initialized()
    Returns True if the Task is initialized and False otherwise.

classmethod from_str_params(params_str)
    Creates an instance from a str->str hash.
    Parameters params_str – dict of param name -> value as string.

to_str_params(only_significant=False, only_public=False)
    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 exist. 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 if 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 subclass 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(). The returned 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
**no_unpicklable_properties(**kwds)**  
Remove unpicklable properties before dump task and resume them after.

This method could be called in subtask’s dump method, to ensure unpicklable properties won’t break dump.

This method is a context-manager which can be called as below:

```python
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)
```  

```python
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.

    run = None
```

```python
luigi.task.externalize(taskclass_or_taskobject)

    Returns an externalized version of a Task. You may both pass an instantiated task object or a task class. Some examples:

```python
class RequiringTask(luigi.Task):
    def requires(self):
        task_object = self.clone(MyTask)
        return externalize(task_object)
...
```

Here’s mostly equivalent code, but externalize is applied to a task class instead.

```python
@luigi.util.requires(externalize(MyTask))
class RequiringTask(luigi.Task):
    pass
...
```

Of course, it may also be used directly on classes and objects (for example for reexporting or other usage).

```python
MyTask = externalize(MyTask)
my_task_2 = externalize(MyTask2(param='foo'))
```

If you however want a task class to be external from the beginning, you’re better off inheriting ExternalTask rather than Task.

This function tries to be side-effect free by creating a copy of the class or the object passed in and then modify that object. In particular this code shouldn’t do anything.

```python
externalize(MyTask)  # BAD: This does nothing (as after luigi 2.4.0)
```
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.

class luigi.task.Config(*args, **kwargs)
    Bases: luigi.task.Task

    Class for configuration. See Configuration classes.

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.StoredTask(task, status, host=None)
    Bases: object

    Interface for methods on TaskHistory

    task_family

    parameters

class luigi.task_history.TaskHistory
    Bases: object

    Abstract Base Class for updating the run history of a task

    task_scheduled(task)

    task_finished(task, successful)

    task_started(task, worker_host)

class luigi.task_history.NopHistory
    Bases: luigi.task_history.TaskHistory

    task_scheduled(task)
**task.finished** *(task, successful)*

**task.started** *(task, worker_host)*

### Luigi.task_register module

Define the centralized register of all *Task* classes.

**exception** `luigi.task_register.TaskClassException`

Bases: `exceptions.Exception`

**exception** `luigi.task_register.TaskClassNotFoundException`

Bases: `luigi.task_register.TaskClassException`

**exception** `luigi.task_register.TaskClassAmbigiousException`

Bases: `luigi.task_register.TaskClassException`

**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.

Custom class creation for namespaces.

Also register all subclasses.

When the set or inherited namespace evaluates to `None`, set the task namespace to whatever the currently declared namespace is.

**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**

Internal note: This function will be deleted soon.

**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**

**Using inherits and requires to ease parameter pain**

Most luigi plumbers will find themselves in an awkward task parameter situation at some point or another. Consider the following “parameter explosion” problem:

```python
class TaskA(luigi.ExternalTask):
    param_a = luigi.Parameter()

    def output(self):
        return luigi.LocalTarget('/tmp/log-{t.param_a}'.format(t=self))

class TaskB(luigi.Task):
    param_b = luigi.Parameter()
    param_a = luigi.Parameter()

    def requires(self):
        return TaskA(param_a=self.param_a)

class TaskC(luigi.Task):
    param_c = luigi.Parameter()
    param_b = luigi.Parameter()
    param_a = luigi.Parameter()

    def requires(self):
        return TaskB(param_b=self.param_b, param_a=self.param_a)
```

In work flows requiring many tasks to be chained together in this manner, parameter handling can spiral out of control. Each downstream task becomes more burdensome than the last. Refactoring becomes more difficult. There are several ways one might try and avoid the problem.

**Approach 1**: Parameters via command line or config instead of `requires()`.

```python
class TaskA(luigi.ExternalTask):
    param_a = luigi.Parameter()

    def output(self):
        return luigi.LocalTarget('/tmp/log-{t.param_a}'.format(t=self))

class TaskB(luigi.Task):
    param_b = luigi.Parameter()

    def requires(self):
        return TaskA()

class TaskC(luigi.Task):
    param_c = luigi.Parameter()

    def requires(self):
        return TaskB()
```

Then run in the shell like so:
Repetitive parameters have been eliminated, but at the cost of making the job’s command line interface slightly clunkier. Often this is a reasonable trade-off.

But parameters can’t always be refactored out every class. Downstream tasks might also need to use some of those parameters. For example, if TaskC needs to use param_a too, then param_a would still need to be repeated.

**Approach 2:** Use a common parameter class

```python
class Params(luigi.Config):
    param_c = luigi.Parameter()
    param_b = luigi.Parameter()
    param_a = luigi.Parameter()

class TaskA(Params, luigi.ExternalTask):
    def output(self):
        return luigi.LocalTarget('/tmp/log-{t.param_a}'.format(t=self))

class TaskB(Params):
    def requires(self):
        return TaskA()

class TaskB(Params):
    def requires(self):
        return TaskB()
```

This looks great at first glance, but a couple of issues lurk. Now TaskA and TaskB have unnecessary significant parameters. Significant parameters help define the identity of a task. Identical tasks are prevented from running at the same time by the central planner. This helps preserve the idempotent and atomic nature of luigi tasks. Unnecessary significant task parameters confuse a task’s identity. Under the right circumstances, task identity confusion could lead to that task running when it shouldn’t, or failing to run when it should.

This approach should only be used when all of the parameters of the config class, are significant (or all insignificant) for all of its subclasses.

And wait a second… there’s a bug in the above code. See it?

TaskA won’t behave as an ExternalTask because the parent classes are specified in the wrong order. This contrived example is easy to fix (by swapping the ordering of the parents of TaskA), but real world cases can be more difficult to both spot and fix. Inheriting from multiple classes derived from Task should be undertaken with caution and avoided where possible.

**Approach 3:** Use `inherits` and `requires`

The `inherits` class decorator in this module copies parameters (and nothing else) from one task class to another, and avoids direct pythonic inheritance.

```python
import luigi
from luigi.util import inherits

class TaskA(luigi.ExternalTask):
    param_a = luigi.Parameter()

    def output(self):
        return luigi.LocalTarget('/tmp/log-{t.param_a}'.format(t=self))

@inherits(TaskA)
class TaskB(luigi.Task):
```

(continues on next page)
def requires(self):
    t = self.clone(TaskA)  # or t = self.clone_parent()

    # Wait... what's this clone thingy do?
    #
    # Pass it a task class. It calls that task. And when it does, it
    # supplies all parameters (and only those parameters) common to
    # the caller and callee!
    #
    # The call to clone is equivalent to the following (note the
    # fact that clone avoids passing param_b).
    #
    # return TaskA(param_a=self.param_a)

    return t

@inherits(TaskB)
class TaskC(luigi.Task):
    param_c = luigi.Parameter()

    def requires(self):
        return self.clone(TaskB)

This totally eliminates the need to repeat parameters, avoids inheritance issues, and keeps the task command line interface as simple (as it can be, anyway). Refactoring task parameters is also much easier.

The `requires` helper function can reduce this pattern even further. It does everything `inherits` does, and also attaches a `requires` method to your task (still all without pythonic inheritance).

But how does it know how to invoke the upstream task? It uses `clone()` behind the scenes!

import luigi
from luigi.util import inherits, requires

class TaskA(luigi.ExternalTask):
    param_a = luigi.Parameter()

    def output(self):
        return luigi.LocalTarget('/tmp/log-{t.param_a}'.format(t=self))

@requires(TaskA)
class TaskB(luigi.Task):
    param_b = luigi.Parameter()

    # The class decorator does this for me!
    # def requires(self):
    #     return self.clone(TaskA)

Use these helper functions effectively to avoid unnecessary repetition and dodge a few potentially nasty workflow pitfalls at the same time. Brilliant!

import luigi.util.common_params(task_instance, task_cls)

Grab all the values in task_instance that are found in task_cls.

class luigi.util.inherits(*tasks_to_inherit)
    Bases: object
Task inheritance.

_New after Luigi 2.7.6:_ multiple arguments support.

Usage:

```python
class AnotherTask(luigi.Task):
    m = luigi.IntParameter()

class YetAnotherTask(luigi.Task):
    n = luigi.IntParameter()

@inherits(AnotherTask)
class MyFirstTask(luigi.Task):
    def requires(self):
        return self.clone_parent()

    def run(self):
        print self.m # this will be defined # ...

@inherits(AnotherTask, YetAnotherTask)
class MySecondTask(luigi.Task):
    def requires(self):
        return self.clone_parents()

    def run(self):
        print self.n # this will be defined # ...
```

```python
class luigi.util.requires(*tasks_to_require)
Bases: object

Same as inherits, but also auto-defines the requires method.

New after Luigi 2.7.6:_ multiple arguments support.

class luigi.util.copies(task_to_copy)
Bases: object

Auto-copies a task.

Usage:

```python
@copies(MyTask):
class CopyOfMyTask(luigi.Task):
    def output(self):
        return LocalTarget(self.date.strftime('/var/xyz/report-%Y-%m-%d'))
```

```python
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:

```python
class PowersOfN(luigi.Task):
    n = luigi.IntParameter()

    def f(self, x):
        return x ** self.n
```
(continues on next page)
@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, Date-HourParameter 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.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 Scheduler instance. When you run a central server, the worker will talk to the scheduler using a RemoteScheduler instance.

Everything in this module is private to luigi and may change in incompatible ways between versions. The exception is the exception types and the worker config class.

**exception luigi.worker.TaskException**

Bases: exceptions.Exception

class luigi.worker.GetWorkResponse(task_id, running_tasks, n_pending_tasks, n_unique_pending, n_pending_last_scheduled, worker_state)

Bases: tuple

Create new instance of GetWorkResponse(task_id, running_tasks, n_pending_tasks, n_unique_pending, n_pending_last_scheduled, worker_state)

**n_pending_last_scheduled**

Alias for field number 4

**n_pending_tasks**

Alias for field number 2

**n_unique_pending**

Alias for field number 3

**running_tasks**

Alias for field number 1

**task_id**

Alias for field number 0

**worker_state**

Alias for field number 5
```

class luigi.worker.TaskProcess(task, worker_id, result_queue, status_reporter,
                             use_multiprocessing=False, worker_timeout=0,
                             check_unfulfilled_deps=True, check_complete_on_run=False)

Bases: multiprocessing.process.Process

Wrap all task execution in this class.
Mainly for convenience since this is run in a separate process.

forward_reporter_attributes = {'decrease_running_resources': 'decrease_running_resources', ...}

run()
    Method to be run in sub-process; can be overridden in sub-class

terminate()
    Terminate this process and its subprocesses.

class luigi.worker.ContextManagedTaskProcess(context, *args, **kwargs)

Bases: luigi.worker.TaskProcess

run()
    Method to be run in sub-process; can be overridden in sub-class

class luigi.worker.TaskStatusReporter(scheduler, task_id, worker_id, scheduler_messages)

Bases: object

Reports task status information to the scheduler.
This object must be pickle-able for passing to TaskProcess on systems where fork method needs to pickle the process object (e.g. Windows).

update_tracking_url(tracking_url)

update_status_message(message)

update_progress_percentage(percentage)

decrease_running_resources(decrease_resources)

class luigi.worker.SchedulerMessage(scheduler, task_id, message_id, content, **payload)

Bases: object

Message object that is build by the the Worker when a message from the scheduler is received and passed to the message queue of a Task.

respond(response)

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)

close()

join()

class luigi.worker.DequeQueue

Bases: collections.deque
deque wrapper implementing the Queue interface.

put(obj, block=None, timeout=None)

get(block=None, timeout=None)
```
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

   id = Parameter (defaults to ): Override the auto-generated worker_id

   ping_interval = FloatParameter (defaults to 1.0)

   keep_alive = BoolParameter (defaults to False)

   count_uniques = BoolParameter (defaults to False): worker-count-uniques means that we keep a worker alive only if it's the only worker with a pending task.

   count_last_scheduled = BoolParameter (defaults to False): Keep a worker alive only if there are pending tasks which it was the last to schedule.

   wait_interval = FloatParameter (defaults to 1.0)

   wait_jitter = FloatParameter (defaults to 5.0)

   max_keep_alive_idle_duration = TimeDeltaParameter (defaults to 0:00:00)

   max_reschedules = IntParameter (defaults to 1)

   timeout = IntParameter (defaults to 0)

   task_limit = IntParameter (defaults to None)

   retry_external_tasks = BoolParameter (defaults to False): If true, incomplete external tasks will be retested for completion while Luigi is running.

   send_failure_email = BoolParameter (defaults to True): If true, send e-mails directly on failure.

   no_install_shutdown_handler = BoolParameter (defaults to False): If true, the SIGUSR1 shutdown handler will NOT be install on the worker.

   check_unfulfilled_deps = BoolParameter (defaults to True): If true, check for completeness of dependencies before running a task.

   check_complete_on_run = BoolParameter (defaults to False): If true, only mark tasks as complete after they have been run.

   force_multiprocessing = BoolParameter (defaults to False): If true, use multiprocessing also when running with 1 worker.

   task_process_context = OptionalParameter (defaults to None): If set to a fully qualified class name, the class will be used as the context for the run() call, so this can be used for obtaining high level customizable monitoring or logging of each individual Task run.

class luigi.worker.KeepAliveThread (scheduler, worker_id, ping_interval, rpc_message_callback)
   Bases: threading.Thread

   Periodically tell the scheduler that the worker still lives.

   stop ()

   run ()

   Method representing the thread’s activity.

   You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

luigi.worker.rpc_message_callback (fn)
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)

add(task, multiprocess=False, processes=0)
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.

handle_interrupt(signum, _)
Stops the assistant from asking for more work on SIGUSR1

run()
Returns True if all scheduled tasks were executed successfully.

set_worker_processes(n)

dispatch_scheduler_message(task_id, message_id, content, **kwargs)

9.1.3 Module contents

Package containing core luigi functionality.

class luigi.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.

Each Parameter of the Task should be declared as members:

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

In addition to any declared properties and methods, there are a few non-declared properties, which are created by the Register metaclass:

• 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 worker-timeout value in config

max_batch_size = inf
Maximum number of tasks to run together as a batch. Infinite by default

batchable
True if this instance can be run as part of a batch. By default, True if it has any batched parameters

retry_count
Override this positive integer to have different retry_count at task level. Check [scheduler]

disable_hard_timeout
Override this positive integer to have different disable_hard_timeout at task level. Check [scheduler]

disable_window_seconds
Override this positive integer to have different disable_window_seconds at task level. Check [scheduler]

owner_email
Override this to send out additional error emails to task owner, in addition to the one defined in the global configuration. This should return a string or a list of strings. e.g. ‘test@example.com’ or ['test1@example.com', 'test2@example.com']

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.

accepts_messages
For configuring which scheduler messages can be received. When falsy, this tasks does not accept any message. When True, all messages are accepted.

task_module
Returns what Python module to import to get access to this class.

task_namespace = '__not_user_specified'
This value can be overridden to set the namespace that will be used. (See Namespaces, families and ids) If it’s not specified and you try to read this value anyway, it will return garbage. Please use get_task_namespace() to read the namespace.

Note that setting this value with @property will not work, because this is a class level value.

classmethod get_task_namespace()
The task family for the given class.

Note: You normally don’t want to override this.

task_family = 'Task'
classmethod get_task_family()
The task family for the given class.

If task_namespace is not set, then it’s simply the name of the class. Otherwise, <task_namespace> is prefixed to the class name.

Note: You normally don’t want to override this.
classmethod get_params()
    Returns all of the Parameters for this Task.

classmethod batch_param_names()

classmethod get_param_names(include_significant=False)

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.

param_args

initialized()
    Returns True if the Task is initialized and False otherwise.

classmethod from_str_params(params_str)
    Creates an instance from a str->str hash.

    Parameters
    • params_str – dict of param name -> value as string.

to_str_params(only_significant=False, only_public=False)
    Convert all parameters to a str->str hash.

classmethod 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

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 subclass 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().* The returned 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

**no_unpicklable_properties(****kwds)**  
Remove unpicklable properties before dump task and resume them after.

This method could be called in subtask’s dump method, to ensure unpicklable properties won’t break dump.

This method is a context-manager which can be called as below:

```python
class luigi.Config(*args, **kwargs):
    Bases: luigi.task.Task

    Class for configuration. See *Configuration classes.*
```

9.1. Luigi package
class `luigi.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.

```python
run = None
```

class `luigi.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.

```python
complete()
   If the task has any outputs, return True if all outputs exist. Otherwise, return False.
```

However, you may freely override this method with custom logic.

