huey Documentation

Release 0.4.9

charles leifer

January 02, 2016

a lightweight alternative.

- written in python
- no deps outside stdlib, except redis (or roll your own backend)
- support for django

supports:

- multi-threaded task execution
- scheduled execution at a given time
- periodic execution, like a crontab
- retrying tasks that fail
- task result storage

CHAPTER 1

Huey's API

```
from huey import RedisHuey, crontab

huey = RedisHuey('my-app', host='redis.myapp.com')

@huey.task()
def add_numbers(a, b):
    return a + b

@huey.periodic_task(crontab(minute='0', hour='3'))
def nightly_backup():
    sync_all_data()
```

named after my cat

Contents:

1.1 Installing

huey can be installed very easily using pip.

```
pip install huey
```

huey has no dependencies outside the standard library, but currently the only fully-implemented queue backend it ships with requires redis. To use the redis backend, you will need to install the python client.

```
pip install redis
```

1.1.1 Using git

If you want to run the very latest, feel free to pull down the repo from github and install by hand.

```
git clone https://github.com/coleifer/huey.git
cd huey
python setup.py install
```

You can run the tests using the test-runner:

```
python setup.py test
```

Browse the source code online at https://github.com/coleifer/huey

1.2 Upgrading

With the release of Huey 0.4, there are a number of changes to the way things work. Unfortunately, many of these changes are backwards incompatible as this was a pretty big rewrite. What follows is a list of things that changed and how to upgrade your code.

To see working examples, be sure to check out the two example apps that ship with huey, or view the source on GitHub.

1.2.1 Invoker became Huey

Invoker was a terrible name. It has been renamed to the much-better "Huey", which serves the same purpose. *Huey* accepts mostly the same args as `Invoker` did, with the exception of the `task_store` argument which has been removed as it was redundant with `result_store`.

```
queue = RedisBlockingQueue(name='foo')
data_store = RedisDataStore(name='foo')

# OLD
invoker = Invoker(queue, data_store)

# NEW
huey = Huey(queue, data_store)
```

1.2.2 Decorators are methods on Huey

Formerly if you wanted to decorate a function you would import one of the decorators from `huey.decorators`. Instead, these decorators are now implemented as methods on the *Huey* object (*task()* and *periodic_task()*).

```
# OLD
@queue_command(invoker)
def do_something(a, b, c):
    return a + b + c

# NEW
@huey.task()
def do_something(a, b, c):
    return a + b + c
```

The arguments are the same, except there is no need to pass in the `invoker` object anymore.

- `queue_command` became *Huey.task()*
- `periodic_command` became *Huey.periodic_task()*

1.2.3 No more `BaseConfiguration`

Configuring the consumer used to be a bit obnoxious because of the need to duplicate information in the `BaseConfiguration` subclass that was already present in your `Invoker`. The `BaseConfiguration` object is gone – now instead of pointing your consumer at the config object, point it at your application's *Huey* instance:

```
# OLD
huey_consumer.py path.to.Configuration

# NEW
huey_consumer.py path.to.huey_instance
```

Options that were formerly hard-coded into the configuration, like threads and logfile, are now exposed as command-line arguments.

For more information check out the *consumer docs*.

1.2.4 Simplified Django Settings

The Django settings are now a bit more simplified. In fact, if you are running Redis locally, Huey will "just work". The new huey settings look like this:

```
HUEY = {
    'backend': 'huey.backends.redis_backend',  # required.
    'name': 'unique name',
    'connection': {'host': 'localhost', 'port': 6379},

    # Options to pass into the consumer when running ``manage.py run_huey``
    'consumer_options': {'workers': 4},
}
```

Additionally, the imports changed. Now everything is imported from `djhuey`:

```
# NEW
from huey.djhuey import task, periodic_task, crontab

@task()
def some_fn(a, b):
    return a + b
```

1.2.5 Django task autodiscovery

The `run_huey` management command no longer auto-imports `commands.py` – instead it will auto-import `tasks.py`.

1.3 Getting Started

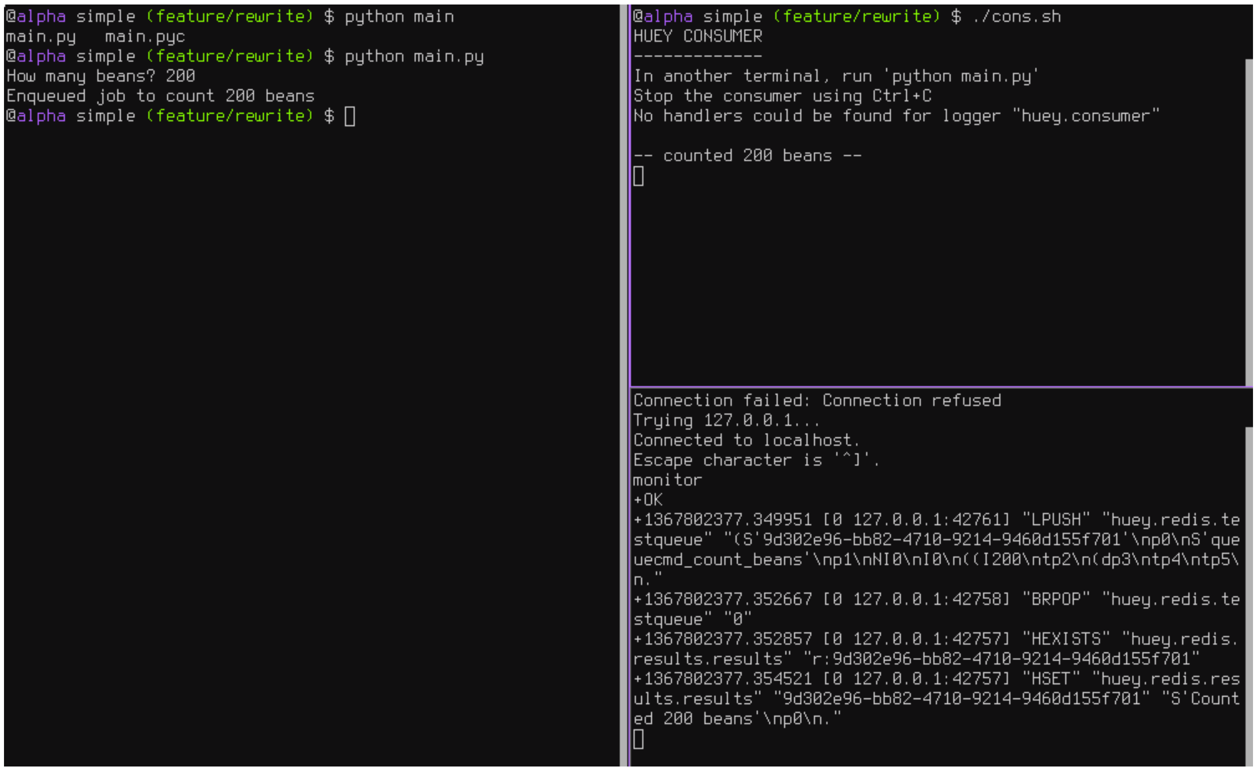
The goal of this document is to help you get running quickly and with as little fuss as possible.