```
lugi.namespace(namespace=None, scope="")
```

Call to set namespace of tasks declared after the call.

It is often desired to call this function with the keyword argument `scope=__name__`.

The `scope` keyword makes it so that this call is only effective for task classes with a matching __module__.
The default value for `scope` is the empty string, which means all classes. Multiple calls with the same scope simply replace each other.

The namespace of a Task can also be changed by specifying the property task_namespace.

```
class Task2(luigi.Task):
   task_namespace = 'namespace2'
```

This explicit setting takes priority over whatever is set in the namespace() method, and it’s also inherited through normal python inheritance.

There’s no equivalent way to set the task family.

**New since Luigi 2.6.0:** `scope` keyword argument.

**See also:**

The new and better scaling `auto_namespace()`

```
lugi.auto_namespace(scope="")
```

Same as namespace(), but instead of a constant namespace, it will be set to the __module__ of the task class. This is desirable for these reasons:

- Two tasks with the same name will not have conflicting task families
- It’s more pythonic, as modules are Python’s recommended way to do naming.
- It’s traceable. When you see the full name of a task, you can immediately identify where it is defined.

We recommend calling this function from your package’s outermost __init__.py file. The file contents could look like this:

```
import luigi

luigi.auto_namespace(scope=__name__)
```

---

When there are multiple levels of matching module scopes like a.b vs a.b.c, the more specific one (a.b.c) wins.
To reset an `auto_namespace()` call, you can use `namespace(scope='my_scope')`. But this will not be needed (and is also discouraged) if you use the `scope` kwarg.

*New since Luigi 2.6.0.*

```python
class luigi.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.

class luigi.LocalTarget (path=None, format=None, is_tmp=False)
    Bases: luigi.target.FileSystemTarget

    fs = <luigi.local_target.LocalFileSystem object>

    makedirs()
    Create all parent folders if they do not exist.

    open(mode='r')
    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. Using b is not supported; initialize with format=Nop instead.

    move (new_path, raise_if_exists=False)

    move_dir (new_path)

    remove()
    Remove the resource at the path specified by this FileSystemTarget.

    This method is implemented by using fs.

    copy (new_path, raise_if_exists=False)

```

```python
class luigi.RemoteScheduler (url='http://localhost:8082/', connect_timeout=None)
    Bases: object

    Scheduler proxy object. Talks to a RemoteSchedulerResponder.

    add_scheduler_message_response (*args, **kwargs)

    add_task (*args, **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_task_batcher(*args, **kwargs)
add_worker(*args, **kwargs)
announce_scheduling_failure(*args, **kwargs)
count_pending(*args, **kwargs)
decrease_running_task_resources(*args, **kwargs)
dep_graph(*args, **kwargs)
disable_worker(*args, **kwargs)
fetch_error(*args, **kwargs)
forgive_failures(*args, **kwargs)
generate_running_task_resources(*args, **kwargs)
generate_scheduler_message_response(*args, **kwargs)
generate_task_progress_percentage(*args, **kwargs)
generate_task_status_message(*args, **kwargs)
generate_work(*args, **kwargs)
grap(*args, **kwargs)
has_task_history(*args, **kwargs)
inverse_dep_graph(*args, **kwargs)
is_pause_enabled(*args, **kwargs)
is_paused(*args, **kwargs)
mark_as_done(*args, **kwargs)
pause(*args, **kwargs)
ping(*args, **kwargs)
prune(*args, **kwargs)
re_enable_task(*args, **kwargs)
resource_list(*args, **kwargs)

Resources usage info and their consumers (tasks).

send_scheduler_message(*args, **kwargs)
set_task_progress_percentage(*args, **kwargs)
set_task_status_message(*args, **kwargs)
set_worker_processes(*args, **kwargs)
task_list(*args, **kwargs)

Query for a subset of tasks by status.

Parameters task_str

Returns
unpause(*args, **kwargs)
update_metrics_task_started(*args, **kwargs)
update_resource(*args, **kwargs)
update_resources(*args, **kwargs)
worker_list(*args, **kwargs)

exception luigi.RPCError(message, sub_exception=None)
Bases: exceptions.Exception
class luigi.Parameter(default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
Bases: object

Parameter whose value is a str, and a base class for other parameter types.
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()
class RequiringTask(luigi.Task):
    def requires(self):
        return MyTask(foo="hello")
    def run(self):
        print(self.requires().foo)  # prints "hello"

This makes it possible to instantiate multiple tasks, e.g. MyTask(foo='bar') and MyTask(foo='baz'). The task will then have the foo attribute set appropriately.

When a task is instantiated, it will first use any argument as the value of the parameter, e.g. if you instantiate a = TaskA(x=44) then a.x == 44. When the value is not provided, the value will be resolved in this order of falling priority:

- Any value provided on the command line:
  - To the root task (e.g. --param xyz)
  - Then to the class, using the qualified task name syntax (e.g. --TaskA-param xyz).
- With [TASK_NAME]>PARAM_NAME: <serialized value> syntax. See Parameters from config Ingestion
- Any default value set using the default flag.

Parameter objects may be reused, but you must then set the positional=False flag.

Parameters

- default – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.
- significant (bool) – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.
• **description** *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the *help* string shown to users. Default: None.

• **config_path** *(dict)* – A dictionary with entries *section* and *name* specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.

• **positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend **positional=False** for abstract base classes and similar cases.

• **always_in_help** *(bool)* – For the –help option in the command line parsing. Set true to always show in –help.

• **batch_method** *(function(iterable[A]–>A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a *ParameterVisibility*. Default value is *ParameterVisibility.PUBLIC*

**has_task_value**(task_name, param_name)

**task_value**(task_name, param_name)

**parse**(x)
  Parse an individual value from the input.
  The default implementation is the identity function, but subclasses should override this method for specialized parsing.
  
  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.

**normalize**(x)
  Given a parsed parameter value, normalizes it.
  
  The value can either be the result of *parse()*, the default value or arguments passed into the task’s constructor by instantiation.
  This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

**next_in Enumeration**(_value)
  If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the *luigi.execution_summary* module for pretty printing purposes. Enabling it to pretty print tasks like `MyTask(num=1), MyTask(num=2), MyTask(num=3)` to `MyTask(num=1,..,3)`.
  
  Parameters value – The value
  
  Returns The next value, like “value + 1”. Or None if there’s no enumerable ordering.
class Luigi Documentation, Release 2.8.13

```python
class luigi.DateParameter(interval=1, start=None, **kwargs)
    Bases: luigi.parameter._DateParameterBase

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.

DateParameters are 90% of the time used to be interpolated into file system paths or the like. Here is a gentle
reminder of how to interpolate date parameters into strings:

```python
class MyTask(luigi.Task):
    date = luigi.DateParameter()

    def run(self):
        templated_path = "/my/path/to/my/dataset/{date:%Y/%m/%d}/"
        instantiated_path = templated_path.format(date=self.date)
        # print(instantiated_path) --> /my/path/to/my/dataset/2016/06/09/
        # ... use instantiated_path ...
```

To set this parameter to default to the current day. You can write code like this:

```python
import datetime

class MyTask(luigi.Task):
    date = luigi.DateParameter(default=datetime.date.today())

date_format = '%Y-%m-%d'
```

```python
next_inEnumeration(value)
    If your Parameter type has an enumerable ordering of values. You can choose to override this method.
    This method is used by the luigi.execution_summary module for pretty printing purposes.
    Enabling it to pretty print tasks like MyTask(num=1), MyTask(num=2), MyTask(num=3) to
    MyTask(num=1..3).

    Parameters value – The value

    Returns The next value, like “value + 1”. Or None if there’s no enumerable ordering.
```

```python
normalize(value)
    Given a parsed parameter value, normalizes it.

    The value can either be the result of parse(), the default value or arguments passed into the task’s construc-
tor by instantiation.

    This is very implementation defined, but can be used to validate/clamp valid values. For example, if you
    wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement
    normalize as x // 2 * 2.
```

```python
class luigi.MonthParameter(interval=1, start=None, **kwargs)
    Bases: luigi.parameter.DateParameter

Parameter whose value is a date, specified to the month (day of date is “rounded” to first of the month).

A MonthParameter is a Date string formatted YYYY-MM. For example, 2013-07 specifies July of 2013. Task
objects constructed from code accept date (ignoring the day value) or Month.

date_format = '%Y-%m'
```

```python
next_inEnumeration(value)
    If your Parameter type has an enumerable ordering of values. You can choose to override this method.
    This method is used by the luigi.execution_summary module for pretty printing purposes.
```
Enabling it to pretty print tasks like `MyTask(num=1), MyTask(num=2), MyTask(num=3)` to `MyTask(num=1..3)`.

**Parameters**

- **value** – The value

**Returns**

The next value, like “value + 1”. Or **None** if there’s no enumerable ordering.

---

**normalize** *(value)*

Given a parsed parameter value, normalizes it.

The value can either be the result of `parse()`, the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement `normalize` as `x // 2 * 2`.

---

**class** `luigi.YearParameter` *(interval=1, start=None, **kwargs)*

**Bases:** `luigi.parameter.DateParameter`  
Parameter whose value is a date, specified to the year (day and month of date is “rounded” to first day of the year).

A YearParameter is a Date string formatted YYYY. Task objects constructed from code accept date (ignoring the month and day values) or `Year`.

**date_format = 'Y'**

---

**next_in_enumeration** *(value)*

If your Parameter type has an enumerable ordering of values. You can choose to override this method.

This method is used by the `luigi.execution_summary` module for pretty printing purposes. Enabling it to pretty print tasks like `MyTask(num=1), MyTask(num=2), MyTask(num=3)` to `MyTask(num=1..3)`.

**Parameters**

- **value** – The value

**Returns**

The next value, like “value + 1”. Or **None** if there’s no enumerable ordering.

---

**normalize** *(value)*

Given a parsed parameter value, normalizes it.

The value can either be the result of `parse()`, the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement `normalize` as `x // 2 * 2`.

---

**class** `luigi.DateHourParameter` *(interval=1, start=None, **kwargs)*

**Bases:** `luigi.parameter._DatetimeParameterBase`  
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'**

---

**class** `luigi.DateMinuteParameter` *(interval=1, start=None, **kwargs)*

**Bases:** `luigi.parameter._DatetimeParameterBase`  
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, `2013-07-10T1907` specifies July 10, 2013 at 19:07.
The interval parameter can be used to clamp this parameter to every N minutes, instead of every minute.

```python
date_format = '%Y-%m-%dT%H%M'
deprecated_date_format = '%Y-%m-%dT%H%M'
parsed s
Parses a string to a datetime.
```

class Luigi.DateSecondParameter (interval=1, start=None, **kwargs)
Bases: Luigi.parameter._DatetimeParameterBase
Parameter whose value is a datetime specified to the second.

A DateSecondParameter is a ISO 8601 formatted date and time specified to the second. For example, 2013-07-10T190738 specifies July 10, 2013 at 19:07:38.

The interval parameter can be used to clamp this parameter to every N seconds, instead of every second.

```python
date_format = '%Y-%m-%dT%H%M%S'
```

class Luigi.DateIntervalParameter (default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
Bases: Luigi.parameter.Parameter
A Parameter whose value is a DateInterval.

Date Intervals are specified using the ISO 8601 date notation for dates (eg. “2015-11-04”), months (eg. “2015-05”), years (eg. “2015”), or weeks (eg. “2015-W35”). In addition, it also supports arbitrary date intervals provided as two dates separated with a dash (eg. “2015-11-04-2015-12-04”).

**Parameters**

- `default` – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.

- `significant` (bool) – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.

- `description` (str) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.

- `config_path` (dict) – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.

- `positional` (bool) – If true, you can set the argument as a positional argument. It’s true by default but we recommend positional=False for abstract base classes and similar cases.

- `always_in_help` (bool) – For the –help option in the command line parsing. Set true to always show in –help.

- `batch_method` (function (iterable[A]) -> A) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See Batching multiple parameter values into a single run
**visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

```python
def parse(s):
    pass
```

Parses a `DateInterval` for details on the parsing of DateIntervals.

See `luigi.date_interval` for details on the parsing of `DateInterval`.

```python
class luigi.TimeDeltaParameter:
    default=<object object>,
    description=None,
    config_path=None,
    positional=True,
    always_in_help=False,
    batch_method=None,
    visibility=<ParameterVisibility.PUBLIC: 0>
```

Bases: `luigi.parameter.Parameter`

Class that maps to timedelta using strings in any of the following forms:

- `n {w[eeek[s]]|d[ay[s]]|h[our[s]]|m[minute[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

**Parameters**

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- **significant** (*bool*) – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- **description** (*str*) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.

- **config_path** (*dict*) – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- **positional** (*bool*) – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- **always_in_help** (*bool*) – For the –help option in the command line parsing. Set true to always show in –help.

- **batch_method** (*function(iterable[A])→A*) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See Batching multiple parameter values into a single run

- **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

```python
def parse(input):
    pass
```

Parses a time delta from the input.

See `TimeDeltaParameter` for details on supported formats.
**serialize**(x)

Converts `datetime.timedelta` to a string

**Parameters**

- `x` – the value to serialize.

**class** `luigi.IntParameter`

```
(default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
```

**Bases:** `luigi.parameter.Parameter`

Parameter whose value is an int.

**Parameters**

- `default` – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- `significant` *(bool)* – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- `description` *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.

- `config_path` *(dict)* – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- `positional` *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- `always_in_help` *(bool)* – For the `--help` option in the command line parsing. Set true to always show in `--help`.

- `batch_method` *(function(iterable[A] -> A))* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*.

- `visibility` – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

**parse**(s)

Parses an `int` from the string using `int()`.

**next_in_enumeration**(value)

If your Parameter type has an enumerable ordering of values. You can choose to override this method. This method is used by the `luigi.execution_summary` module for pretty printing purposes. Enabling it to pretty print tasks like `MyTask(num=1)`, `MyTask(num=2)`, `MyTask(num=3)` to `MyTask(num=1..3)`.

**Parameters**

- `value` – The value

**Returns**

The next value, like “value + 1”. Or `None` if there’s no enumerable ordering.

**class** `luigi.FloatParameter`

```
(default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
```

**Bases:** `luigi.parameter.Parameter`
Parameter whose value is a float.

Parameters

- **default** – the default value for this parameter. This should match the type of the parameter, i.e. `datetime.date` for `DateTimeParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- **significant** *(bool)* – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

- **description** *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the *help* string shown to users. Default: `None`.

- **config_path** *(dict)* – a dictionary with entries *section* and *name* specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

- **positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

- **always_in_help** *(bool)* – For the --help option in the command line parsing. Set true to always show in --help.

- **batch_method** *(function(iterable[A])→A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See Batching multiple parameter values into a single run

- **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

```python
parse(s)
```

Parses a float from the string using `float()`.

```python
class Luigi.BoolParameter(*args, **kwargs)
```

Bases: `luigi.parameter.Parameter`

A Parameter whose value is a bool. This parameter has an implicit default value of `False`. For the command line interface this means that the value is `False` unless you add "--the-bool-parameter" to your command without giving a parameter value. This is considered *implicit* parsing (the default). However, in some situations one might want to give the explicit bool value ("--the-bool-parameter true|false"), e.g. when you configure the default value to be `True`. This is called *explicit* parsing. When omitting the parameter value, it is still considered `True` but to avoid ambiguities during argument parsing, make sure to always place bool parameters behind the task family on the command line when using explicit parsing.

You can toggle between the two parsing modes on a per-parameter base via

```python
class MyTask(luigi.Task):
```

or globally by

```python
```
for all bool parameters instantiated after this line.

```
IMPLICIT_PARSING = 'implicit'
EXPLICIT_PARSING = 'explicit'
parsing = 'implicit'
```

```python
parse(val)
```

Parses a bool from the string, matching ‘true’ or ‘false’ ignoring case.

```python
normalize(value)
```

Given a parsed parameter value, normalizes it.

The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as `x // 2 * 2`.

```python
class luigi.TaskParameter(default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
```

Bases: `luigi.parameter.Parameter`

A parameter that takes another luigi task class.

When used programatically, the parameter should be specified directly with the `luigi.task.Task` (sub) class. Like `MyMetaTask(my_task_param=my_tasks.MyTask)`. On the command line, you specify the `luigi.task.Task.get_task_family()`.

```
$ luigi --module my_tasks MyMetaTask --my_task_param my_namespace.MyTask
```

Where `my_namespace.MyTask` is defined in the `my_tasks` python module.

When the `luigi.task.Task` class is instantiated to an object. The value will always be a task class (and not a string).

**Parameters**

- `default` – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- `significant` (bool) – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.

- `description` (str) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.

- `config_path` (dict) – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.

- `positional` (bool) – If true, you can set the argument as a positional argument. It’s true by default but we recommend positional=False for abstract base classes and similar cases.
• **always_in_help** *(bool)* – For the --help option in the command line parsing. Set true to always show in --help.

• **batch_method** *(function(iterable[A])→A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See [Batching multiple parameter values into a single run](#).

• **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`

### parse *(input)*
Parse a task family using the `Register`.

### serialize *(cls)*
Converts the `luigi.task.Task` (sub) class to its family name.

```python
class luigi.ListParameter
    default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
```

Bases: `luigi.parameter.Parameter`

Parameter whose value is a list.

In the task definition, use

```python
class MyTask(luigi.Task):
    grades = luigi.ListParameter()

    def run(self):
        sum = 0
        for element in self.grades:
            sum += element
        avg = sum / len(self.grades)
```

At the command line, use

```
$ luigi --module my_tasks MyTask --grades <JSON string>
```

Simple example with two grades:

```
$ luigi --module my_tasks MyTask --grades '[100,70]'
```

**Parameters**

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

- **significant** *(bool)* – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.

- **description** *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.

- **config_path** *(dict)* – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.
• **positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

• **always_in_help** *(bool)* – For the –help option in the command line parsing. Set true to always show in –help.

• **batch_method** *(function(iterable[A]→A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`

```python
normalize(x)
```
Ensure that struct is recursively converted to a tuple so it can be hashed.

Parameters

- **x** *(str)* – the value to parse.

Returns the normalized (hashable/immutable) value.

```python
def parse(x)
```
Parse an individual value from the input.

Parameters

- **x** *(str)* – the value to parse.

Returns the parsed value.

```python
serialize(x)
```
Opposite of `parse()`.

Converts the value `x` to a string.

Parameters

- **x** – the value to serialize.

---

```python
class luigi.TupleParameter(
default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>))
```
Bases: `luigi.parameter.ListParameter`

Parameter whose value is a tuple or tuple of tuples.

In the task definition, use

```python
class MyTask(luigi.Task):
    book_locations = luigi.TupleParameter()

    def run(self):
        for location in self.book_locations:
            print("Go to page %d, line %d" % (location[0], location[1]))
```

At the command line, use

```bash
$ luigi --module my_tasks MyTask --book_locations <JSON string>
```

Simple example with two grades:

```bash
$ luigi --module my_tasks MyTask --book_locations '[(12,3),(4,15),(52,1)]'
```

Parameters
**default** – the default value for this parameter. This should match the type of the Parameter, i.e. `datetime.date` for `DateParameter` or `int` for `IntParameter`. By default, no default is stored and the value must be specified at runtime.

**significant** *(bool)* – specify `False` if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: `True`.

**description** *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the `help` string shown to users. Default: `None`.

**config_path** *(dict)* – a dictionary with entries `section` and `name` specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: `None`.

**positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend `positional=False` for abstract base classes and similar cases.

**always_in_help** *(bool)* – For the `-help` option in the command line parsing. Set true to always show in `-help`.

**batch_method** *(function(iterable[A] -> A)*) – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

**visibility** – A Parameter whose value is a `ParameterVisibility`. Default value is `ParameterVisibility.PUBLIC`.

### parse(x)

Parse an individual value from the input.

**Parameters**

- **x (str)** – the value to parse.

**Returns**

the parsed value.

### class `luigi.EnumParameter`(*args, **kwargs*)

**Bases:** `luigi.parameter.Parameter`

A parameter whose value is an `Enum`.

In the task definition, use

```python
class Model(enum.Enum):
    Honda = 1
    Volvo = 2

class MyTask(luigi.Task):
    my_param = luigi.EnumParameter(enum=Model)
```

At the command line, use,

```
$ luigi --module my_tasks MyTask --my-param Honda
```

### parse(s)

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.
Parameters \( x \) (str) – the value to parse.

Returns the parsed value.

\[
\text{serialize}(e) \\
\text{Opposite of parse().}
\]

Converts the value \( x \) to a string.

Parameters \( x \) – the value to serialize.

```
class luigi.DictParameter (default=<object object>, is_global=False, significant=True, description=None, config_path=None, positional=True, always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
```

Bases: luigi.parameter.Parameter

Parameter whose value is a dict.

In the task definition, use

```
class MyTask(luigi.Task):
    tags = luigi.DictParameter()

    def run(self):
        logging.info("Find server with role: \$s", self.tags['role'])
        server = aws.ec2.find_my_resource(self.tags)
```

At the command line, use

```
$ luigi --module my_tasks MyTask --tags <JSON string>
```

Simple example with two tags:

```
$ luigi --module my_tasks MyTask --tags '{"role": "web", "env": "staging"}'
```

It can be used to define dynamic parameters, when you do not know the exact list of your parameters (e.g. list of tags, that are dynamically constructed outside Luigi), or you have a complex parameter containing logically related values (like a database connection config).

Parameters

- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.

- **significant** (bool) – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.

- **description** (str) – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.

- **config_path** (dict) – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.

- **positional** (bool) – If true, you can set the argument as a positional argument. It’s true by default but we recommend positional=False for abstract base classes and similar cases.
Luigi Documentation, Release 2.8.13

- **always_in_help** *(bool)* – For the -help option in the command line parsing. Set true to always show in -help.

- **batch_method** *(function(iterable[A]) -> A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

- **visibility** – A Parameter whose value is a *ParameterVisibility*. Default value is ParameterVisibility.PUBLIC

**normalize**(value)
Ensure that dictionary parameter is converted to a FrozenOrderedDict so it can be hashed.

**parse**(source)
Parses an immutable and ordered *dict* from a JSON string using standard JSON library.

We need to use an immutable dictionary, to create a hashable parameter and also preserve the internal structure of parsing. The traversal order of standard *dict* is undefined, which can result various string representations of this parameter, and therefore a different task id for the task containing this parameter. This is because task id contains the hash of parameters’ JSON representation.

**Parameters**

- **s** – String to be parse

**serialize**(x)
Opposite of **parse()**.

Converts the value x to a string.

**Parameters**

- **x** – the value to serialize.

**luigi.run**(>*args, **kwargs*)
Please dont use. Instead use luigi binary.

Run from cmdline using argparse.

**Parameters**

- **use_dynamic_argparse** – Deprecated and ignored

**luigi.build**(tasks, worker_scheduler_factory=None, detailed_summary=False, **env_params*)
Run internally, bypassing the cmdline parsing.

Useful if you have some luigi code that you want to run internally. Example:

```python
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** True if there were no scheduling errors, even if tasks may fail.

class **luigi.Event**
Bases: object

DEPENDECY_DISCOVERED = 'event.core.dependency.discovered'

DEPENDECY_MISSING = 'event.core.dependency.missing'

DEPENDECY_PRESENT = 'event.core.dependency.present'
BROKEN_TASK = 'event.core.task.broken'
START = 'event.core.start'
PROGRESS = 'event.core.progress'
This event can be fired by the task itself while running. The purpose is for the task to report progress,
metadata or any generic info so that event handler listening for this can keep track of the progress of
running task.
FAILURE = 'event.core.failure'
SUCCESS = 'event.core.success'
PROCESSING_TIME = 'event.core.processing_time'
TIMEOUT = 'event.core.timeout'
PROCESS_FAILURE = 'event.core.process_failure'
class luigi.NumericalParameter(left_op=<built-in function le>, right_op=<built-in function lt>,
*args,**kwargs)
Bases: luigi.parameter.Parameter
Parameter whose value is a number of the specified type, e.g. int or float and in the range specified.
In the task definition, use
class MyTask(luigi.Task):
    my_param_1 = luigi.NumericalParameter(var_type=int, min_value=-3, max_value=7)  # -3 <= my_param_1 < 7
    my_param_2 = luigi.NumericalParameter(var_type=int, min_value=-3, max_value=7, left_op=operator.lt, right_op=operator.le)  # -3 < my_param_2 <= 7
At the command line, use
$ luigi --module my_tasks MyTask --my-param-1 -3 --my-param-2 -2

Parameters
• var_type (function) – The type of the input variable, e.g. int or float.
• min_value – The minimum value permissible in the accepted values range. May be
  inclusive or exclusive based on left_op parameter. This should be the same type as var_type.
• max_value – The maximum value permissible in the accepted values range. May be
  inclusive or exclusive based on right_op parameter. This should be the same type as var_type.
• left_op (function) – The comparison operator for the left-most comparison in the
  expression min_value left_op value right_op value. This operator should
generally be either operator.lt or operator.le. Default: operator.le.
• right_op (function) – The comparison operator for the right-most comparison in the
  expression min_value left_op value right_op value. This operator should
generally be either operator.lt or operator.le. Default: operator.lt.

parse(s)
Parse an individual value from the input.
The default implementation is the identity function, but subclasses should override this method for spe-
cialized parsing.
Parameters x(str) – the value to parse.
Returns the parsed value.

class Luigi.ChoiceParameter(var_type=<type 'str'>, *args, **kwargs)
Bases: Luigi.parameter.Parameter

A parameter which takes two values:
1. an instance of Iterable and
2. the class of the variables to convert to.

In the task definition, use

class MyTask(Luigi.Task):
    my_param = Luigi.ChoiceParameter(choices=[0.1, 0.2, 0.3], var_type=float)

At the command line, use

$ Luigi --module my_tasks MyTask --my-param 0.1

Consider using Luigi.EnumParameter for a typed, structured alternative. This class can perform the same role when all choices are the same type and transparency of parameter value on the command line is desired.

Parameters
- **var_type** (function) – The type of the input variable, e.g. str, int, float, etc. Default: str
- **choices** – An iterable, all of whose elements are of var_type to restrict parameter choices to.

**parse**(s)
Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

**Parameters** x (str) – the value to parse.

**Returns** the parsed value.

**normalize**(var)
Given a parsed parameter value, normalizes it.

The value can either be the result of parse(), the default value or arguments passed into the task’s constructor by instantiation.

This is very implementation defined, but can be used to validate/clamp valid values. For example, if you wanted to only accept even integers, and “correct” odd values to the nearest integer, you can implement normalize as x // 2 * 2.

class Luigi.OptionalParameter(default=<object object>, is_global=False, significant=True,
description=None, config_path=None, positional=True,
always_in_help=False, batch_method=None, visibility=<ParameterVisibility.PUBLIC: 0>)
Bases: Luigi.parameter.Parameter

A Parameter that treats empty string as None

Parameters
- **default** – the default value for this parameter. This should match the type of the Parameter, i.e. datetime.date for DateParameter or int for IntParameter. By default, no default is stored and the value must be specified at runtime.
• **significant** *(bool)* – specify False if the parameter should not be treated as part of the unique identifier for a Task. An insignificant Parameter might also be used to specify a password or other sensitive information that should not be made public via the scheduler. Default: True.

• **description** *(str)* – A human-readable string describing the purpose of this Parameter. For command-line invocations, this will be used as the help string shown to users. Default: None.

• **config_path** *(dict)* – a dictionary with entries section and name specifying a config file entry from which to read the default value for this parameter. DEPRECATED. Default: None.

• **positional** *(bool)* – If true, you can set the argument as a positional argument. It’s true by default but we recommend positional=False for abstract base classes and similar cases.

• **always_in_help** *(bool)* – For the –help option in the command line parsing. Set true to always show in –help.

• **batch_method** *(function(iterable[A])->A)* – Method to combine an iterable of parsed parameter values into a single value. Used when receiving batched parameter lists from the scheduler. See *Batching multiple parameter values into a single run*

• **visibility** – A Parameter whose value is a ParameterVisibility. Default value is ParameterVisibility.PUBLIC

**serialize** *(x)*

Opposite of **parse**().

Converts the value x to a string.

**Parameters**

- x – the value to serialize.

**parse** *(x)*

Parse an individual value from the input.

The default implementation is the identity function, but subclasses should override this method for specialized parsing.

**Parameters**

- x *(str)* – the value to parse.

**Returns**

the parsed value.

class luigi.LuigiStatusCode

Bases: enum.Enum

All possible status codes for the attribute status in LuigiRunResult when the argument detailed_summary=True in luigi.run()/luigi.build. Here are the codes and what they mean:

<table>
<thead>
<tr>
<th>Status Code Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESS</td>
<td>There were no failed tasks or missing dependencies</td>
</tr>
<tr>
<td>SUCCESS_WITH_RETRY</td>
<td>There were failed tasks but they all succeeded in a retry</td>
</tr>
<tr>
<td>FAILED</td>
<td>There were failed tasks</td>
</tr>
<tr>
<td>FAILED_AND_SCHEDULING_FAILED</td>
<td>There were failed tasks and tasks whose scheduling failed</td>
</tr>
<tr>
<td>SCHEDULING_FAILED</td>
<td>There were tasks whose scheduling failed</td>
</tr>
<tr>
<td>NOT_RUN</td>
<td>There were tasks that were not granted run permission by the scheduler</td>
</tr>
<tr>
<td>MISSING_EXT</td>
<td>There were missing external dependencies</td>
</tr>
</tbody>
</table>

```
SUCCESS = ('\:)', 'there were no failed tasks or missing dependencies')
```
SUCCESS_WITH_RETRY = (':)', 'there were failed tasks but they all succeeded in a retry')
FAILED = (':(', 'there were failed tasks')
FAILED_AND_SCHEDULING_FAILED = (':(', 'there were failed tasks and tasks whose scheduling failed')
SCHEDULING_FAILED = (':(', 'there were tasks whose scheduling failed')
NOT_RUN = (':|', 'there were tasks that were not granted run permission by the scheduler')
MISSING_EXT = (':|', 'there were missing external dependencies')