1.3.1 General guide

There are three main components (or processes) to consider when running huey:

- the producer(s), i.e. a web application
- the consumer(s), which executes jobs placed into the queue
- the queue where tasks are stored, e.g. Redis

These three processes are shown in the screenshots that follow. The left-hand pane shows the producer: a simple program that asks the user for input on how many "beans" to count. In the top-right, the consumer is running. It is doing the actual "computation", for example printing the number of beans counted. In the bottom-right is the queue, Redis in this example. We can see the tasks being enqueued (`LPUSH`) and read (`BRPOP`) from the database.



```
@alpha simple (feature/rewrite) $ python main
main.py   main.pyc
@alpha simple (feature/rewrite) $ python main.py
How many beans? 200
Enqueued job to count 200 beans
@alpha simple (feature/rewrite) $
```

```
@alpha simple (feature/rewrite) $ ./cons.sh
HUEY CONSUMER
-------------
In another terminal, run 'python main.py'
Stop the consumer using Ctrl+C
No handlers could be found for logger "huey.consumer"

-- counted 200 beans --
```

```
Connection failed: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
monitor
+OK
+1367802377.349951 [0 127.0.0.1:42761] "LPUSH" "huey.redis.te
stqueue" "(S'9d302e96-bb82-4710-9214-9460d155f701'\np0\nS'que
uecmd_count_beans'\np1\nNI0\nI0\n((I200\ntp2\n(dp3\ntp4\ntp5\
n."
+1367802377.352667 [0 127.0.0.1:42758] "BRPOP" "huey.redis.te
stqueue" "0"
+1367802377.352857 [0 127.0.0.1:42757] "HEXISTS" "huey.redis.
results.results" "r:9d302e96-bb82-4710-9214-9460d155f701"
+1367802377.354521 [0 127.0.0.1:42757] "HSET" "huey.redis.res
ults.results" "9d302e96-bb82-4710-9214-9460d155f701" "S'Count
ed 200 beans'\np0\n."
```

Trying it out yourself

Assuming you've got *huey installed*, let's look at the code from this example.

The first step is to configure your queue. The consumer needs to be pointed at a `Huey` instance, which specifies which backend to use.

```python
# config.py
from huey import Huey
from huey.backends.redis_backend import RedisBlockingQueue

queue = RedisBlockingQueue('test-queue', host='localhost', port=6379)
huey = Huey(queue)
```

The interesting parts of this configuration module are the `Huey` object and the `RedisBlockingQueue` object. The `queue` is responsible for storing and retrieving messages, and the `huey` is used by your application code to coordinate function calls with a queue backend. We'll see how the `huey` is used when looking at the actual function responsible for counting beans:

```python
# tasks.py
from config import huey # import the huey we instantiated in config.py


@huey.task()
def count_beans(num):
    print '-- counted %s beans --' % num
```

The above example shows the API for writing "tasks" that are executed by the queue consumer – simply decorate the code you want executed by the consumer with the `task()` decorator and when it is called, the main process will return *immediately* after enqueueing the function call.

The main executable is very simple. It imports both the configuration **and** the tasks - this is to ensure that when we run the consumer by pointing it at the configuration, the tasks are also imported and loaded into memory.

```
# main.py
from config import huey  # import our "huey" object
from tasks import count_beans  # import our task


if __name__ == '__main__':
    beans = raw_input('How many beans? ')
    count_beans(int(beans))
    print 'Enqueued job to count %s beans' % beans
```

To run these scripts, follow these steps:

1. Ensure you have Redis running locally
2. Ensure you have *installed huey*
3. Start the consumer: `huey_consumer.py main.huey` (notice this is "main.huey" and not "config.huey").
4. Run the main program: `python main.py`

Getting results from jobs

The above example illustrates a "send and forget" approach, but what if your application needs to do something with the results of a task? To get results from your tasks, we'll set up the `RedisDataStore` by adding the following lines to the `config.py` module:

```
from huey import Huey
from huey.backends.redis_backend import RedisBlockingQueue
from huey.backends.redis_backend import RedisDataStore  # ADD THIS LINE


queue = RedisBlockingQueue('test-queue', host='localhost', port=6379)
result_store = RedisDataStore('results', host='localhost', port=6379)  # ADDED

huey = Huey(queue, result_store=result_store) # ADDED result store
```

We can actually shorten this code to:

```
from huey import RedisHuey

huey = RedisHuey('test-queue', host='localhost', port=6379)
```

To better illustrate getting results, we'll also modify the `tasks.py` module to return a string rather in addition to printing to stdout:

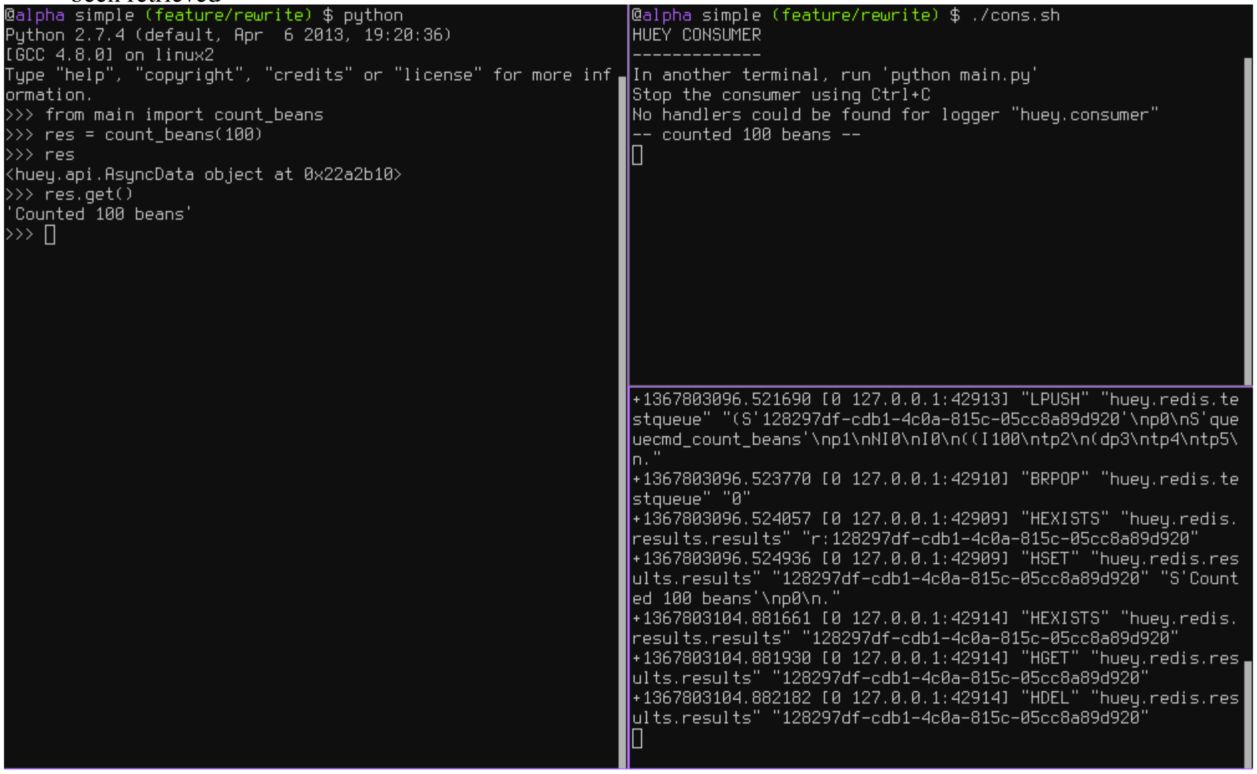
```
from config import huey


@huey.task()
def count_beans(num):
    print '-- counted %s beans --' % num
    return 'Counted %s beans' % num
```