9.2 Indices and tables

- genindex
- modindex
- search
Python Module Index
luigi.mock, 176
luigi.notifications, 177
luigi.parameter, 179
luigi.process, 196
luigi.retcodes, 196
luigi.rpc, 197
luigi.scheduler, 198
luigi.server, 203
luigi.setup_logging, 206
luigi.target, 206
luigi.task, 209
luigi.task_history, 215
luigi.task_register, 216
luigi.task_status, 217
luigi.tools, 163
luigi.tools.deps, 157
luigi.tools.deps_tree, 157
luigi.tools.luigi_grep, 158
luigi.tools.range, 158
luigi.util, 217
luigi.worker, 221
<table>
<thead>
<tr>
<th>Method/Function</th>
<th>Module/Class</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abort()</code></td>
<td><code>luigi.contrib.hdfs.format.HdfsAtomicWriteDirPipe</code> method</td>
<td>69</td>
</tr>
<tr>
<td><code>abort()</code></td>
<td><code>luigi.contrib.hdfs.format.HdfsAtomicWritePipe</code> method</td>
<td>69</td>
</tr>
<tr>
<td><code>abort()</code></td>
<td><code>luigi.format.OutputPipeProcessWrapper</code> method</td>
<td>170</td>
</tr>
<tr>
<td><code>abort_job()</code></td>
<td><code>luigi.contrib.salesforce.SalesforceAPI</code> method</td>
<td>143</td>
</tr>
<tr>
<td><code>AbstractPySparkRunner</code></td>
<td>(class in <code>luigi.contrib.pyspark_runner</code>)</td>
<td>129</td>
</tr>
<tr>
<td><code>accept_trailing_slash()</code></td>
<td>(in module <code>luigi.contrib.dropbox</code>)</td>
<td>91</td>
</tr>
<tr>
<td><code>accept_trailing_slash_in_existing_dirpaths()</code></td>
<td>(in module <code>luigi.contrib.dropbox</code>)</td>
<td>91</td>
</tr>
<tr>
<td><code>accepts_messages</code></td>
<td><code>luigi.Task</code> attribute</td>
<td>225</td>
</tr>
<tr>
<td><code>accepts_messages</code></td>
<td><code>luigi.task.Task</code> attribute</td>
<td>211</td>
</tr>
<tr>
<td><code>acquire_for()</code></td>
<td>(in module <code>luigi.lock</code>)</td>
<td>175</td>
</tr>
<tr>
<td><code>active_deadline_seconds</code></td>
<td><code>luigi.contrib.kubernetes.KubernetesJobTask</code> attribute</td>
<td>113</td>
</tr>
<tr>
<td><code>add()</code></td>
<td><code>luigi.contrib.opener.OpenerRegistry</code> method</td>
<td>119</td>
</tr>
<tr>
<td><code>add()</code></td>
<td><code>luigi.scheduler.OrderedSet</code> method</td>
<td>200</td>
</tr>
<tr>
<td><code>add()</code></td>
<td><code>luigi.worker.Worker</code> method</td>
<td>224</td>
</tr>
<tr>
<td><code>add_config_path()</code></td>
<td>(in module <code>luigi.configuration</code>)</td>
<td>65</td>
</tr>
<tr>
<td><code>add_config_path()</code></td>
<td>(in module <code>luigi.configuration.core</code>)</td>
<td>65</td>
</tr>
<tr>
<td><code>add_config_path()</code></td>
<td>(in module <code>luigi.configuration.base_parser.BaseParser</code> class method)</td>
<td>63</td>
</tr>
<tr>
<td><code>add_disable()</code></td>
<td><code>luigi.batch_notifier.BatchNotifier</code> method</td>
<td>164</td>
</tr>
<tr>
<td><code>add_failure()</code></td>
<td><code>luigi.batch_notifier.BatchNotifier</code> method</td>
<td>164</td>
</tr>
<tr>
<td><code>add_failure()</code></td>
<td><code>luigi.scheduler.Failures</code> method</td>
<td>200</td>
</tr>
<tr>
<td><code>add_failure()</code></td>
<td><code>luigi.scheduler.Task</code> method</td>
<td>199</td>
</tr>
<tr>
<td><code>add_failure()</code></td>
<td><code>luigi.scheduler.Task</code> method</td>
<td>200</td>
</tr>
<tr>
<td><code>add_info()</code></td>
<td><code>luigi.scheduler.Worker</code> method</td>
<td>200</td>
</tr>
<tr>
<td><code>add_link()</code></td>
<td><code>luigi.contrib.hadoop.JobTask</code> method</td>
<td>108</td>
</tr>
<tr>
<td><code>add_rpc_message()</code></td>
<td><code>luigi.scheduler.Worker</code> method</td>
<td>200</td>
</tr>
<tr>
<td><code>add_scheduler_message_response()</code></td>
<td><code>luigi.RemoteScheduler</code> method</td>
<td>229</td>
</tr>
<tr>
<td><code>add_scheduler_message_response()</code></td>
<td><code>luigi.rpc.RemoteScheduler</code> method</td>
<td>197</td>
</tr>
<tr>
<td><code>add_scheduler_message_response()</code></td>
<td><code>luigi.scheduler.Scheduler</code> method</td>
<td>202</td>
</tr>
<tr>
<td><code>add_scheduling_fail()</code></td>
<td><code>luigi.batch_notifier.BatchNotifier</code> method</td>
<td>164</td>
</tr>
<tr>
<td><code>add_task()</code></td>
<td><code>luigi.RemoteScheduler</code> method</td>
<td>229</td>
</tr>
<tr>
<td><code>add_task()</code></td>
<td><code>luigi.rpc.RemoteScheduler</code> method</td>
<td>197</td>
</tr>
<tr>
<td><code>add_task()</code></td>
<td><code>luigi.scheduler.Scheduler</code> method</td>
<td>202</td>
</tr>
<tr>
<td><code>add_task()</code></td>
<td><code>luigi.RemoteScheduler</code> method</td>
<td>230</td>
</tr>
<tr>
<td><code>add_task()</code></td>
<td><code>luigi.rpc.RemoteScheduler</code> method</td>
<td>197</td>
</tr>
<tr>
<td><code>add_task()</code></td>
<td><code>luigi.scheduler.Scheduler</code> method</td>
<td>202</td>
</tr>
<tr>
<td><code>add_worker()</code></td>
<td><code>luigi.RemoteScheduler</code> method</td>
<td>230</td>
</tr>
<tr>
<td><code>add_worker()</code></td>
<td><code>luigi.rpc.RemoteScheduler</code> method</td>
<td>197</td>
</tr>
<tr>
<td><code>add_worker()</code></td>
<td><code>luigi.scheduler.Scheduler</code> method</td>
<td>202</td>
</tr>
<tr>
<td><code>ALL_METHOD_NAMES</code></td>
<td><code>luigi.contrib.target.CascadingClient</code> attribute</td>
<td>156</td>
</tr>
<tr>
<td><code>allow_any_origin</code></td>
<td><code>luigi.server.cors attribute</code></td>
<td>203</td>
</tr>
<tr>
<td><code>allow_credentials</code></td>
<td><code>luigi.server.cors attribute</code></td>
<td>203</td>
</tr>
<tr>
<td><code>allow_jagged_rows</code></td>
<td><code>luigi.contrib.bigquery.BigQueryLoadTask</code> attribute</td>
<td>85</td>
</tr>
<tr>
<td><code>allow_null_origin</code></td>
<td><code>luigi.server.cors attribute</code></td>
<td>203</td>
</tr>
<tr>
<td><code>allow_quoted_new_lines</code></td>
<td><code>luigi.contrib.bigquery.BigQueryLoadTask</code> attribute</td>
<td>203</td>
</tr>
</tbody>
</table>
BaseTaskHistoryHandler (class in luigi.server), 204
BaseWrapper (class in luigi.format), 170
BATCH (luigi.contrib.bigquery.QueryMode attribute), 81
batch_counter_default (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 106
batch_email (class in luigi.batch_notifier), 163
batch_emails (luigi.schedular.schedular attribute), 199
batch_mode (luigi.batch_notifier.batch_email attribute), 226
batch_param_names () (luigi.Task class method), 212
batchable (luigi.Task attribute), 225
batchable (luigi.task.Task attribute), 211
BatchClient (class in luigi.contrib.batch)
BatchJobException, 77
BatchNotifier (class in luigi.batch_notifier), 164
BatchTask (class in luigi.contrib.batch), 78
bcolors (class in luigi.tools.deps_tree), 158
BeamDataflowJobTask (class luigi.contrib.beam_dataflow), 79
before_get () (luigi.configuration.cfg_parser.combined_interpolation method), 64
before_get () (luigi.configuration.cfg_parser.environment_interpolation method), 214
before_read () (luigi.configuration.cfg_parser.combined_interpolation method), 64
before_run () (luigi.contrib.beam_dataflow.beam_dataflow_job_task method), 80
beginAt (luigi.contrib.pai.Port attribute), 122
BigQueryClient (class in luigi.contrib.bigquery), 82
BigQueryClient (class in module luigi.contrib.bigquery), 87
BigQueryCreateViewTask (class luigi.contrib.bigquery), 86
BigQueryCreateViewTask (class luigi.contrib.bigquery), 87
BigQueryExtractTask (class luigi.contrib.bigquery), 86
BigQueryLoadAvro (class luigi.contrib.bigquery.avro), 87
BigQueryLoadTask (class in luigi.contrib.bigquery), 84
BigQueryLoadTask (class in luigi.contrib.bigquery), 87
BigQueryRunQueryTask (class in luigi.contrib.bigquery), 87
BigQueryRunQueryTask (in module luigi.contrib.bigquery), 85
BigQueryTarget (class in luigi.contrib.bigquery), 87
BigQueryTarget (in module luigi.contrib.bigquery), 87
BigQueryTarget (class in luigi.contrib.bigquery), 87
binds (luigi.contrib.docker_runner.docker_task attribute), 90
block_on_batch () (luigi.contrib.salesforce.salesforceapi method), 143
BoolParameter (class in luigi), 238
BoolParameter (class in luigi.parameter), 186
BQDataset (class in luigi.contrib.bigquery), 82
BQTable (class in luigi.contrib.bigquery), 82
BROKEN_TASK (luigi.Event attribute), 244
BROKEN_TASK (luigi.event.Event attribute), 168
build () (in module luigi), 244
build () (in module luigi.interface), 173
build_job_jar () (luigi.contrib.scalding.scalding_job_runner method), 144
build_tracking_url () (luigi.contrib.external_program.external_program_task method), 99
in bulk_complete () (luigi.contrib.bigquery.mixin_big_query_bulk_complete class method), 84
bulk_complete () (luigi.Task class method), 226
bulk_complete () (luigi.task.mixin_naive_bulk_complete method), 212
bulk_complete () (luigi.contrib.mysqldb.copy_to_table attribute), 119
bulk_complete () (luigi.contrib.hdfs.hadoopcli_clients.hdfs_client attribute), 210
BulkCompleteNotImplementedError, 210
ByIdHandler (class in luigi.server), 205
ByParamHandler (class in luigi.server), 205
ByParamHandler (in module luigi.server), 205
check_complete() (luigi.contrib.hive.hive_query_task attribute), 110
BigQueryClient (class in luigi.contrib.bigquery), 82
Bzip2Format (class in luigi.format), 172
C
call_check () (luigi.contrib.hadoop.hadoopcli_clients.hdfs_client static method), 70
capture_output (luigi.contrib.external_program.external_program_task attribute), 99
CascadingClient (class in luigi.contrib.external_program.external_program_task), 156
catalog (luigi.contrib.presto.presto_attribute), 127
catalog (luigi.contrib.presto.presto_task_attribute), 128
ChainFormat (class in luigi.format), 171
check_complete() (in module luigi.worker), 223
check_complete_on_run (luigi.worker.worker_attribute), 223
check_output () (luigi.contrib.ssh.remote_context method), 155

Index 253
Luigi Documentation, Release 2.8.13

check_pid() (in module luigi.process), 196
close() (luigi.format.InputPipeProcessWrapper method), 170
close() (luigi.format.OutputPipeProcessWrapper method), 170
close() (luigi.worker.worker attribute), 223
chown() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 67
cmd_line_runner (luigi.contrib.beam_dataflow.BeanDataflowJobTask attribute), 80
ChoiceParameter (class in luigi), 246
code_dir (luigi.contrib.pai.PaiTask attribute), 124
codeDir (class in luigi.parameter), 195
column_separator (luigi.contrib.sqla.CopyToTable attribute), 130
column_separator (luigi.contrib.sqla.CopyToTable attribute), 130
chunk_size (luigi.contrib.esindex.CopyToIndex attribute), 97
chunk_size (class in luigi.contrib.sqla.CopyToTable attribute), 153
cleanup_on_error() (luigi.contrib.beam_dataflow.BeanDataflowJobTask method), 81
clear() (luigi.mock.MockFileSystem method), 177
clear() (luigi.scheduler.Failsures method), 200
clear_instance_cache() (luigi.task_register.Register class method), 216
client (luigi.contrib.hdfs.config.hdfs attribute), 68
client (luigi.contrib.hdfs.webhdfs_client.WebHdfsClient attribute), 75
client_type (luigi.contrib.hdfs.webhdfs_client.webhdfs attribute), 75
client_version (luigi.contrib.hdfs.config.hdfs attribute), 68
clone() (luigi.Task method), 226
clone() (luigi.task.Task method), 212
close() (luigi.contrib.dropbox.ReadableDropboxFile method), 92
close() (luigi.contrib.hdfs.format.HdfsAtomicWriteDirPipe method), 69
close() (luigi.contrib.hdfs.format.HdfsAtomicWritePipe method), 69
close() (luigi.contrib.s3.ReadableS3File method), 139
close() (luigi.contrib.ssh.AtomicRemoteFileWriter method), 155
close() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 156

Luigi Documentation, Release 2.8.13

check_pid() (luigi.rpc.RequestsFetcher method), 197
close() (luigi.worker.worker attribute), 223
chmod() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 67
cmd_line_runner (luigi.contrib.beam_dataflow.BeanDataflowJobTask attribute), 80
ChoiceParameter (class in luigi), 246
code_dir (luigi.contrib.pai.PaiTask attribute), 124
codeDir (class in luigi.parameter), 195
column_separator (luigi.contrib.sqla.CopyToTable attribute), 130
column_separator (luigi.contrib.sqla.CopyToTable attribute), 130
chunk_size (luigi.contrib.esindex.CopyToIndex attribute), 97
chunk_size (class in luigi.contrib.sqla.CopyToTable attribute), 153
cleanup_on_error() (luigi.contrib.beam_dataflow.BeanDataflowJobTask method), 81
clear() (luigi.mock.MockFileSystem method), 177
clear() (luigi.scheduler.Failsures method), 200
clear_instance_cache() (luigi.task_register.Register class method), 216
client (luigi.contrib.hdfs.config.hdfs attribute), 68
client (luigi.contrib.hdfs.webhdfs_client.WebHdfsClient attribute), 75
client_type (luigi.contrib.hdfs.webhdfs_client.webhdfs attribute), 75
client_version (luigi.contrib.hdfs.config.hdfs attribute), 68
clone() (luigi.Task method), 226
clone() (luigi.task.Task method), 212
close() (luigi.contrib.dropbox.ReadableDropboxFile method), 92
close() (luigi.contrib.hdfs.format.HdfsAtomicWriteDirPipe method), 69
close() (luigi.contrib.hdfs.format.HdfsAtomicWritePipe method), 69
close() (luigi.contrib.s3.ReadableS3File method), 139
close() (luigi.contrib.ssh.AtomicRemoteFileWriter method), 155
close() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 156

Luigi Documentation, Release 2.8.13

check_pid() (in module luigi.process), 196
close() (luigi.format.InputPipeProcessWrapper method), 170
close() (luigi.format.OutputPipeProcessWrapper method), 170
close() (luigi.worker.worker attribute), 223
chown() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 67
cmd_line_runner (luigi.contrib.beam_dataflow.BeanDataflowJobTask attribute), 80
ChoiceParameter (class in luigi), 246
code_dir (luigi.contrib.pai.PaiTask attribute), 124
codeDir (class in luigi.parameter), 195
column_separator (luigi.contrib.sqla.CopyToTable attribute), 130
column_separator (luigi.contrib.sqla.CopyToTable attribute), 130
chunk_size (luigi.contrib.esindex.CopyToIndex attribute), 97
chunk_size (class in luigi.contrib.sqla.CopyToTable attribute), 153
cleanup_on_error() (luigi.contrib.beam_dataflow.BeanDataflowJobTask method), 81
clear() (luigi.mock.MockFileSystem method), 177
clear() (luigi.scheduler.Failsures method), 200
clear_instance_cache() (luigi.task_register.Register class method), 216
client (luigi.contrib.hdfs.config.hdfs attribute), 68
client (luigi.contrib.hdfs.webhdfs_client.WebHdfsClient attribute), 75
client_type (luigi.contrib.hdfs.webhdfs_client.webhdfs attribute), 75
client_version (luigi.contrib.hdfs.config.hdfs attribute), 68
clone() (luigi.Task method), 226
clone() (luigi.task.Task method), 212
close() (luigi.contrib.dropbox.ReadableDropboxFile method), 92
close() (luigi.contrib.hdfs.format.HdfsAtomicWriteDirPipe method), 69
close() (luigi.contrib.hdfs.format.HdfsAtomicWritePipe method), 69
close() (luigi.contrib.s3.ReadableS3File method), 139
close() (luigi.contrib.ssh.AtomicRemoteFileWriter method), 155
close() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 156

Index
CREATE_IF_NEEDED (luigi.contrib.bigquery.CreateDisposition attribute), 81
create_index() (luigi.contrib.esindex.CopyToIndex method), 98
create_marker_index() (luigi.contrib.esindex.ElasticsearchTarget method), 96
create_marker_table() (luigi.contrib.mssqldb.MSSqlTarget method), 118
create_marker_table() (luigi.contrib.mysqldb.MySqlTarget method), 118
create_marker_table() (luigi.contrib.postgres.PostgresTarget method), 126
create_marker_table() (luigi.contrib.sqla.SQLAlchemyTarget method), 153
CREATE_NEVER (luigi.contrib.bigquery.CreateDisposition attribute), 81
create_operation_job() (luigi.contrib.salesforce.SalesforceAPI method), 142
create_packages_archive() (in module luigi.contrib.hadoop), 104
create_schema() (luigi.contrib.redshift.S3CopyToTableDataprocBaseTask (class in luigi.contrib.dataproc), 88
create_subprocess() (luigi.format.InputPipeProcessWrapper method), 170
create_table() (luigi.contrib.rdbms.CopyToTable method), 130
create_table() (luigi.contrib.redshift.S3CopyToTable method), 134
create_table() (luigi.contrib.sqla.CopyToTable method), 153
CreateDataprocClusterTask (class in luigi.contrib.dataproc), 89
CreateDisposition (class in luigi.contrib.bigquery), 81
CSV (luigi.contrib.bigquery.DestinationFormat attribute), 82
CSV (luigi.contrib.bigquery.SourceFormat attribute), 81
CURRENT_SOURCE_VERSION (luigi.db_task_history.DbTaskHistory attribute), 167
Custom (class in luigi.date_interval), 166
D
daemonize() (in module luigi.process), 196
DaemonLogging (class in luigi.setup_logging), 206
data (luigi.configuration.LuigiTomlParser attribute), 66
data_dir (luigi.contrib.pai.PaiTask attribute), 123
data_interchange_format (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 106
database (luigi.contrib.hive.ExternalHiveTask attribute), 111
database (luigi.contrib.rdbms.CopyToTable attribute), 130
database (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 136
dataDir (luigi.contrib.pai.PaiJob attribute), 122
datadog (class in luigi.contrib.datadog_metric), 88
datadog (luigi.metrics.MetricsCollectors attribute), 175
DatadogMetricsCollector (class in luigi.contrib.datadog_metric), 88
dataflow_executable() (luigi.contrib.beam_dataflow.BeamDataflowJobTask method), 80
dataflow_params (luigi.contrib.beam_dataflow.BeamDataflowJobTask method), 80
DataflowParamKeys (class in luigi.contrib.beam_dataflow), 78
DataprocBaseTask (class in luigi.contrib.dataproc), 88
DataprocPysparkTask (class in luigi.contrib.dataproc), 89
DataprocSparkTask (class in luigi.contrib.dataproc), 89
dataset (luigi.contrib.bigquery.BQTable attribute), 82
dataset_exists() (luigi.contrib.bigquery.BigQueryClient method), 83
dataset_id (luigi.contrib.bigquery.BQDataset attribute), 82
DATASTORE_BACKUP (luigi.contrib.bigquery.SourceFormat attribute), 81
Date (class in luigi.date_interval), 165
date (luigi.contrib.external_daily_snapshot.ExternalDailySnapshot attribute), 99
date_format (luigi.DateHourParameter attribute), 234
date_format (luigi.DateMinuteParameter attribute), 235
date_format (luigi.DateSecondParameter attribute), 233
date_format (luigi.MonthParameter attribute), 233
date_format (luigi.parameter.DateHourParameter attribute), 184
date_format (luigi.parameter.DateMinuteParameter attribute), 184
date_format (luigi.parameter.DateParameter attribute), 183
date_format (luigi.parameter.DateSecondParameter attribute), 184
date_format (luigi.parameter.MonthParameter attribute), 183
date_format (luigi.parameter.YearParameter attribute), 184
date_format (luigi.YearParameter attribute), 234
dateHourParameter (class in luigi), 234
dateHourParameter (class in luigi.parameter), 184
dateInterval (class in luigi.date_interval), 165
dateIntervalParameter (class in luigi), 235
dateIntervalParameter (class in luigi.parameter), 184
datetime_to_parameter () (luigi.tools.range.RangeBase method), 159
datetime_to_parameter () (luigi.tools.range.RangeByMinutesBase method), 161
datetime_to_parameter () (luigi.tools.range.RangeDailyBase method), 160
datetime_to_parameter () (luigi.tools.range.RangeHourlyBase method), 160
datetime_to_parameter () (luigi.tools.range.RangeMonthly method), 162
datetime_to_parameters () (luigi.tools.range.RangeBase method), 159
datetime_to_parameters () (luigi.tools.range.RangeByMinutesBase method), 161
datetime_to_parameters () (luigi.tools.range.RangeDailyBase method), 160
datetime_to_parameters () (luigi.tools.range.RangeHourlyBase method), 160
datetime_to_parameters () (luigi.tools.range.RangeMonthly method), 162
days_back (luigi.tools.range.RangeDailyBase attribute), 160
days_forward (luigi.tools.range.RangeDailyBase attribute), 160
DbTaskHistory (class in luigi.db_task_history), 167
decrease_running_resources () (luigi.worker.TaskStatusReporter method), 222
decrease_running_task_resources () (luigi.RemoteScheduler method), 230
decrease_running_task_resources () (luigi.rpc.RemoteScheduler method), 197
decrease_running_task_resources () (luigi.scheduler.Scheduler method), 203
default (luigi.metrics.MetricsCollectors attribute), 175
DEFAULT_DB_PORT (luigi.contrib.postgres.PostgresTarget attribute), 126
DEFAULT_DB_PORT (luigi.contrib.redshift.RedshiftTarget attribute), 132
DEFAULT_PART_SIZE (luigi.contrib.s3.S3Client attribute), 137
default_tags (luigi.contrib.datadog_metric.datadog attribute), 88
default_tags (luigi.contrib.datadog_metric.DatadogMetricsCollector attribute), 88
DEFAULT_THREADS (luigi.contrib.s3.S3Client attribute), 137
DefaultHadoopJobRunner (class in luigi.contrib.hadoop), 105
DELAY (luigi.tools.range.RangeEvent attribute), 158
delegates () (in module luigi.util), 220
delete_dataset () (luigi.contrib.bigquery.BigQueryClient method), 83
delete_index () (luigi.contrib.esindex.CopyToIndex method), 98
delete_on_success (luigi.contrib.kubernetes.KubernetesJobTask attribute), 113
delete_table () (luigi.contrib.bigquery.BigQueryClient method), 83
DeleteDataprocClusterTask (class in luigi.contrib.dataproc), 89
dep_graph () (luigi.RemoteScheduler method), 230
dep_graph () (luigi.rpc.RemoteScheduler method), 197
dep_graph () (luigi.scheduler.Scheduler method), 202
DEPENDENCY_DISCOVERED (luigi.Event attribute), 244
DEPENDENCY_DISCOVERED (luigi.event.Event attribute), 168
DEPENDENCY_MISSING (luigi.Event attribute), 244
DEPENDENCY_MISSING (luigi.event.Event attribute), 168
DEPENDENCY_PRESENT (luigi.Event attribute), 244
DEPENDENCY_PRESENT (luigi.event.Event attribute), 168
deploy_mode (luigi.contrib.spark.SparkSubmitTask attribute), 149
deprecated_date_format
(luigi.DateMinuteParameter attribute), 235
deprecated_date_format
(luigi.parameter.DateMinuteParameter attribute), 184
DeprecatedBotoClientException, 137
deps () (luigi.contrib.hadoop.BaseHadoopJobTask method), 106
deps () (luigi.Task method), 227
deps () (luigi.task.Task method), 213
DequeueQueue (class in luigi.worker), 222
dereference () (in module luigi.contrib.hadoop), 104
destination_format
(luigi.contrib.bigquery.BigQueryExtractTask attribute), 87
destination_url (luigi.contrib.bigquery.BigQueryExtractTask attribute), 86
DestinationFormat (class in luigi.contrib.bigquery), 82
dfs_paths () (in module luigi.tools.deps), 157
DictParameter (class in luigi), 243
DictParameter (class in luigi.parameter), 191
disable_hard_timeout
(luigi.scheduler.RetryPolicy attribute), 199
disable_hard_timeout
(luigi.scheduler.scheduler attribute), 199
disable_hard_timeout (luigi.Task attribute), 225
disable_hard_timeout (luigi.Task attribute), 211
disable_instance_cache ()
(luigi.task_register.Register class method), 216
disable_persist
(luigi.scheduler.scheduler attribute), 199
disable_window
(luigi.scheduler.RetryPolicy attribute), 199
disable_window
(luigi.scheduler.scheduler attribute), 199
disable_window_seconds
(luigi.Task attribute), 225
disable_window_seconds
(luigi.task.Task attribute), 211
disable_worker ()
(luigi.RemoteScheduler method), 230
disable_worker ()
(luigi.rpc.RemoteScheduler method), 197
disable_worker ()
(luigi.scheduler.Scheduler method), 202
disable_workers ()
(luigi.scheduler.SimpleTaskState method), 201
disk_size_gb
(luigi.contrib.beam_dataflow.DataflowParamKeys attribute), 79
disk_size_gb
(luigi.contrib.beam_dataflowBeamDataflowJobTask attribute), 80
dispatch_scheduler_message ()
(luigi.worker.Worker method), 224
do_prune ()
(luigi.contrib.redshift.S3CopyToTable method), 133
do_truncate_table
(luigi.contrib.redshift.S3CopyToTable attribute), 133
do_work_on_compute_node () (in module luigi.contrib.lsf_runner), 115
doc_type
(luigi.contrib.esindex.CopyToIndex attribute), 97
DockerTask (class in luigi.contrib.docker_runner), 90
docs ()
(luigi.contrib.esindex.CopyToIndex method), 98
does_schema_exist ()
(luigi.contrib.redshift.S3CopyToTable method), 134
does_table_exist ()
(luigi.contrib.redshift.S3CopyToTable method), 134
done ()
(luigi.contrib.simulate.RunAnywayTarget method), 148
dont_remove_tmp_dir
(luigi.contrib.sge.SGEJobTask attribute), 147
download ()
(luigi.contrib.gcs.GCSClient method), 103
download ()
(luigi.contrib.hdfs.webhdfs_client.WebHdfsClient method), 76
download_as_bytes ()
(luigi.contrib.dropbox.DropboxClient method), 92
driver_class_path
(luigi.contrib.spark.SparkSubmitTask attribute), 149
driver_cores
(luigi.contrib.spark.SparkSubmitTask attribute), 149
driver_java_options
(luigi.contrib.spark.SparkSubmitTask attribute), 149
driver_library_path
(luigi.contrib.spark.SparkSubmitTask attribute), 149
driver_memory
(luigi.contrib.spark.SparkSubmitTask attribute), 149
DropboxClient (class in luigi.contrib.dropbox), 91
field_delimiter (luigi.contrib.bigquery.BigQueryLoadTask attribute), 85
FieldDelimiter (class in luigi.contrib.bigquery), 81
field_pattern() (luigi.contrib.beam_dataflow.BeanDataflowJobTask attribute), 80
FileAlreadyExists, 206
FileNotFoundException, 137
files (luigi.contrib.spark.PySparkTask attribute), 150
files (luigi.contrib.spark.SparkSubmitTask attribute), 149
FileSystem (class in luigi.target), 206
FileSystemTarget, 206
FileSystemException, 206
FileSystemTarget (class in luigi.target), 208
FileWrapper (class in luigi.format), 170
filter_kwargs (luigi.contrib.opener.Opener attribute), 120
filter_kwargs (luigi.contrib.opener.S3Opener attribute), 121
final_combiner (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 106
final_mapper (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 106
final_reducer (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 106
find_all_by_name() (luigi.db_task_history.DbTaskHistory method), 167
find_all_by_parameters() (luigi.db_task_history.DbTaskHistory method), 167
find_all_events() (luigi.db_task_history.DbTaskHistory method), 167
find_all_runs() (luigi.db_task_history.DbTaskHistory method), 167
find_deps() (in module luigi.tools.deps), 157
find_deps_cli() (in module luigi.tools.deps), 157
find_latest_runs() (luigi.db_task_history.DbTaskHistory method), 167
find_task_by_id() (luigi.db_task_history.DbTaskHistory method), 167
finish() (luigi.contrib.hadoop.HadoopJobRunner method), 105
finite_datetimes() (luigi.tools.range.RangeBase method), 159
finite_datetimes() (luigi.tools.range.RangeByMinutesBase method), 161
finite_datetimes() (luigi.tools.range.RangeDailyBase method), 160
finite_datetimes()
get_batch_running_tasks() (luigi.scheduler.SimpleTaskState method), 201
get_batcher() (luigi.scheduler.SimpleTaskState method), 201
get_bite() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 72
get_build_dir() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_collection() (luigi.contrib.mongodb.MongoTarget method), 115
get_config() (in module luigi.configuration), 65
get_config() (in module luigi.configuration.core), 65
get_configured_hadoop_version() (in module luigi.contrib.hdfs.config), 68
get_configured_hdfs_client() (in module luigi.contrib.hdfs.config), 68
get_data() (luigi.mock.MockFileSystem method), 176
get_dataproc_client() (in module luigi.contrib.dataproc), 88
get_default_client() (in module luigi.contrib.hive), 110
get_default_format() (in module luigi.format), 172
get_empty_ids() (luigi.contrib.mongodb.MongoRangeTarget method), 116
get_environment() (luigi.contrib.spark.SparkSubmitTask method), 149
get_extra_files() (in module luigi.contrib.hadoop), 104
get_hive_syntax() (in module luigi.contrib.hive), 109
get_hive_warehouse_location() (in module luigi.contrib.hive), 109
get_ignored_file_masks() (in module luigi.contrib.hive), 109
get_index() (luigi.contrib.mongodb.MongoTarget method), 115
get_info() (in module luigi.lock), 175
get_instance() (luigi.cmdline_parser.CmdlineParser class method), 164
get_job_class() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_job_details() (luigi.contrib.salesforce.SalesforceAPI method), 142
get_job_id_from_name() (luigi.contrib.batch.BatchClient method), 77
get_job_status() (luigi.contrib.batch.BatchClient method), 77
get_key() (luigi.contrib.s3.S3Client method), 137
get_libjars() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_log_format() (in module luigi.process), 196
get_logs() (luigi.contrib.batch.BatchClient method), 77
get_opener() (luigi.contrib.opener.OpenerRegistry method), 119
get_params() (luigi.Task class method), 226
get_param_names() (luigi.Task.Task class method), 212
get_param_values() (luigi.Task.Task class method), 212
get_provided_jars() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_running_task_resources() (luigi.contrib.hdfs.config method), 68
get_running_task_resources() (luigi.contrib.hdfs.config method), 68
get_scala_jars() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_scalding_core() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_scalding_jars() (luigi.contrib.scalding.ScaldingJobRunner method), 144
get_scheduler_message_response() (luigi.RemoteScheduler method), 230
get_scheduler_message_response() (luigi.rpc.RemoteScheduler method), 197
get_scheduler_message_response() (luigi.scheduler.Scheduler method), 202
get_soql_fields() (in module luigi.contrib.salesforce), 141
get_spool_handler() (in module luigi.process), 196
get_state() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 73
get_target() (luigi.contrib.opener.LocalOpener class method), 120
get_target() (luigi.contrib.opener.MockOpener method), 201
get_template_path() (luigi.contrib.opener.Openner class method), 120
get_target() (luigi.contrib.opener.S3Openner class method), 120
get_target() (luigi.contrib.opener.Openner class method), 121
get_target_path() (luigi.contrib.beam_dataflowBeamDataflowJobTask static method), 81
get_task() (luigi.scheduler.SimpleTaskState method), 201
get_task_cls() (luigi.task_register.Register class method), 216
get_task_family() (luigi.Task class method), 225
get_task_family() (luigi.task.Task class method), 212
get_task_namespace() (luigi.Task class method), 225
get_task_namespace() (luigi.task.Task class method), 211
get_task_obj() (luigi.cmdline_parser.CmdlineParser method), 164
get_task_output_description() (in module luigi.tools.deps), 157
get_task_progress_percentage() (luigi.RemoteScheduler method), 230
get_task_progress_percentage() (luigi.rpc.RemoteScheduler method), 198
get_task_progress_percentage() (luigi.task.Task class method), 203
get_task_requires() (luigi.task.Task class method), 200
get_task_status_message() (luigi.RemoteScheduler method), 230
get_task_status_message() (luigi.rpc.RemoteScheduler method), 198
get_task_status_message() (luigi.task.Task class method), 203
get_tasks() (luigi.task.Task class method), 200
get_template_path() (luigi.server.BaseTaskHistoryHandler method), 204
get_temp_job_jar() (luigi.contrib.scalding.ScaldingJobRunner class method), 144
get_view() (luigi.contrib.bigquery.BigQueryClient method), 83
get_work() (luigi.RemoteScheduler method), 230
get_work() (luigi.rpc.RemoteScheduler method), 198
get_work() (luigi.task.Task class method), 202
get_worker() (luigi.task.Task class method), 201
get_worker_ids() (luigi.task.Task class method), 201
get_wrapped() (luigi.freezing.FrozenOrderedDict method), 172
getboolean() (luigi.configuration.cfg_parser.LuigiConfigParser method), 64
getboolean() (luigi.configuration.LuigiConfigParser method), 66
getboolean() (luigi.configuration.toml_parser.LuigiTomlParser method), 66
getboolean() (luigi.configuration.toml_parser.LuigiTomlParser method), 65
getbool() (luigi.configuration.cfg_parser.LuigiConfigParser method), 64
getbool() (luigi.configuration.LuigiConfigParser method), 66
getbool() (luigi.configuration.toml_parser.LuigiTomlParser method), 66
getbool() (luigi.configuration.toml_parser.LuigiTomlParser method), 65
getbool() (luigi.configuration.cfg_parser.LuigiConfigParser method), 64
getbool() (luigi.configuration.LuigiConfigParser method), 66
getint() (luigi.configuration.cfg_parser.LuigiConfigParser method), 66
getint() (luigi.configuration.LuigiConfigParser method), 66
getint() (luigi.configuration.toml_parser.LuigiTomlParser method), 66
getint() (luigi.configuration.toml_parser.LuigiTomlParser method), 65
getintdict() (luigi.configuration.cfg_parser.LuigiConfigParser method), 65
getintdict() (luigi.configuration.LuigiConfigParser method), 66
getintdict() (luigi.configuration.toml_parser.LuigiTomlParser method), 66
getintdict() (luigi.configuration.toml_parser.LuigiTomlParser method), 65
getintdict() (luigi.contrib.hdfs.target.HdfsTarget method), 74
global_instance() (luigi.cmdline_parser.CmdlineParser class method), 164
gpu_type (luigi.contrib.pai.PaiTask attribute), 124
gpuNumber (luigi.contrib.pai.TaskRole attribute), 123
gpuType (luigi.contrib.pai.PaiJob attribute), 122
graph() (luigi.RemoteScheduler method), 230
graph() (luigi.rpc.RemoteScheduler method), 198
graph() (luigi.task.Task class method), 202
graph_url (luigi.tools.luigi_grep.LuigiGrep attribute), 158
group() (luigi.contrib.hadoop.LocalJobRunner method), 106
group_by_error_messages
input (luigi.format.GzipFormat attribute), 171
input (luigi.format.NewlineFormat attribute), 171
input (luigi.format.TextFormat attribute), 171
input() (luigi.Task method), 227
input() (luigi.task.Task method), 213
input_hadoop() (luigi.contrib.hadoop.BaseHadoopJobTask method), 106
input_local() (luigi.contrib.hadoop.BaseHadoopJobTask method), 106
InputPipeProcessWrapper (class in luigi.format), 170
instance() (luigi.configuration.base_parser.BaseParser class method), 63
INTERACTIVE (luigi.contrib.bigquery.QueryMode attribute), 81
InterfaceLogging (class in luigi.setup_logging), 206
internal_reader() (luigi.contrib.hadoop.JobTask method), 108
internal_writer() (luigi.contrib.hadoop.JobTask method), 108
InterpolationMissingEnvvarError, 64
IntParameter (class in luigi), 237
IntParameter (class in luigi.parameter), 184
InvalidDeleteException, 102, 137
InvalidQuery, 119
inverse_dep_graph() (luigi.RemoteScheduler method), 230
inverse_dep_graph() (luigi.rpc.RemoteScheduler method), 198
inverse_dep_graph() (luigi.scheduler.Scheduler method), 203
is_batchable() (luigi.scheduler.Task method), 200
is_dir() (luigi.contrib.s3.S3Client method), 138
is_pause_enabled() (luigi.RemoteScheduler method), 230
is_pause_enabled() (luigi.rpc.RemoteScheduler method), 198
is_pause_enabled() (luigi.scheduler.Scheduler method), 202
is_paused() (luigi.RemoteScheduler method), 230
is_paused() (luigi.rpc.RemoteScheduler method), 198
is_paused() (luigi.scheduler.Scheduler method), 202
is_sql_file (luigi.contrib.salesforce.Query salesforce attribute), 141
is_trivial_worker() (luigi.scheduler.Worker method), 200
is_writable() (luigi.contrib.hdfs.target.HdfsTarget method), 75
isdir() (luigi.contrib.dropbox.DropboxClient method), 91
isdir() (luigi.contrib.gcs.GCSClient method), 102
isdir() (luigi.contrib.s3.S3Client method), 138
isdir() (luigi.contrib.ssh.RemoteFileSystem method), 155
isdir() (luigi.local_target.LocalFileSystem method), 174
isdir() (luigi.mock.MockFileSystem method), 176
J
jar() (luigi.contrib.hadoop.jar.HadoopJarJobTask method), 108
jar() (luigi.contrib.scalding.ScaldingJobTask method), 145
jars (luigi.contrib.dataproc.DataprocSparkTask attribute), 89
jars (luigi.contrib.spark.sparkSubmitTask attribute), 149
job_args (luigi.contrib.dataproc.DataprocPySparkTask attribute), 89
job_args (luigi.contrib.dataproc.DataprocSparkTask attribute), 89
job_args() (luigi.contrib.scalding.ScaldingJobTask method), 145
job_class() (luigi.contrib.scalding.ScaldingJobTask method), 145
job_definition (luigi.contrib.batch.BatchTask attribute), 78
job_file (luigi.contrib.dataproc.DataprocPySparkTask attribute), 89
job_name (luigi.contrib.batch.BatchTask attribute), 78
job_name (luigi.contrib.beam_dataflow BeamDataflowJobTask attribute), 80
job_name (luigi.contrib.beam_dataflow.DataflowParamKeys attribute), 79
job_name (luigi.contrib.sge.SGEJobTask attribute), 147
job_name_flag (luigi.contrib.lsf.LSFJobTask attribute), 114
job_name_format (luigi.contrib.sge.SGEJobTask attribute), 147
job_queue (luigi.contrib.batch.BatchTask attribute), 78
job_runner() (luigi.contrib.hadoop.BaseHadoopJobTask method), 106
job_runner() (luigi.contrib.hadoop.JobTask method), 107
job_runner() (luigi.contrib.hadoop.jar.HadoopJarJobTask method), 108
job_runner() (luigi.contrib.hive.HiveQueryTask method), 111
job_runner() (luigi.contrib.scalding.ScaldingJobTask method), 145