We're ready to fire up the consumer. Instead of simply executing the main program, though, we'll start an interpreter and run the following:

```
>>> from main import count_beans
>>> res = count_beans(100)
>>> res  # what is "res" ?
<huey.api.AsyncData object at 0xb7471a4c>
>>> res.get()  # get the result of this task
'Counted 100 beans'
```

Following the same layout as our last example, here is a screenshot of the three main processes at work:

1. Top-left, interpreter which produces a job then asks for the result
2. Top-right, the consumer which runs the job and stores the result
3. Bottom-right, the Redis database, which we can see is storing the results and then deleting them after they've been retrieved



```
@alpha simple (feature/rewrite) $ python
Python 2.7.4 (default, Apr  6 2013, 19:20:36)
[GCC 4.8.0] on linux2
Type "help", "copyright", "credits" or "license" for more inf
ormation.
>>> from main import count_beans
>>> res = count_beans(100)
>>> res
<huey.api.AsyncData object at 0x22a2b10>
>>> res.get()
'Counted 100 beans'
>>> 
```

```
@alpha simple (feature/rewrite) $ ./cons.sh
HUEY CONSUMER
-------------
In another terminal, run 'python main.py'
Stop the consumer using Ctrl+C
No handlers could be found for logger "huey.consumer"
-- counted 100 beans --
```

```
+1367803096.521690 [0 127.0.0.1:42913] "LPUSH" "huey.redis.te
stqueue" "(S'128297df-cdb1-4c0a-815c-05cc8a89d920'\np0\nS'que
uecmd_count_beans'\np1\nNI0\nI0\n((I100\ntp2\n(dp3\ntp4\ntp5\
n."
+1367803096.523770 [0 127.0.0.1:42910] "BRPOP" "huey.redis.te
stqueue" "0"
+1367803096.524057 [0 127.0.0.1:42909] "HEXISTS" "huey.redis.
results.results" "r:128297df-cdb1-4c0a-815c-05cc8a89d920"
+1367803096.524936 [0 127.0.0.1:42909] "HSET" "huey.redis.res
ults.results" "128297df-cdb1-4c0a-815c-05cc8a89d920" "S'Count
ed 100 beans'\np0\n."
+1367803104.881661 [0 127.0.0.1:42914] "HEXISTS" "huey.redis.
results.results" "128297df-cdb1-4c0a-815c-05cc8a89d920"
+1367803104.881930 [0 127.0.0.1:42914] "HGET" "huey.redis.res
ults.results" "128297df-cdb1-4c0a-815c-05cc8a89d920"
+1367803104.882182 [0 127.0.0.1:42914] "HDEL" "huey.redis.res
ults.results" "128297df-cdb1-4c0a-815c-05cc8a89d920"
```

Executing tasks in the future

It is often useful to enqueue a particular task to execute at some arbitrary time in the future, for example, mark a blog entry as published at a certain time.

This is very simple to do with huey. Returning to the interpreter session from the last section, let's schedule a bean counting to happen one minute in the future and see how huey handles it. Execute the following:

```
>>> import datetime
>>> res = count_beans.schedule(args=(100,), delay=60)
>>> res
<huey.api.AsyncData object at 0xb72915ec>
>>> res.get()  # this returns None, no data is ready
>>> res.get()  # still no data...
>>> res.get(blocking=True)  # ok, let's just block until its ready
'Counted 100 beans'
```

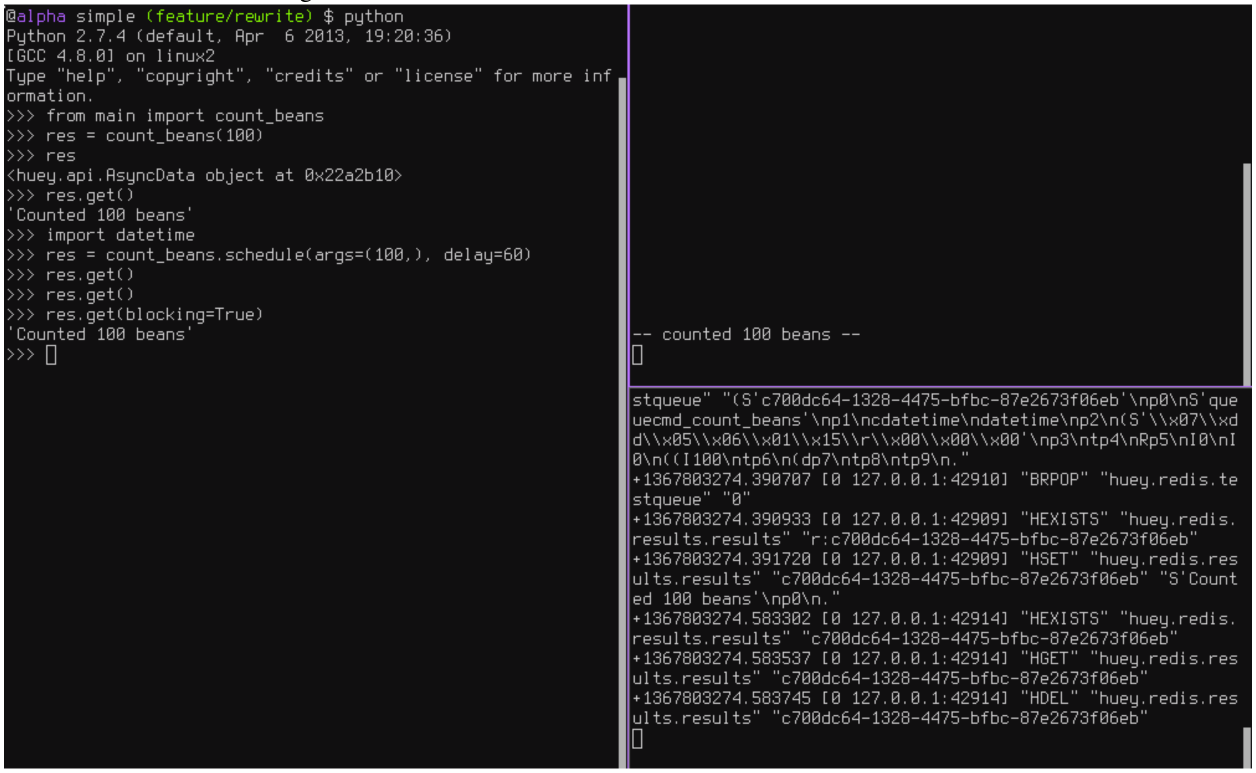
You can specify an "estimated time of arrival" as well using datetimes:

```
>>> in_a_minute = datetime.datetime.now() + datetime.timedelta(seconds=60)
>>> res = count_beans.schedule(args=(100,), eta=in_a_minute)
```

Looking at the redis output, we see the following (simplified for reability):

```
+1325563365.910640 "LPUSH" count_beans(100)
+1325563365.911912 "BRPOP" wait for next job
+1325563365.912435 "HSET" store 'Counted 100 beans'
+1325563366.393236 "HGET" retrieve result from task
+1325563366.393464 "HDEL" delete result after reading
```