Index 267
logger (in module luigi.contrib.scailing), 143
logging_conf_file (luigi.interface.core attribute), 173
LSFJobTask (class in luigi.contrib.lsf), 114
luigi (module), 224
luigi.batch_notifier (module), 163
luigi.cmdline (module), 164
luigi.cmdline_parser (module), 164
luigi.configuration (module), 65
luigi.configuration.base_parser (module), 63
luigi.configuration.cfg_parser (module), 64
luigi.configuration.core (module), 65
luigi.configuration.toml_parser (module), 65
luigi.contrib (module), 157
luigi.contrib.batch (module), 77
luigi.contrib.beam_dataflow (module), 78
luigi.contrib.bigquery (module), 81
luigi.contrib.bigquery_aven (module), 87
luigi.contrib.datadog_metric (module), 88
luigi.contrib.dataproc (module), 88
luigi.contrib.docker_runner (module), 90
luigi.contrib.dropbox (module), 91
luigi.contrib.ecs (module), 93
luigi.contrib.esindex (module), 95
luigi.contrib.external_daily_snapshot (module), 98
luigi.contrib.external_program (module), 99
luigi.contrib.ftp (module), 100
luigi.contrib.gcp (module), 101
luigi.contrib.gcs (module), 102
luigi.contrib.hadoop (module), 104
luigi.contrib.hadoop_jar (module), 108
luigi.contrib.hdfs (module), 76
luigi.contrib.hdfs.abstract_client (module), 66
luigi.contrib.hdfs.clients (module), 67
luigi.contrib.hdfs.config (module), 68
luigi.contrib.hdfs.error (module), 69
luigi.contrib.hdfs.format (module), 69
luigi.contrib.hdfs.hadoopcli_clients (module), 70
luigi.contrib.hdfs.sanbebite_client (module), 71
luigi.contrib.hdfs.target (module), 74
luigi.contrib.hdfs.webhdfs_client (module), 75
luigi.contrib.hive (module), 109
luigi.contrib.kubernetes (module), 112
luigi.contrib.lsf (module), 114
luigi.contrib.lsf_runner (module), 115
luigi.contrib.mongodb (module), 115
luigi.contrib.mr_runner (module), 117
luigi.contrib.msgsql (module), 117
luigi.contrib.mssql (module), 118
luigi.contrib.opener (module), 119
luigi.contrib.pai (module), 121
luigi.contrib.pig (module), 124
luigi.contrib.postgres (module), 125
luigi.contrib.postgres (module), 127
luigi.contrib.pyspark_runner (module), 129
luigi.contrib.rdbms (module), 129
luigi.contrib.redis_store (module), 132
luigi.contrib.redshift (module), 132
luigi.contrib.s3 (module), 137
luigi.contrib.salesforce (module), 141
luigi.contrib.scaling (module), 143
luigi.contrib.sge (module), 145
luigi.contrib.sge_runner (module), 147
luigi.contrib.simulate (module), 148
luigi.contrib.spark (module), 148
luigi.contrib.sparkkey (module), 150
luigi.contrib.sqla (module), 150
luigi.contrib.ssh (module), 154
luigi.contrib.target (module), 156
luigi.contrib.webhdfs (module), 156
luigi.date_interval (module), 164
luigi.db_task_history (module), 167
luigi.event (module), 168
luigi.execution_summary (module), 169
luigi.format (module), 170
luigi.freezing (module), 172
luigi.interface (module), 172
luigi.local_target (module), 173
luigi.lock (module), 175
luigi.metrics (module), 175
luigi.mock (module), 176
luigi.notifications (module), 177
luigi.parameter (module), 179
luigi.process (module), 196
luigi.retcodes (module), 196
luigi.rpc (module), 197
luigi.scheduler (module), 198
luigi.server (module), 203
luigi.setup_logging (module), 206
luigi.target (module), 206
luigi.task (module), 209
luigi.task_history (module), 215
luigi.task_register (module), 216
luigi.task_status (module), 217
luigi.tools (module), 163
luigi.tools.deps (module), 157
luigi.tools.deps_tree (module), 157
luigi.tools.luigi_grep (module), 158
luigi.tools.range (module), 158
luigi.util (module), 217
luigi.worker (module), 221
luigi_run() (in module luigi.cmdline), 164
LuigiConfigParser (class in luigi.configuration), 65
LuigiConfigParser (class in luigi.configuration.cfg_parser), 64
luigid() (in module luigi.cmdline), 164
LuigiGrep (class in luigi.tools.luigi_grep), 158
LuigiRunResult (class in luigi.execution_summary), 169
LuigiStatusCode (class in luigi), 247
LuigiStatusCode (class in luigi.execution_summary), 169
LuigiTomlParser (class in luigi.configuration), 66
LuigiTomlParser (class in luigi.configuration.toml_parser), 65

M
main() (in module luigi.contrib.lsf_runner), 115
main() (in module luigi.contrib.mrrunner), 117
main() (in module luigi.contrib.sge_runner), 147
main() (in module luigi.tools.deps), 157
main() (in module luigi.tools.deps_tree), 158
main() (in module luigi.tools.luigi_grep), 158
main() (luigi.contrib.hadoopJar.HadoopJarJobTask method), 108
main() (luigi.contrib.spark.PySparkTask method), 150
main_class (luigi.contrib.dataproc.DataProcSparkTask attribute), 89
make_dataset() (luigi.contrib.bigquery.BigQueryClient method), 83
makedirs() (luigi.LocalTarget.LocalTarget method), 175
makedirs() (luigi.LocalTarget method), 229
map_column() (luigi.contrib.postgres.CopyToTable method), 127
mapper() (luigi.contrib.hadoop.JobTask method), 107
mapping (luigi.contrib.esindex.CopyToIndex attribute), 97
mark_as_done() (luigi.RemoteScheduler method), 230
mark_as_done() (luigi.rpc.RemoteScheduler method), 198
mark_as_done() (luigi.schedular.Scheduler method), 202
marker_doc_type (luigi.contrib.esindex.ElasticsearchTarget attribute), 96
marker_index (luigi.contrib.esindex.ElasticsearchTarget attribute), 96
marker_index_document_id() (luigi.contrib.esindex.ElasticsearchTarget method), 96
marker_index_hist_size (luigi.contrib.esindex.CopyToIndex attribute), 97
marker_key() (luigi.contrib.rediss.RedisTarget method), 132
marker_prefix (luigi.contrib.rediss.RedisTarget attribute), 132
marker_table (luigi.contrib.mssqldb.MSsqlTarget attribute), 117
marker_table (luigi.contrib.mysqldb.MySqlTarget attribute), 118
marker_table (luigi.contrib.postgres.PostgresTarget attribute), 126
marker_table (luigi.contrib.rediss.RedisTarget attribute), 132
marker_table (luigi.contrib.sqla.SQLAlchemyTarget attribute), 152
master (luigi.contrib.spark.SparkSubmitTask attribute), 148
master_disk_size (luigi.contrib.dataproc.CreateDataprocClusterTask attribute), 89
master_node_type (luigi.contrib.dataproc.CreateDataprocClusterTask attribute), 89
max_age (luigi.server.cors attribute), 203
max_bad_records (luigi.contrib.bigquery.BigQueryLoadTask attribute), 85
max_batch_size (luigi.Task attribute), 225
max_batch_size (luigi.task.Task attribute), 211
max_graph_nodes (luigi.schedular.schedular attribute), 199
max_keep_alive_idle_duration (luigi.worker.worker attribute), 223
max_num_workers (luigi.contrib.beam_dataflow.DataflowParamKeys attribute), 79
max_num_workers (luigi.contrib.beam_dataflow.DataflowJobTask attribute), 80
max_reschedules (luigi.worker.worker attribute), 223
max_retrials (luigi.contrib.kubernetes.kubernetes attribute), 112
max_retrials (luigi.contrib.kubernetes.KubernetesJobTask attribute), 113
max_shown_tasks (luigi.schedular.schedular attribute), 199
may_prune() (luigi.schedular.SimpleTaskState method), 201
memoryMB (luigi.contrib.pai.TaskRole attribute), 123
merge_batch_results() (luigi.contrib.salesforce.QuerySalesforce method), 142
MetastoreClient (class in luigi.contrib.hive), 110
method (luigi.notifications.email attribute), 178
Index 271

luigi.contrib.datadog_metric.datadog

luigi.contrib.hdfs.webhdfs_client.WebHdfsClient method, 76

luigi.contrib.s3.S3Client method, 138

luigi.contrib.ssh.RemoteFileSystem method, 155

luigi.local_target.LocalFileSystem method, 174

luigi.target.FileSystem method, 207

MockFilesystem (class in luigi.mock), 176

MockOpener (class in luigi.contrib.opener), 120

MockTarget (class in luigi.mock), 177

module (luigi.interface.core attribute), 173

MongoCellTarget (class in luigi.contrib.mongodbc), 115

MongoCountTarget (class in luigi.contrib.mongodbc), 116

MongoRangeTarget (class in luigi.contrib.mongodbc), 116

Month (class in luigi.date_interval), 166

MonthParameter (class in luigi), 233

MonthParameter (class in luigi.parameter), 183

months_back (luigi.tools.range.RangeMonthly attribute), 162

months_forward (luigi.tools.range.RangeMonthly attribute), 162

most_common() (in module luigi.tools.range), 161

mount_tmp (luigi.contrib.docker_runner.DockerTask attribute), 90

move() (luigi.contrib.dropbox.DropboxClient method), 92

move() (luigi.contrib.gcs.GCSClient method), 103

move() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 70

move() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 72

move() (luigi.contrib.hdfs.target.HdfsTarget method), 74

move() (luigi.contrib.lmdb.LmdbTarget method), 175

move() (luigi.LocalTarget method), 229

move() (luigi.mock.MockFileSystem method), 176

move() (luigi.mock.MockTarget method), 177

move() (luigi.target.FileSystem method), 207

move_dir() (luigi.contrib.hdfs.target.HdfsTarget method), 74

move_dir() (luigi.local_target.LocalTarget method), 174

MixedUnicodeBytesFormat (class in luigi.format), 171

MixedUnicodeBytesWrapper (class in luigi.format), 170

MixinBigQueryBulkComplete (class in luigi.contrib.bigquery), 84

MixinBigqueryBulkComplete (in module luigi.contrib.bigquery), 87

MixinNaiveBulkComplete (class in luigi.task), 214

mkdir() (in module luigi.contrib.hdfs.clients), 68

mkdir() (luigi.contrib.dropbox.DropboxClient method), 91

mkdir() (luigi.contrib.gcs.GCSClient method), 103

mkdir() (luigi.contrib.hdfs.abstract_client.HdfsFileSystem method), 67

mkdir() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 70

mkdir() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 71

mkdir() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 73

mkdir() (luigi.contrib.s3.S3Client method), 138

mkdir() (luigi.contrib.ssh.RemoteFileSystem method), 155

mkdir() (luigi.local_target.LocalFileSystem method), 174

mkdir() (luigi.mock.MockFileSystem method), 177

MockFilesystem (class in luigi.mock), 176

MockOpener (class in luigi.contrib.opener), 120

MockTarget (class in luigi.mock), 177

module (luigi.interface.core attribute), 173

MongoCountTarget (class in luigi.contrib.mongodbc), 116

MongoRangeTarget (class in luigi.contrib.mongodbc), 116

MongoTarget (class in luigi.contrib.mongodbc), 115

Month (class in luigi.date_interval), 166

MonthParameter (class in luigi), 233

MonthParameter (class in luigi.parameter), 183

months_back (luigi.tools.range.RangeMonthly attribute), 162

months_forward (luigi.tools.range.RangeMonthly attribute), 162

most_common() (in module luigi.tools.range), 161

mount_tmp (luigi.contrib.docker_runner.DockerTask attribute), 90

move() (luigi.contrib.dropbox.DropboxClient method), 92

move() (luigi.contrib.gcs.GCSClient method), 103

move() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 70

move() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 72

move() (luigi.contrib.hdfs.target.HdfsTarget method), 74

move() (luigi.contrib.lmdb.LmdbTarget method), 175

move() (luigi.LocalTarget method), 229

move() (luigi.mock.MockFileSystem method), 176

move() (luigi.mock.MockTarget method), 177

move() (luigi.target.FileSystem method), 207

move_dir() (luigi.contrib.hdfs.target.HdfsTarget method), 74

move_dir() (luigi.local_target.LocalTarget method), 174

MissingParameterException, 179

MissingParentDirectory, 206

MissingUnicodeBytesFormat (class in luigi.format), 171

MissingUnicodeBytesWrapper (class in luigi.format), 170

MockTarget (class in luigi.mock), 177

module (luigi.interface.core attribute), 173

MongoCellTarget (class in luigi.contrib.mongodbc), 115

MongoCountTarget (class in luigi.contrib.mongodbc), 116

MongoRangeTarget (class in luigi.contrib.mongodbc), 116

MongoTarget (class in luigi.contrib.mongodbc), 115

Month (class in luigi.date_interval), 166

MonthParameter (class in luigi), 233

MonthParameter (class in luigi.parameter), 183

months_back (luigi.tools.range.RangeMonthly attribute), 162

months_forward (luigi.tools.range.RangeMonthly attribute), 162

most_common() (in module luigi.tools.range), 161

mount_tmp (luigi.contrib.docker_runner.DockerTask attribute), 90

move() (luigi.contrib.dropbox.DropboxClient method), 92

move() (luigi.contrib.gcs.GCSClient method), 103

move() (luigi.contrib.hdfs.hadoopcli_clients.HdfsClient method), 70

move() (luigi.contrib.hdfs.snakebite_client.SnakebiteHdfsClient method), 72

move() (luigi.contrib.hdfs.target.HdfsTarget method), 74

move() (luigi.contrib.lmdb.LmdbTarget method), 175

move() (luigi.LocalTarget method), 229

move() (luigi.mock.MockFileSystem method), 176

move() (luigi.mock.MockTarget method), 177

move() (luigi.target.FileSystem method), 207

move_dir() (luigi.contrib.hdfs.target.HdfsTarget method), 74

move_dir() (luigi.local_target.LocalTarget method), 174

luigi.contrib.datadog_metric.datadog

luigi.contrib.hdfs.webhdfs_client.WebHdfsClient method, 76

luigi.contrib.s3.S3Client method, 138

luigi.contrib.ssh.RemoteFileSystem method, 155

luigi.local_target.LocalFileSystem method, 174

luigi.target.FileSystem method, 207

luigi.mock.MockFilesystem method, 177

luigi.mock.MockTarget method, 177

luigi.target.FileSystem method, 207

luigi.contrib.hdfs.target.HdfsTarget method, 74

luigi.local_target.LocalTarget method, 174
move_dir() (luigi.LocalTarget method), 229
move_to_final_destination() (luigi.contrib.dropbox.AtomicWritableDropboxFile method), 92
move_to_final_destination() (luigi.contrib.s3.AtomicS3File method), 139
move_to_final_destination() (luigi.contrib.gcs.AtomicGCSFile method), 103
move_to_final_destination() (luigi.contrib.hdfs.LocalHdfsFile method), 174
move_to_final_destination() (luigi.target.AtomicLocalFile method), 159
move_to_final_destination() (luigi.target.AtomicLocalFile method), 209
moving_start() (luigi.tools.range.RangeBase method), 159
moving_start() (luigi.tools.range.RangeByMinutesBase method), 161
moving_start() (luigi.tools.range.RangeDailyBase method), 160
moving_start() (luigi.tools.range.RangeHourlyBase method), 161
moving_start() (luigi.tools.range.RangeMonthly method), 162
moving_stop() (luigi.tools.range.RangeBase method), 159
moving_stop() (luigi.tools.range.RangeByMinutesBase method), 161
moving_stop() (luigi.tools.range.RangeDailyBase method), 160
moving_stop() (luigi.tools.range.RangeHourlyBase method), 161
moving_stop() (luigi.tools.range.RangeMonthly method), 162
mr_priority (luigi.contrib.hadoop.BaseHadoopJobTask attribute), 106
MSSqlTarget (class in luigi.contrib.mssql), 117
MultiReplacer (class in luigi.contrib.postgresql), 125
MySqlTarget (class in luigi.contrib.mysql), 118
n_cpu (luigi.contrib.sge.SGEJobTask attribute), 147
n_cpu_flag (luigi.contrib.lsf.LSFJobTask attribute), 114
n_pending_last_scheduled (luigi.worker.GetWorkResponse attribute), 221
n_pending_tasks (luigi.worker.GetWorkResponse attribute), 221
n_reduce_tasks (luigi.contrib.hadoop.JobTask attribute), 107
n_reduce_tasks (luigi.contrib.hive.HiveQueryTask attribute), 110
n_unique_pending (luigi.worker.GetWorkResponse attribute), 221
name (luigi.contrib.docker_runner.DockerTask attribute), 90
name (luigi.contrib.kubernetes.KubernetesJobTask attribute), 113
name (luigi.contrib.pai.PaiTask attribute), 123
name (luigi.contrib.pai.TaskRole attribute), 123
name (luigi.contrib.spark.PySparkTask attribute), 149
name (luigi.contrib.spark.SparkSubmitTask attribute), 148
name (luigi.db_task_history.TaskParameter attribute), 167
name (luigi.db_task_history.TaskRecord attribute), 168
namenode_host (luigi.contrib.hadoop.config.hadoop attribute), 68
namenode_port (luigi.contrib.hadoop.config.hadoop attribute), 68
names (luigi.contrib.opener.LocalOpener attribute), 120
names (luigi.contrib.opener.MockOpener attribute), 120
names (luigi.contrib.opener.S3Opener attribute), 121
namespace() (in module luigi), 228
namespace() (in module luigi.task), 209
network (luigi.contrib.beam_dataflowBeamDataflowJobTask attribute), 80
network (luigi.contrib.beam_dataflow.DataflowParamKeys attribute), 79
network_mode (luigi.contrib.docker_runner.DockerTask attribute), 90
NEWLINE_DELIMITED_JSON (luigi.contrib.bigquery.DestinationFormat attribute), 82
NEWLINE_DELIMITED_JSON (luigi.contrib.bigquery.SourceFormat attribute), 81
NewlineFormat (class in luigi.format), 171
NewlineWrapper (class in luigi.format), 170
next() (luigi.date_interval.DateInterval method), 165
next_in_enumeration() (luigi.DateParameter method), 233
next_in_enumeration() (luigi.IntParameter method), 237
next_in_enumeration() (luigi.MonthParameter method), 233
next_in_enumeration() (luigi.Parameter method), 232
next_in_enumeration() (luigi.parameter.DateParameter method), 221
Index
Index 275

parameters (luigi.contrib.batch.BatchTask attribute), 78
parameters_to_datetime() (luigi.tools.range.RangeBase method), 160
parameters_to_datetime() (luigi.tools.range.RangeByMinutesBase method), 161
parameters_to_datetime() (luigi.tools.range.RangeDailyBase method), 160
parameters_to_datetime() (luigi.tools.range.RangeHourlyBase method), 161
parameters_to_datetime() (luigi.tools.range.RangeMonthly method), 162
ParameterVisibility (class in luigi.parameter), 179
parse() (luigi.BoolParameter method), 239
parse() (luigi.ChoiceParameter method), 246
parse() (luigi.date_interval.Custom class method), 167
parse() (luigi.date_interval.Date class method), 166
parse() (luigi.date_interval.DateInterval class method), 165
parse() (luigi.date_interval.Month class method), 166
parse() (luigi.date_interval.Week class method), 166
parse() (luigi.DateIntervalParameter method), 236
parse() (luigi.DictParameter method), 235
parse() (luigi.EnumParameter method), 242
parse() (luigi.FloatParameter method), 238
parse() (luigi.IntParameter method), 237
parse() (luigi.ListParameter method), 241
parse() (luigi.NumericalParameter method), 245
parse() (luigi.OptionalParameter method), 247
parse() (luigi.Parameter method), 232
parse() (luigi.parameter.BoolParameter method), 187
parse() (luigi.parameter.ChoiceParameter method), 196
parse() (luigi.parameter.DateMinuteParameter method), 184
parse() (luigi.parameter.DictParameter method), 192
parse() (luigi.parameter.EnumListParameter method), 190
parse() (luigi.parameter.EnumParameter method), 190
parse() (luigi.parameter.FloatParameter method), 186
parse() (luigi.parameter.IntParameter method), 185
parse() (luigi.parameter.ListParameter method), 193
parse() (luigi.parameter.NumericalParameter method), 195
parse() (luigi.parameter.OptionalParameter method), 182
parse() (luigi.parameter.Parameter method), 181
parse() (luigi.parameter.TaskParameter method), 189
parse() (luigi.parameter.TimeDeltaParameter method), 188
parse() (luigi.parameter.TupleParameter method), 194
parse() (luigi.TaskParameter method), 240
parse() (luigi.TimeDeltaParameter method), 236
parse() (luigi.TupleParameter method), 242
parse_results() (in module luigi.contrib.salesforce), 141
parsing (luigi.BoolParameter attribute), 239
parsing (luigi.parameter.BoolParameter attribute), 187
partition (luigi.contrib.hive.ExternalHiveTask attribute), 112
partition (luigi.contrib.presto.PrestoTask attribute), 128
partition_spec() (luigi.contrib.hive.HiveClient method), 109
partition_spec() (luigi.contrib.hive.HiveCommandClient method), 109
partition_spec() (luigi.contrib.hive.MetastoreClient method), 110
partition_spec() (luigi.contrib.hive.WarehouseHiveClient method), 110
password (luigi.contrib.pai.OpenPai attribute), 123
password (luigi.contrib.presto.presto attribute), 127
password (luigi.contrib.presto.PrestoTask attribute), 128
password (luigi.contrib.rdbms.CopyToTable attribute), 130
password (luigi.contrib.rdbms.Query attribute), 131
password (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 136
password (luigi.contrib.salesforce.KillOpenRedshiftSessions attribute), 136
password (luigi.contrib.salesforce.salesforce attribute), 141
password (luigi.notifications.smtp.salesforce attribute), 178
path (luigi.contrib.hive.HivePartitionTarget attribute), 111
path (luigi.contrib.s3.S3EmrTask attribute), 140

Luigi Documentation, Release 2.8.13
method), 103
put_string() (luigi.contrib.gcs.GCSClient method), 103
put_string() (luigi.contrib.s3.S3Client method), 137
py_files (luigi.contrib.spark.SparkSubmitTask attribute), 149
py_packages (luigi.contrib.spark.PySparkTask attribute), 150
pyspark_driver_python (luigi.contrib.spark.PySparkTask attribute), 148
pyspark_python (luigi.contrib.spark.PySparkSubmitTask attribute), 148
PySparkRunner (class in luigi.contrib.pyspark_runner), 129
PySparkSessionRunner (class in luigi.contrib.pyspark_runner), 129
PySparkTask (class in luigi.contrib.spark), 149

Q
queries (luigi.contrib.redshift.S3CopyToTable attribute), 134
Query (class in luigi.contrib.rdbms), 131
query (luigi.contrib.bigquery.BigQueryRunQueryTask attribute), 86
query (luigi.contrib.presto.PrestoTask attribute), 129
query (luigi.contrib.rdbms.Query attribute), 131
query() (luigi.contrib.hive.HiveQueryTask method), 110
query() (luigi.contrib.salesforce.SalesforceAPI method), 142
query_all() (luigi.contrib.salesforce.SalesforceAPI method), 142
query_mode (luigi.contrib.bigquery.BigQueryRunQueryTask attribute), 86
query_more() (luigi.contrib.salesforce.SalesforceAPI method), 142
QueryMode (class in luigi.contrib.bigquery), 81
QuerySalesforce (class in luigi.contrib.salesforce), 141
queue (luigi.contrib.spark.SparkSubmitTask attribute), 149
queue_flag (luigi.contrib.lsf.LSFJobTask attribute), 114

R
raise_in_complete (luigi.notifications.TestNotificationsTask attribute), 177
raise_on_error (luigi.contrib.esindex.CopyToIndex attribute), 97
raises (luigi.rpc.URLLibFetcher attribute), 197
RangeBase (class in luigi.tools.range), 197
RangeByMinutesBase (class in luigi.tools.range), 163
RangeDaily (class in luigi.tools.range), 161
RangeDailyBase (class in luigi.tools.range), 160
RangeEvent (class in luigi.tools.range), 158
RangeHourly (class in luigi.tools.range), 162
RangeHourlyBase (class in luigi.tools.range), 160
RangeMonthly (class in luigi.tools.range), 162
re_enable() (luigi.scheduler.SimpleTaskState method), 201
re_enable_task() (luigi.RemoteScheduler method), 230
re_enable_task() (luigi.rpc.RemoteScheduler method), 198
re_enable_task() (luigi.scheduler.Scheduler method), 203
read() (luigi.configuration.LuigiTomlParser method), 66
read() (luigi.configuration.toml_parser.LuigiTomlParser method), 65
read() (luigi.contrib.dropbox.ReadableDropboxFile method), 92
read() (luigi.contrib.hadoop.JobTask method), 92
read() (luigi.contrib.hdfs.webhdfs_client.WebHdfsClient method), 76
read() (luigi.contrib.mongodb.MongoCellTarget method), 116
read() (luigi.contrib.mongodb.MongoCollectionTarget method), 116
read() (luigi.contrib.mongodb.MongoCountTarget method), 117
read() (luigi.contrib.mongodb.MongoRangeTarget method), 116
read() (luigi.contrib.s3.ReadableS3File method), 139
read() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 156
read() (luigi.format.NewlineWrapper method), 170
readable() (luigi.contrib.dropbox.ReadableDropboxFile method), 92
readable() (luigi.contrib.s3.ReadableS3File method), 139
readable() (luigi.format.InputPipeProcessWrapper method), 170
readable() (luigi.format.OutputPipeProcessWrapper method), 170
ReadableDropboxFile (class in luigi.contrib.dropbox), 92
ReadableS3File (class in luigi.contrib.s3), 139
ReadableWebHdfsFile (class in luigi.contrib.webhdfs), 156
reader() (luigi.contrib.hadoop.JobTask method), 107
readlines() (luigi.contrib.webhdfs.ReadableWebHdfsFile method), 156
receiver (luigi.notifications.email attribute), 178
RecentRunHandler (class in luigi.server), 205
Index
Luigi Documentation, Release 2.8.13

run() (luigi.contrib.bigquery.BigQueryCreateViewTask method), 87
run() (luigi.contrib.bigquery.BigQueryLoadTask method), 85
run() (luigi.contrib.bigquery.BigQueryRunQueryTask method), 86
run() (luigi.contrib.bigquery_avro.BigQueryLoadAvro method), 88
run() (luigi.contrib.dataproc.CreateDataprocClusterTask method), 89
run() (luigi.contrib.dataproc.DataprocPysparkTask method), 89
run() (luigi.contrib.dataproc.DataprocSparkTask method), 89
run() (luigi.contrib.dataproc.DeleteDataprocClusterTask method), 90
run() (luigi.contrib.docker_runner.DockerTask method), 91
run() (luigi.contrib.esindex.CopyToIndex method), 98
run() (luigi.contrib.external_program.ExternalProgramTask method), 100
run() (luigi.contrib.hadoop.BaseHadoopJobTask method), 106
run() (luigi.contrib.kubernetes.KubernetesJobTask method), 114
run() (luigi.contrib.lsf.LocalLSFJobTask method), 115
run() (luigi.contrib.mrrunner.Runner method), 117
run() (luigi.contrib.pai.PaiJob method), 124
run() (luigi.contrib.pig.PigJob method), 125
run() (luigi.contrib.postgres.CopyToTable method), 127
run() (luigi.contrib.postgres.PostgresQuery method), 127
run() (luigi.contrib.postgres.PostgresQuery method), 129
run() (luigi.contrib.pyspark_runner.AbstractPySparkRunner method), 129
run() (luigi.contrib.redshift.KillOpenRedshiftSessions method), 136
run() (luigi.contrib.redshift.RedshiftManifestTask method), 135
run() (luigi.contrib.redshift.RedshiftUnloadTask method), 136
run() (luigi.contrib.redshift.S3CopyToTable method), 134
run() (luigi.contrib.salesforce.QuerySalesforce method), 141
run() (luigi.contrib.sge.LocalSGEJobTask method), 147
run() (luigi.contrib.spark.PySparkTask method), 150