Here is a screenshot showing the same:



```
@alpha simple (feature/rewrite) $ python
Python 2.7.4 (default, Apr  6 2013, 19:20:36)
[GCC 4.8.0] on linux2
Type "help", "copyright", "credits" or "license" for more inf
ormation.
>>> from main import count_beans
>>> res = count_beans(100)
>>> res
<huey.api.AsyncData object at 0x22a2b10>
>>> res.get()
'Counted 100 beans'
>>> import datetime
>>> res = count_beans.schedule(args=(100,), delay=60)
>>> res.get()
>>> res.get()
>>> res.get(blocking=True)
'Counted 100 beans'
>>> 
```

```
-- counted 100 beans --
```

```
stqueue" "(S'c700dc64-1328-4475-bfbc-87e2673f06eb'\np0\nS'que
uecmd_count_beans'\np1\ncdatetime\ndatetime\np2\n(S'\\x07\\xd
d\\x05\\x06\\x01\\x15\\r\\x00\\x00\\x00'\np3\ntp4\nRp5\nI0\nI
0\n((I100\ntp6\n(dp7\ntp8\ntp9\n."
+1367803274.390707 [0 127.0.0.1:42910] "BRPOP" "huey.redis.te
stqueue" "0"
+1367803274.390933 [0 127.0.0.1:42909] "HEXISTS" "huey.redis.
results.results" "r:c700dc64-1328-4475-bfbc-87e2673f06eb"
+1367803274.391720 [0 127.0.0.1:42909] "HSET" "huey.redis.res
ults.results" "c700dc64-1328-4475-bfbc-87e2673f06eb" "S'Count
ed 100 beans'\np0\n."
+1367803274.583302 [0 127.0.0.1:42914] "HEXISTS" "huey.redis.
results.results" "c700dc64-1328-4475-bfbc-87e2673f06eb"
+1367803274.583537 [0 127.0.0.1:42914] "HGET" "huey.redis.res
ults.results" "c700dc64-1328-4475-bfbc-87e2673f06eb"
+1367803274.583745 [0 127.0.0.1:42914] "HDEL" "huey.redis.res
ults.results" "c700dc64-1328-4475-bfbc-87e2673f06eb"
```

Retrying tasks that fail

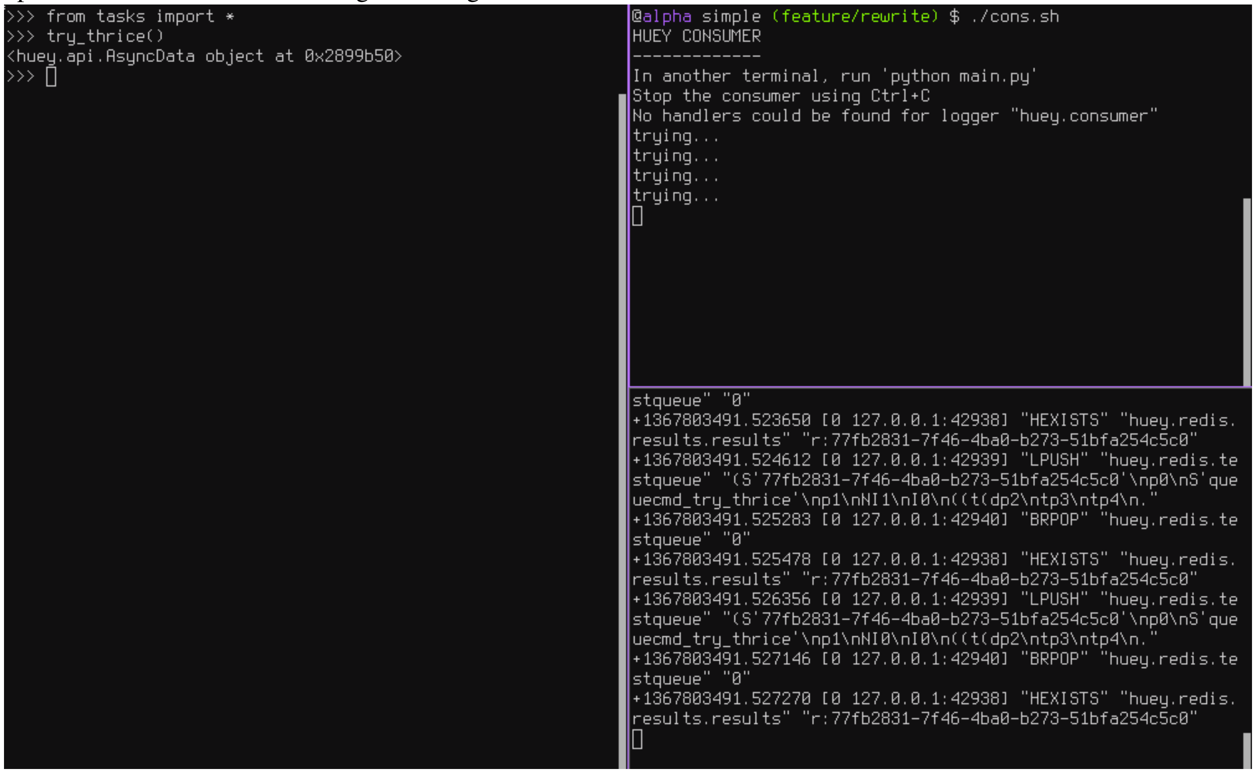
Huey supports retrying tasks a finite number of times. If an exception is raised during the execution of the task and `retries` have been specified, the task will be re-queued and tried again, up to the number of retries specified.

Here is a task that will be retried 3 times and will blow up every time:

```
# tasks.py
from config import huey


@huey.task()
def count_beans(num):
    print '-- counted %s beans --' % num
    return 'Counted %s beans' % num

@huey.task(retries=3)
```

```
def try_thrice():
    print 'trying....'
    raise Exception('nope')
```

The console output shows our task being called in the main interpreter session, and then when the consumer picks it up and executes it we see it failing and being retried:



```
>>> from tasks import *
>>> try_thrice()
<huey.api.AsyncData object at 0x2899b50>
>>> 
```