method), 106
requires_local() (luigi.contrib.hadoop.BaseHadoopJobTask method), 106
resource_flag (luigi.contrib.lsf.LSFJobTask attribute), 114
resource_list() (luigi.RemoteScheduler method), 230
resource_list() (luigi.rpc.RemoteScheduler method), 198
resource_list() (luigi.schedulerscheduler.Scheduler method), 203
resources (luigi.Task attribute), 224
resources (luigi.Task attribute), 211
resources() (luigi.schedulerscheduler.Scheduler method), 203
respond() (luigi.worker.SchedulerMessage method), 222
restful() (luigi.contrib.salesforce.SalesforceAPI method), 142
retcode (class in luigi.retcodes), 196
retry_count (luigi.contrib.pai.PaiTask attribute), 124
retry_count (luigi.schedulerscheduler.RetryPolicy attribute), 199
retry_count (luigi.schedulerscheduler.attribute), 199
retry_count (luigi.Task attribute), 225
retry_count (luigi.Task attribute), 211
retry_delay (luigi.schedulerscheduler.attribute), 199
retry_external_tasks (luigi.worker.worker attribute), 223
retryCount (luigi.contrib.pai.PaiJob attribute), 122
RetryPolicy (class in luigi.schedulerscheduler), 198
reverse (luigi.tools.range.RangeBase attribute), 159
RootPathHandler (class in luigi.server), 205
rows() (luigi.contrib.mysqldb.CopyToTable method), 119
rows() (luigi.contrib.postgres.CopyToTable method), 127
rows() (luigi.contrib.postgres.CopyToTable method), 127
rows() (luigi.contrib.sqla.CopyToTable method), 154
rpc_message_callback() (in module luigi.worker), 223
rpc_method() (in module luigi.schedulerscheduler), 199
RPCError, 197, 231
RPCHandler (class in luigi.server), 204
run (luigi.ExternalTask attribute), 228
run (luigi.task.ExternalTask attribute), 214
run() (in module luigi), 244
run() (in module luigi.interface), 173
run() (in module luigi.server), 205
run() (luigi.contrib.batch.BatchTask method), 78
run() (luigi.contrib.beam_dataflowBeamDataflowJobTask method), 81
run() (luigi.contrib.bigquery.BigQueryCreateViewTask method), 86
run() (luigi.contrib.sparkey.SparkeyExportTask method), 150
run() (luigi.contrib.s3.CopyToTable method), 154
run() (luigi.notifications.TestNotificationsTask method), 177
run() (luigi.Task method), 227
run() (luigi.task.Task method), 213
run() (luigi.worker.ContextManagedTaskProcess method), 222
run() (luigi.worker.KeepAliveThread method), 223
run() (luigi.worker.TaskProcess method), 222
run_and_track_hadoop_job() (in module luigi.contrib.hadoop), 105
run_combiner() (luigi.contrib.hadoop.JobTask method), 108
run_hive() (in module luigi.contrib.hive), 109
run_hive_cmd() (in module luigi.contrib.hive), 109
run_hive_script() (in module luigi.contrib.hive), 109
run_job (luigi.contrib.hadoop.JobRunner attribute), 105
run_job() (luigi.contrib.bigquery.BigQueryClient method), 84
run_job() (luigi.contrib.hadoop.HadoopJobRunner method), 105
run_job() (luigi.contrib.hadoop.LocalJobRunner method), 106
run_job() (luigi.contrib.hadoop.jar.HadoopJarJobRunner method), 108
run_job() (luigi.contrib.hive.HiveQueryRunner method), 111
run_job() (luigi.contrib.scalding.ScaldingJobRunner method), 144
run_locally (luigi.contrib.sge.SGEJobTask attribute), 147
run_mapper() (luigi.contrib.hadoop.JobTask method), 108
run.reducer() (luigi.contrib.hadoop.JobTask method), 108
run_with_retcodes() (in module luigi.retcodes), 197
RunAnywayTarget (class in luigi.contrib.simulate), 148
Runner (class in luigi.contrib.mrrunner), 117
runner (luigi.contrib.beam_dataflowBeamDataflowJobTask attribute), 80
runner (luigi.contrib.beam_dataflow.DataflowParamKeys attribute), 78
running_tasks (luigi.worker.GetWorkResponse attribute), 221
runtime_flag (luigi.contrib.lsf.LSFJobTask attribute), 114
S
s3 (luigi.contrib.s3.S3Client attribute), 137
s3_load_path() (luigi.contrib.redshift.S3CopyToTable method), 133
s3_unload_path (luigi.contrib.redshift.RedshiftUnloadTask attribute), 136
S3Client (class in luigi.contrib.s3), 137
S3CopyToTable (class in luigi.contrib.redshift), 134
S3EmrTarget (class in luigi.contrib.s3), 140
S3EmrTask (class in luigi.contrib.s3), 140
S3FlagTarget (class in luigi.contrib.s3), 139
S3FlagTask (class in luigi.contrib.s3), 140
S3PathTask (class in luigi.contrib.s3), 140
S3Target (class in luigi.contrib.s3), 139
salesforce (class in luigi.contrib.salesforce), 141
SalesforceAPI (class in luigi.contrib.salesforce), 142
sample() (luigi.contrib.hadoop.LocalJobRunner method), 106
sandbox_name (luigi.contrib.salesforce.SalesforceQuerySalesforce attribute), 141
save_job_info (luigi.contrib.lsf.LSFJobTask attribute), 114
sb_security_token (luigi.contrib.salesforce.salesforce attribute), 141
Sc (luigi.contrib.pyspark_runner.SparkContextEntryPoint attribute), 129
ScaldingJobRunner (class in luigi.contrib.scalding), 144
ScaldingJobTask (class in luigi.contrib.scalding), 144
Scheduler (class in luigi.scheduler), 201
scheduler (class in luigi.scheduler), 199
scheduler_host (luigi.interface.core attribute), 172
scheduler_port (luigi.interface.core attribute), 172
scheduler_url (luigi.interface.core attribute), 172
SchedulerMessage (class in luigi.worker), 222
scheduling_error (luigi.retcodes.retcode attribute), 196
SCHEDULING_FAILED (luigi.execution_summary.LuigiStatusCode attribute), 169
SCHEDULING_FAILED (luigi.LuigiStatusCode attribute), 248
schema (luigi.contrib.bigquery.BigQueryLoadTask attribute), 85
schema (luigi.contrib.presto.PrestoTask attribute), 128
schema (luigi.contrib.s3.CopyToTable attribute), 153
security_token (luigi.contrib.salesforce.salesforce attribute), 141
Luigi Documentation, Release 2.8.13

Index 283

setup() (luigi.contrib.spark.PySparkTask method), 150
setup() (luigi.setup_logging.BaseLogging class method), 206
setup_remote() (luigi.contrib.spark.PySparkTask method), 150
SGEJobTask (class in luigi.contrib.sge), 146
shared_tmp_dir (luigi.contrib.lsf.LSFJobTask attribute), 114
shared_tmp_dir (luigi.contrib.sge.SGEJobTask attribute), 147
shmMB (luigi.contrib.pai.TaskRole attribute), 123
signal_complete() (luigi.contrib.kubernetes.KubernetesJobTask method), 113
SimpleTaskState (class in luigi.scheduler), 201
SingleProcessPool (class in luigi.worker), 222
skip_leading_rows (luigi.contrib.bigquery.BigQueryLoadTask attribute), 85
slot_to_dict() (in module luigi.contrib.pai), 121
smtp (class in luigi.notifications), 178
snakebite_autoconfig (luigi.contrib.hdfs.config.hdfs attribute), 68
SnakebiteHdfsClient (class in luigi.contrib.hdfs), 71
SOAP_NS (luigi.contrib.salesforce.SalesforceAPI attribute), 142
soql (luigi.contrib.salesforce.QuerySalesforce attribute), 141
source (luigi.contrib.presto.PrestoTask attribute), 128
source() (luigi.contrib.scaling.ScalingJobTask method), 144
source_format (luigi.contrib.bigquery.BigQueryLoadTask attribute), 84
source_format (luigi.contrib.bigquery_avro.BigQueryLoadAvro method), 87
source_uris() (luigi.contrib.bigquery.BigQueryLoadTask op (class in luigi.contrib.pyspark_runner), 129
source_uris() (luigi.contrib.bigquery_avro.BigQueryLoadAvro method), 85
SourceFormat (class in luigi.contrib.bigquery), 81
spark (luigi.contrib.pyspark_runner.SparkSessionEntryPoint op () (in module luigi.server), 205
spark_command() (luigi.contrib.spark.SparkSubmitTaskStoredProcedureTask (class in luigi.task_history), 215
spark_submit (luigi.contrib.spark.SparkSubmitTask attribute), 148
spark_version (luigi.contrib.spark.SparkSubmitTask attribute), 148
SparkContextEntryPoint (class in luigi.contrib.pyspark_runner), 129
SparkeyExportTask (class in luigi.contrib.spark), 150
SparkSessionEntryPoint (class in luigi.contrib.pyspark_runner), 129
SparkSubmitTask (class in luigi.contrib.spark), 148
spec_schema (luigi.contrib.kubernetes.KubernetesJobTask attribute), 113
SQLAlchemyTarget (class in luigi.contrib.sqla), 152
SQLAlchemyTarget.Connection (class in luigi.contrib.sqla), 153
ssh() (luigi.contrib.hadoop_jar.HadoopJarJobTask method), 108
ssl1 (luigi.notifications.smtp attribute), 178
stable_done_cooldown_secs (luigi.scheduler.scheduler attribute), 199
staging_location (luigi.contrib.beam_dataflow.BeamDataflowJobTask attribute), 80
staging_location (luigi.contrib.beam_dataflow.DataflowParamKeys attribute), 78
START (luigi.Event attribute), 245
START (luigi.event.Event attribute), 168
start (luigi.tools.range.RangeBase attribute), 159
start (luigi.tools.range.RangeByMinutesBase attribute), 161
start (luigi.tools.range.RangeDailyBase attribute), 160
start (luigi.tools.range.RangeHourlyBase attribute), 160
start (luigi.tools.range.RangeMonthly attribute), 162
start_session() (luigi.contrib.salesforce.SalesforceAPI method), 142
state (luigi.scheduler.Worker attribute), 200
state_path (luigi.scheduler.schedulerc attribute), 199
statsd_host (luigi.contrib.datadog_metric.datadog attribute), 88
statsd_port (luigi.contrib.datadog_metric.datadog attribute), 88
status_search() (luigi.tools.luigi_grep.LuigiGrep attribute), 161
stop (luigi.tools.range.RangeBase attribute), 159
stop (luigi.tools.range.RangeByMinutesBase attribute), 161
stream_for_searching_tracking_url (luigi.contrib.external_program.ExternalProgramTask attribute), 99
stream_for_searching_tracking_url (luigi.contrib.spark.SparkSubmitTask attribute), 148
submit_job() (luigi.contrib.batch.BatchClient method), 223
SUCCESS
SUCCESS
SUCCESS
SUCCESS

subnetwork (luigi.contrib.beam_dataflowBeamDataflowJobTask
attribute), 80

su

submit_spark_job()
(subluigi.contrib.dataproc.DataprocBaseTask
method), 88

submit_pyspark_job()
(subluigi.contrib.dataproc.DataprocBaseTask
method), 88

submit_job()
(subluigi.contrib.dataproc.DataprocBaseTask
method), 88


table_attributes

summary()

(LuigiStatus
code
attribute), 109

table_location() (luigi.contrib.hive.HiveCommandClient
method), 109

table_location() (luigi.contrib.hive.MetastoreClient
method), 109

table_location() (luigi.contrib.hive.WarehouseHiveClient
method), 110

table_schema() (luigi.contrib.hive.ApacheHiveCommandClient
method), 110

table_schema() (luigi.contrib.hive.HiveClient
method), 110

table_type (luigi.contrib.redshift.S3CopyToTable
attribute), 110

take_lock (luigi.interface.core attribute), 173

Target (class in luigi), 229

Target (class in luigi.target), 206

Task (class in luigi), 210

Task (class in luigi.scheduler), 200

Task (class in luigi.task), 210

task_id (luigi.contrib.hadoop.BaseHadoopJobTask
attribute), 109

task_id (luigi.db_task_history.TaskEvent attribute), 168

task_id (luigi.db_task_history.TaskParameter
attribute), 167

task_id (luigi.db_task_history.TaskRecord
attribute), 168

task_id (luigi.worker.GetWorkResponse attribute), 221

task_id_str () (in module luigi.task), 210

task_limit (luigi.tools.range.RangeBase attribute), 159
Index
Index

U
udf_resource_uris (luigi.contrib.bigquery.BigQueryRunQueryTask attribute), 86
unhandled_exception (luigi.retcodes.recode attribute), 196
unique (luigi.contrib.simulate.RunAnywayTarget attribute), 148
UnknownParameterException, 179
unload_options (luigi.contrib.redshift.RedshiftUnloadTask attribute), 136
unload_query (luigi.contrib.redshift.RedshiftUnloadTask attribute), 136
unpause() (luigi.RemoteScheduler method), 230
unpause() (luigi.rpc.RemoteScheduler method), 198
unpause() (luigi.scheduler.Scheduler method), 202
update() (luigi.batch_notifier.BatchNotifier method), 164
update() (luigi.scheduler.Worker method), 200
update_id (luigi.contrib.rdbms.CopyToTable attribute), 225
update_id (luigi.contrib.rdbms.CopyToTable attribute), 130
update_id (luigi.contrib.redshift.KillOpenRedshiftSessions attribute), 136
update_id() (luigi.contrib.esindex.CopyToIndex attribute), 98
update_metrics_task_started() (luigi.RemoteScheduler method), 231
update_metrics_task_started() (luigi.rpc.RemoteScheduler method), 198
update_metrics_task_started() (luigi.scheduler.Scheduler method), 203
update_progress_percentage() (luigi.contrib.simulate.RunAnywayTarget attribute), 148
update_resource() (luigi.RemoteScheduler method), 231
update_resource() (luigi.rpc.RemoteScheduler method), 198
update_resource() (luigi.scheduler.Scheduler method), 202
update_resources() (luigi.contrib.esindex.CopyToIndex attribute), 98
update_resources() (luigi.contrib.rdbms.CopyToTable attribute), 222
update_status() (luigi.contrib.esindex.CopyToIndex method), 201
update_status() (luigi.contrib.yarn.YarnCommand method), 201
update_status_message() (luigi.contrib.esindex.CopyToIndex method), 222
update_tracking_url() (luigi.contrib.esindex.CopyToIndex method), 222
update_view() (luigi.contrib.bigquery.BigQueryClient method), 83
upload() (luigi.contrib.dropbox.DropboxClient method), 92
upload() (luigi.contrib.hdfs.webhdfs_client.WebHdfsClient method), 76
upstream (class in luigi.tools.deps), 157
UPSTREAM_SEVERITY_KEY () (in module luigi.scheduler), 198
uri (luigi.contrib.bigquery.BQTable attribute), 82
url (luigi.contrib.hdfs.webhdfs_client.WebHdfsClient attribute), 75
URLLibFetcher (class in luigi.rpc), 197
use_cmdline_section (luigi.interface.core attribute), 172
use_cmdline_section (luigi.Task attribute), 225
use_cmdline_section (luigi.task.Task attribute),
WrapperTask (class in luigi), 228
WrapperTask (class in luigi.task), 214
writable() (luigi.contrib.dropbox.ReadableDropboxFile method), 92
writable() (luigi.contrib.s3.ReadableS3File method), 139
writable() (luigi.format.InputPipeProcessWrapper method), 170
writable() (luigi.format.OutputPipeProcessWrapper method), 170
write() (luigi.contrib.mongodb.MongoCellTarget method), 116
write() (luigi.contrib.mongodb.MongoRangeTarget method), 116
write() (luigi.format.MixedUnicodeBytesWrapper method), 170
write() (luigi.format.NewlineWrapper method), 170
WRITE_APPEND (luigi.contrib.bigquery.WriteDisposition attribute), 81
write_disposition (luigi.contrib.bigquery.BigQueryLoadTask attribute), 85
write_disposition (luigi.contrib.bigquery.BigQueryRunQueryTask attribute), 85
WRITE_EMPTY (luigi.contrib.bigquery.WriteDisposition attribute), 81
write_pid() (in module luigi.process), 196
WRITE_TRUNCATE (luigi.contrib.bigquery.WriteDisposition attribute), 81
WriteDisposition (class in luigi.contrib.bigquery), 81
writeLine() (luigi.format.OutputPipeProcessWrapper method), 170
writelines() (luigi.format.MixedUnicodeBytesWrapper method), 171
writelines() (luigi.format.NewlineWrapper method), 170
writer() (luigi.contrib.hadoop.JobTask method), 107
WRITES_BEFORE_FLUSH (luigi.format.OutputPipeProcessWrapper attribute), 170

Y
Year (class in luigi.date_interval), 166
YearParameter (class in luigi), 234
YearParameter (class in luigi.parameter), 183

Z
zone (luigi.contrib.beam_dataflow.BeamDataflowJobTask attribute), 80