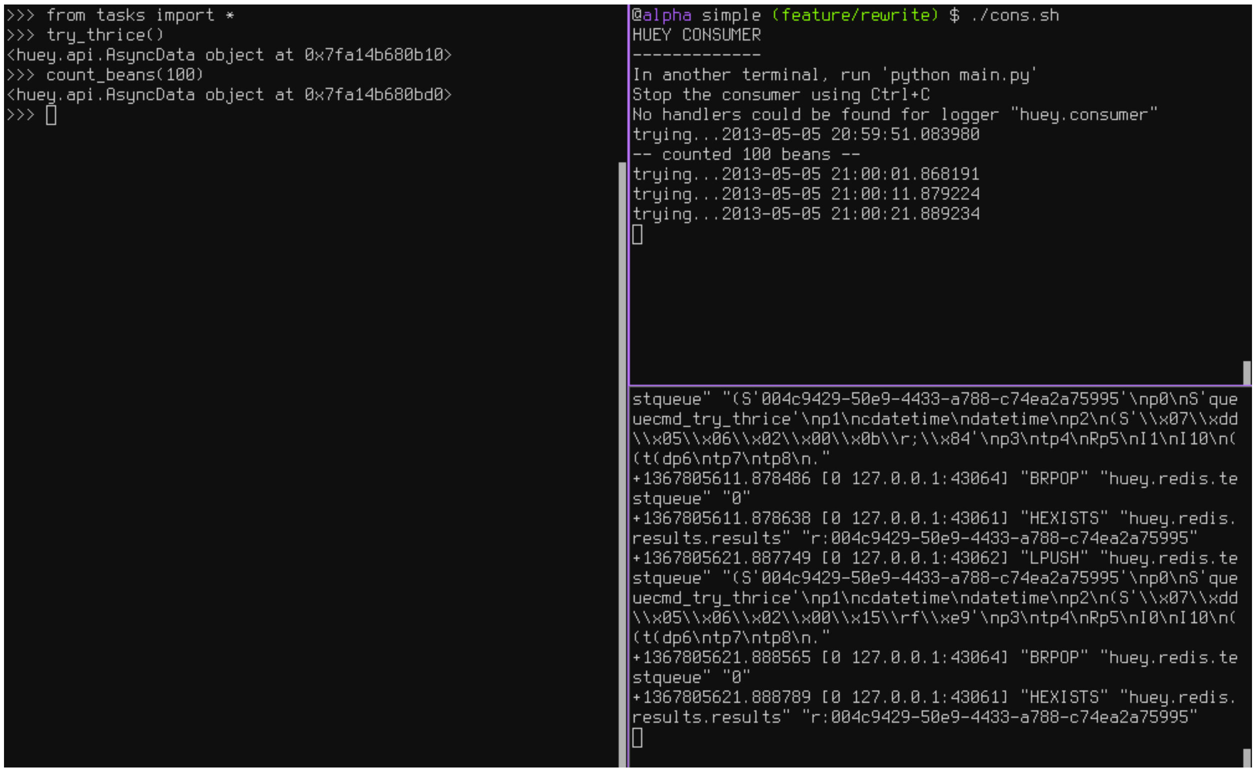
```
@alpha simple (feature/rewrite) $ ./cons.sh
HUEY CONSUMER
-------------
In another terminal, run 'python main.py'
Stop the consumer using Ctrl+C
No handlers could be found for logger "huey.consumer"
trying...
trying...
trying...
trying...
```

```
stqueue" "0"
+1367803491.523650 [0 127.0.0.1:42938] "HEXISTS" "huey.redis.
results.results" "r:77fb2831-7f46-4ba0-b273-51bfa254c5c0"
+1367803491.524612 [0 127.0.0.1:42939] "LPUSH" "huey.redis.te
stqueue" "(S'77fb2831-7f46-4ba0-b273-51bfa254c5c0'\np0\nS'que
uecmd_try_thrice'\np1\nNI1\nI0\n((t(dp2\ntp3\ntp4\n."
+1367803491.525283 [0 127.0.0.1:42940] "BRPOP" "huey.redis.te
stqueue" "0"
+1367803491.525478 [0 127.0.0.1:42938] "HEXISTS" "huey.redis.
results.results" "r:77fb2831-7f46-4ba0-b273-51bfa254c5c0"
+1367803491.526356 [0 127.0.0.1:42939] "LPUSH" "huey.redis.te
stqueue" "(S'77fb2831-7f46-4ba0-b273-51bfa254c5c0'\np0\nS'que
uecmd_try_thrice'\np1\nNI0\nI0\n((t(dp2\ntp3\ntp4\n."
+1367803491.527146 [0 127.0.0.1:42940] "BRPOP" "huey.redis.te
stqueue" "0"
+1367803491.527270 [0 127.0.0.1:42938] "HEXISTS" "huey.redis.
results.results" "r:77fb2831-7f46-4ba0-b273-51bfa254c5c0"
```

Oftentimes it is a good idea to wait a certain amount of time between retries. You can specify a *delay* between retries, in seconds, which is the minimum time before the task will be retried. Here we've modified the command to include a delay, and also to print the current time to show that its working.

```
# tasks.py
from datetime import datetime

from config import huey

@huey.task(retries=3, retry_delay=10)
def try_thrice():
    print 'trying....%s' % datetime.now()
    raise Exception('nope')
```

The console output below shows the task being retried, but in between retries I've also "counted some beans" – that gets executed normally, in between retries.



```
>>> from tasks import *
>>> try_thrice()
<huey.api.AsyncData object at 0x7fa14b680b10>
>>> count_beans(100)
<huey.api.AsyncData object at 0x7fa14b680bd0>
>>> 
```

```
@alpha simple (feature/rewrite) $ ./cons.sh
HUEY CONSUMER
-------------
In another terminal, run 'python main.py'
Stop the consumer using Ctrl+C
No handlers could be found for logger "huey.consumer"
trying...2013-05-05 20:59:51.083980
-- counted 100 beans --
trying...2013-05-05 21:00:01.868191
trying...2013-05-05 21:00:11.879224
trying...2013-05-05 21:00:21.889234
```

```
stqueue" "(S'004c9429-50e9-4433-a788-c74ea2a75995'\np0\nS'que
uecmd_try_thrice'\np1\ncdatetime\ndatetime\np2\n(S'\\x07\\xdd
\\x05\\x06\\x02\\x00\\x0b\\r;\\x84'\np3\ntp4\nRp5\nI1\nI10\n(
(t(dp6\ntp7\ntp8\n."
+1367805611.878486 [0 127.0.0.1:43064] "BRPOP" "huey.redis.te
stqueue" "0"
+1367805611.878638 [0 127.0.0.1:43061] "HEXISTS" "huey.redis.
results.results" "r:004c9429-50e9-4433-a788-c74ea2a75995"
+1367805621.887749 [0 127.0.0.1:43062] "LPUSH" "huey.redis.te
stqueue" "(S'004c9429-50e9-4433-a788-c74ea2a75995'\np0\nS'que
uecmd_try_thrice'\np1\ncdatetime\ndatetime\np2\n(S'\\x07\\xdd
\\x05\\x06\\x02\\x00\\x15\\rf\\xe9'\np3\ntp4\nRp5\nI0\nI10\n(
(t(dp6\ntp7\ntp8\n."
+1367805621.888565 [0 127.0.0.1:43064] "BRPOP" "huey.redis.te
stqueue" "0"
+1367805621.888789 [0 127.0.0.1:43061] "HEXISTS" "huey.redis.
results.results" "r:004c9429-50e9-4433-a788-c74ea2a75995"
```

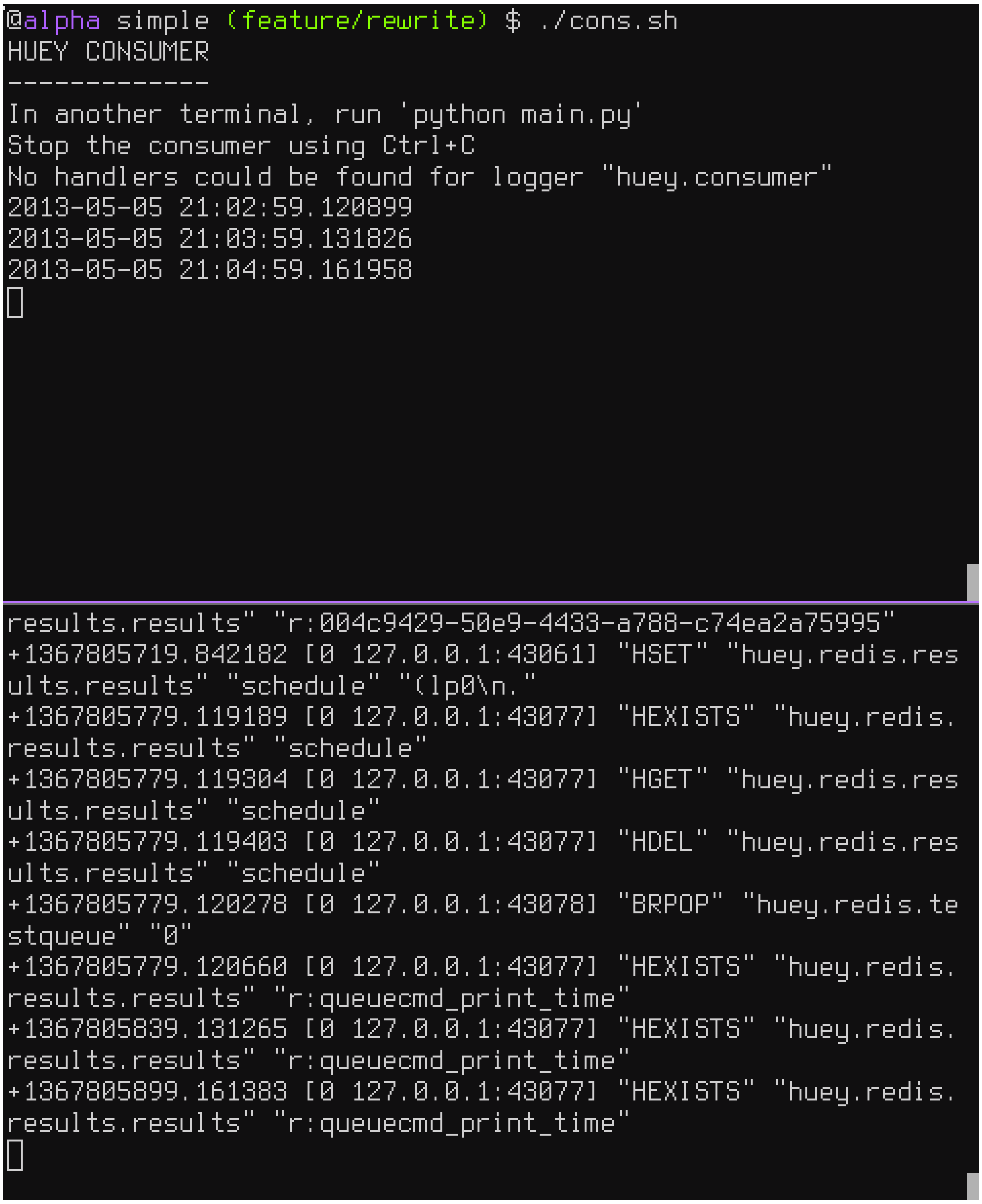
Executing tasks at regular intervals

The final usage pattern supported by huey is the execution of tasks at regular intervals. This is modeled after `crontab` behavior, and even follows similar syntax. Tasks run at regular intervals and should not return meaningful results, nor should they accept any parameters.

Let's add a new task that prints the time every minute – we'll use this to test that the consumer is executing the tasks on schedule.

```python
# tasks.py
from datetime import datetime
from huey import crontab

from config import huey

@huey.periodic_task(crontab(minute='*'))
def print_time():
    print datetime.now()
```

Now, when we run the consumer it will start printing the time every minute:



```
@alpha simple (feature/rewrite) $ ./cons.sh
HUEY CONSUMER
-------------
In another terminal, run 'python main.py'
Stop the consumer using Ctrl+C
No handlers could be found for logger "huey.consumer"
2013-05-05 21:02:59.120899
2013-05-05 21:03:59.131826
2013-05-05 21:04:59.161958
```

```
results.results" "r:004c9429-50e9-4433-a788-c74ea2a75995"
+1367805719.842182 [0 127.0.0.1:43061] "HSET" "huey.redis.res
ults.results" "schedule" "(lp0\n."
+1367805779.119189 [0 127.0.0.1:43077] "HEXISTS" "huey.redis.
results.results" "schedule"
+1367805779.119304 [0 127.0.0.1:43077] "HGET" "huey.redis.res
ults.results" "schedule"
+1367805779.119403 [0 127.0.0.1:43077] "HDEL" "huey.redis.res
ults.results" "schedule"
+1367805779.120278 [0 127.0.0.1:43078] "BRPOP" "huey.redis.te
stqueue" "0"
+1367805779.120660 [0 127.0.0.1:43077] "HEXISTS" "huey.redis.
results.results" "r:queuecmd_print_time"
+1367805839.131265 [0 127.0.0.1:43077] "HEXISTS" "huey.redis.
results.results" "r:queuecmd_print_time"
+1367805899.161383 [0 127.0.0.1:43077] "HEXISTS" "huey.redis.
results.results" "r:queuecmd_print_time"
```

1.3.2 Preventing tasks from executing

It is possible to prevent tasks from executing. This applies to normal tasks, tasks scheduled in the future, and periodic tasks.

Note: In order to "revoke" tasks you will need to specify a `result_store` when instantiating your *Huey* object.

Canceling a normal task or one scheduled in the future

You can cancel a normal task provided the task has not started execution by the consumer:

```
# count some beans
res = count_beans(10000000)

# provided the command has not started executing yet, you can
# cancel it by calling revoke() on the AsyncData object
res.revoke()
```

The same applies to tasks that are scheduled in the future:

```
res = count_beans.schedule(args=(100000,), eta=in_the_future)
res.revoke()

# and you can actually change your mind and restore it, provided
# it has not already been "skipped" by the consumer
res.restore()
```

Canceling tasks that execute periodically

When we start dealing with periodic tasks, the options for revoking get a bit more interesting.

We'll be using the print time command as an example:

```
@huey.task(crontab(minute='*'))
def print_time():
    print datetime.now()
```

We can prevent a periodic task from executing on the next go-round:

```
# only prevent it from running once
print_time.revoke(revoke_once=True)
```

Since the above task executes every minute, what we will see is that the output will skip the next minute and then resume normally.

We can prevent a task from executing until a certain time:

```
# prevent printing time for 10 minutes
now = datetime.datetime.utcnow()
in_10 = now + datetime.timedelta(seconds=600)

print_time.revoke(revoke_until=in_10)
```

Note: Remember to use UTC if the consumer is using UTC.

Finally, we can prevent the task from running indefinitely:

```
# will not print time until we call revoke() again with
# different parameters or restore the task
print_time.revoke()
```

At any time we can restore the task and it will resume normal execution:

```
print_time.restore()
```

Reading more

That sums up the basic usage patterns of huey. Below are links for details on other aspects of the API:

- `Huey` - responsible for coordinating executable tasks and queue backends
- `task()` - decorator to indicate an executable task
- `periodic_task()` - decorator to indicate a task that executes at periodic intervals
- `crontab()` - a function for defining what intervals to execute a periodic command
- `BaseQueue` - the queue interface and writing your own backends
- `BaseDataStore` - the simple data store used for results and schedule serialization

Also check out the *notes on running the consumer*.

Note: If you're using Django, check out the *django integration*.

1.4 Consuming Tasks

To run the consumer, simply point it at the "import path" to your application's `Huey` instance. For example, here is how I run it on my blog:

```
huey_consumer.py blog.main.huey --logfile=../logs/huey.log
```

The concept of the "import path" has been the source of a few questions, but its actually quite simple. It is simply the dotted-path you might use if you were to try and import the "huey" object in the interactive interpreter:

```
>>> from blog.main import huey
```

You may run into trouble though when "blog" is not on your python-path. To work around this:

1. Manually specify your pythonpath: `PYTHONPATH=/some/dir/:$PYTHONPATH huey_consumer.py blog.main.huey`.
2. Run `huey_consumer.py` from the directory your config module is in. I use supervisord to manage my huey process, so I set the `directory` to the root of my site.
3. Create a wrapper and hack `sys.path`.

Warning: If you plan to use supervisord to manage your consumer process, be sure that you are running the consumer directly and without any intermediary shell scripts. Shell script wrappers interfere with supervisor's ability to terminate and restart the consumer Python process. For discussion see GitHub issue 88.

1.4.1 Options for the consumer

The following table lists the options available for the consumer as well as their default values.

-l, --logfile Path to file used for logging. When a file is specified, by default Huey will use a rotating file handler (1MB / chunk) with a maximum of 3 backups. You can attach your own handler (`huey.logger`) as well. The default loglevel is `INFO`.

-v, --verbose

Verbose logging (equates to `DEBUG` level). If no logfile is specified and verbose is set, then the consumer will log to the console. **This is very useful for testing/debugging.**

-q, --quiet Only log errors. The default loglevel for the consumer is `INFO`.

-w, --workers Number of worker threads, the default is `1` thread but for applications that have many I/O bound tasks, increasing this number may lead to greater throughput.

-p, --periodic Indicate that this consumer process should start a thread dedicated to enqueueing "periodic" tasks (crontab-like functionality). This defaults to `True`, so should not need to be specified in practice.

-n, --no-periodic Indicate that this consumer process should *not* enqueue periodic tasks.

-d, --delay When using a "polling"-type queue backend, the amount of time to wait between polling the backend. Default is 0.1 seconds.

-m, --max-delay The maximum amount of time to wait between polling, if using weighted backoff. Default is 10 seconds.

-b, --backoff The amount to back-off when polling for results. Must be greater than one. Default is 1.15.

-u, --utc Indicates that the consumer should use UTC time for all tasks, crontabs and scheduling. Default is True, so in practice you should not need to specify this option.

--localtime Indicates that the consumer should use localtime for all tasks, crontabs and scheduling. Default is False.

Examples

Running the consumer with 8 threads, a logfile for errors only, and a very short polling interval:

```
    huey_consumer.py my.app.huey -l /var/log/app.huey.log -w 8 -b 1.1 -m 1.0
```

Running single-threaded without a crontab and logging to stdout:

```
huey_consumer.py my.app.huey -v -n
```

1.4.2 Consumer Internals

The consumer is composed of 3 types of threads:

- Worker threads
- Scheduler
- Periodic task scheduler (optional)

These threads coordinate the receipt, execution and scheduling of various tasks. What happens when you call a decorated function in your application?

1. You call a function – huey has decorated it, which triggers a message being put into the queue. At this point your application returns. If you are using a "data store", then you will be return an *`AsyncData`* object.
2. In a separate process, the consumer will be listening for new messages – one of the worker threads will pull down the message. If your backend supports blocking, it will block until a new message is available, otherwise it will poll.

3. The worker looks at the message and checks to see if it can be run (i.e., was this message "revoked"? Is it scheduled to actually run later?). If it is revoked, the message is thrown out. If it is scheduled to run later, it gets added to the schedule. Otherwise, it is executed.
4. The worker thread executes the task. If the task finishes, any results are published to the result store (if one is configured). If the task fails and can be retried, it is either enqueued or added to the schedule (which happens if a delay is specified between retries).

While all this is going on, the Scheduler thread is continually looking at its schedule to see if any commands are ready to be executed. If a command is ready to run, it is enqueued and will be processed by the Message receiver thread.

Similarly, the Periodic task thread will run every minute to see if there are any regularly-scheduled tasks to run at that time. Those tasks will be enqueued and processed by the Message receiver thread.

When the consumer is shut-down (SIGTERM) it will save the schedule and finish any jobs that are currently being worked on.

1.4.3 Consumer Event Emitter

If you specify a `RedisEventEmitter` when setting up your *`Huey`* instance (or if you choose to use `RedisHuey`), the consumer will publish real-time events about the status of various tasks. You can subscribe to these events in your own application.

When an event is emitted, the following information is provided (serialized as JSON):

- `status`: a String indicating what type of event this is.
- `id`: the UUID of the task.
- `task`: a user-friendly name indicating what type of task this is.
- `retries`: how many retries the task has remaining.
- `retry_delay`: how long to sleep before retrying the task in event of failure.
- **`execute_time`: A unix timestamp indicating when the task is scheduled to** execute (this may be `None`).
- `error`: A boolean value indicating if there was an error.
- `traceback`: A string traceback of the error, if one occurred.

The following events are emitted by the consumer:

- `enqueued`: sent when a task is enqueued.
- **`scheduled`: sent when a task is added to the schedule for execution in** the future.
- `revoked`: sent when a task is not executed because it has been revoked.
- `started`: sent when a worker thread begins executing a task.
- **`finished`: sent when a worker thread finishes executing a task and has** stored the result.
- `error`: sent when an exception occurs while executing a task.
- `retrying`: sent when retrying a task that failed.

1.5 Understanding how tasks are imported

Behind-the-scenes when you decorate a function with *`task()`* or *`periodic_task()`*, the function registers itself with a centralized in-memory registry. When that function is called, a reference is put into the queue (among other things), and when that message is consumed the function is then looked-up in the consumer's registry. Because of the way this works, it is strongly recommended that **all decorated functions be imported when the consumer starts up**.

Note: If a task is not recognized, the consumer will throw a `QueueException`

The consumer is executed with a single required parameter – the import path to a *Huey* object. It will import the object along with anything else in the module – thus you must be sure **imports of your tasks should also occur with the import of the Huey object**.

1.5.1 Suggested organization of code

Generally, I structure things like this, which makes it very easy to avoid circular imports. If it looks familiar, that's because it is exactly the way the project is laid out in the *getting started* guide.

- `config.py`, the module containing the *Huey* object.

```
# config.py
from huey import RedisHuey

huey = RedisHuey('testing', host='localhost')
```

- `tasks.py`, the module containing any decorated functions. Imports the `huey` object from the `config.py` module:

```
# tasks.py
from config import huey

@huey.task()
def count_beans(num):
    print 'Counted %s beans' % num
```

- `main.py` / `app.py`, the "main" module. Imports both the `config.py` module **and** the `tasks.py` module.

```
# main.py
from config import huey  # import the "huey" object.
from tasks import count_beans  # import any tasks / decorated functions


if __name__ == '__main__':
    beans = raw_input('How many beans? ')
    count_beans(int(beans))
    print 'Enqueued job to count %s beans' % beans
```

To run the consumer, point it at `main.huey`, in this way everything gets imported correctly:

```
$ huey_consumer.py main.huey
```

1.6 Troubleshooting and Common Pitfalls

This document outlines some of the common pitfalls you may encounter when getting set up with huey. It is arranged in a problem/solution format.

Tasks not running First step is to increase logging verbosity by running the consumer with `--verbose`. You can also specify a logfile using the `--logfile` option.

Check for any exceptions. The most common cause of tasks not running is that they are not being loaded, in which case you will see `QueueException` "XXX not found in TaskRegistry" errors.

"QueueException: XXX not found in CommandRegistry" in log file Exception occurs when a task is called by a task producer, but is not imported by the consumer. To fix this, ensure that by loading the `Huey` object, you also import any decorated functions as well.

For more information on how tasks are imported, see the *docs*

"Error importing XXX" when starting consumer This error message occurs when the module containing the configuration specified cannot be loaded (not on the pythonpath, mistyped, etc). One quick way to check is to open up a python shell and try to import the configuration.

Example syntax: `huey_consumer.py main_module.huey`

Tasks not returning results Ensure that you have specified a `result_store` when creating your `Huey` object.

Periodic tasks are being executed multiple times per-interval If you are running multiple consumer processes, it means that more than one of them is also enqueueing periodic tasks. To fix, only run one consumer with `--periodic` and run the others with `--no-periodic`.

Scheduled tasks are not being run at the correct time Check the time on the server the consumer is running on - if different from the producer this may cause problems. By default all local times are converted to UTC when calling `.schedule()`, and the consumer runs in UTC.

1.7 Using Huey with Django

Huey comes with special integration for use with the Django framework. This is to accomodate:

1. Configuring your queue and consumer via django settings module.
2. Run the consumer as a management command.

1.7.1 Apps

`huey.djhuey` must be included in the INSTALLED_APPS within the Django settings.py file.

```
INSTALLED_APPS = (
    'huey.djhuey',
    ...
```

1.7.2 Huey Settings

Note: Huey settings are optional. If not provided, Huey will default to using Redis running locally.

All configuration is kept in `settings.HUEY`. Here are some examples:

Using redis

```
HUEY = {
    'backend': 'huey.backends.redis_backend',  # required.
    'name': 'unique name',
    'connection': {'host': 'localhost', 'port': 6379},
    'always_eager': False, # Defaults to False when running via manage.py run_huey

    # Options to pass into the consumer when running ``manage.py run_huey``
    'consumer_options': {'workers': 4},
}
```

Using sqlite.

```
HUEY = {
    'backend': 'huey.backends.sqlite_backend',  # required.
    'name': 'unique name',
    'connection': {'location': 'sqlite filename'},
    'always_eager': False, # Defaults to False when running via manage.py run_huey

    # Options to pass into the consumer when running ``manage.py run_huey``
    'consumer_options': {'workers': 4},
}
```

You can use the 'default' sqlite database by seting the filename to `DATABASE['default']['NAME']` A database file will automaticly be created using the value of `'location'`

1.7.3 Running the Consumer

To run the consumer, use the `run_huey` management command. This command will automatically import any modules in your `INSTALLED_APPS` named "tasks.py". The consumer can be configured by the `consumer_options` settings.

In addition to the `consumer_options`, you can also pass some options to the consumer at run-time.

-w, --workers Number of worker threads.

-p, --periodic Indicate that this consumer process should start a thread dedicated to enqueueing "periodic" tasks (crontab-like functionality). This defaults to `True`, so should not need to be specified in practice.

-n, --no-periodic Indicate that this consumer process should *not* enqueue periodic tasks.

For more information, check the *consumer docs*.

1.7.4 Task API

The task API is a little bit simplified for Django. The function decorators are available in the `huey.djhuey` module.

Here is how you might create two tasks:

```
from huey.djhuey import crontab, periodic_task, task

@task()
def count_beans(number):
    print '-- counted %s beans --' % number
    return 'Counted %s beans' % number

@periodic_task(crontab(minute='*/5'))
def every_five_mins():
    print 'Every five minutes this will be printed by the consumer'
```

Tasks that execute queries

If you plan on executing queries inside your task, it is a good idea to close the connection once your task finishes. To make this easier, huey provides a special decorator to use in place of `task` and `periodic_task` which will automatically close the connection for you.

```
from huey.djhuey import crontab, db_periodic_task, db_task

@db_task()
def do_some_queries():
    # This task executes queries. Once the task finishes, the connection
    # will be closed.

@db_periodic_task(crontab(minute='*/5'))
def every_five_mins():
    # This is a periodic task that executes queries.
```

1.8 Huey's API

Most end-users will interact with the API using the two decorators:

- *Huey.task()*
- *Huey.periodic_task()*

The API documentation will follow the structure of the huey API, starting with the highest-level interfaces (the decorators) and eventually discussing the lowest-level interfaces, the *BaseQueue* and *BaseDataStore* objects.

1.8.1 Function decorators and helpers

class Huey(*queue*[, *result_store=None*[, *schedule=None*[, *events=None*[, *store_none=False*[, *always_eager=False*]]]]])

Huey executes tasks by exposing function decorators that cause the function call to be enqueued for execution by the consumer.

Typically your application will only need one Huey instance, but you can have as many as you like – the only caveat is that one consumer process must be executed for each Huey instance.

Parameters

- **queue** – a queue instance, e.g. *RedisQueue*.
- **result_store** – a place to store results and the task schedule, e.g. *RedisDataStore*.
- **schedule** – scheduler implementation, e.g. an instance of *RedisSchedule*.
- **events** – event emitter implementation, e.g. an instance of `RedisEventEmitter`.
- **store_none** (*boolean*) – Flag to indicate whether tasks that return `None` should store their results in the result store.
- **always_eager** – Useful for testing, this will execute all tasks immediately, without enqueueing them.

Example usage:

```
from huey.api import Huey, crontab
from huey.backends.redis_backend import RedisBlockingQueue, RedisDataStore,\
    RedisSchedule

huey = RedisHuey('my-app')

# THIS IS EQUIVALENT TO ABOVE CODE:
#queue = RedisBlockingQueue('my-app')
```

```
    #result_store = RedisDataStore('my-app')
    #schedule = RedisSchedule('my-app')
    #huey = Huey(queue, result_store, schedule)

    @huey.task()
    def slow_function(some_arg):
        # ... do something ...
        return some_arg

    @huey.periodic_task(crontab(minute='0', hour='3'))
    def backup():
        # do a backup every day at 3am
        return
```

task([*retries=0*[, *retry_delay=0*[, *retries_as_argument=False*[, *include_task=False*]]]])

Function decorator that marks the decorated function for processing by the consumer. Calls to the decorated function will do the following:

1. Serialize the function call into a message suitable for storing in the queue
2. Enqueue the message for execution by the consumer
3. If a `result_store` has been configured, return an *AsyncData* instance which can retrieve the result of the function, or `None` if not using a result store.

Note: Huey can be configured to execute the function immediately by instantiating it with `always_eager = True` – this is useful for running in debug mode or when you do not wish to run the consumer.

Here is how you might use the `task` decorator:

```
# assume that we've created a huey object
from huey import RedisHuey

huey = RedisHuey()

@huey.task()
def count_some_beans(num):
    # do some counting!
    return 'Counted %s beans' % num
```

Now, whenever you call this function in your application, the actual processing will occur when the consumer dequeues the message and your application will continue along on its way.

Without a result store:

```
>>> res = count_some_beans(1000000)
>>> res is None
True
```

With a result store:

```
>>> res = count_some_beans(1000000)
>>> res
<huey.api.AsyncData object at 0xb7471a4c>
>>> res.get()
'Counted 1000000 beans'
```

Parameters

- **retries** (*int*) – number of times to retry the task if an exception occurs
- **retry_delay** (*int*) – number of seconds to wait between retries
- **retries_as_argument** (*boolean*) – whether the number of retries should be passed in to the decorated function as an argument.
- **include_task** (*boolean*) – whether the task instance itself should be passed in to the decorated function as the `task` argument.

Return type decorated function

The return value of any calls to the decorated function depends on whether the *Huey* instance is configured with a `result_store`. If a result store is configured, the decorated function will return an *AsyncData* object which can fetch the result of the call from the result store – otherwise it will simply return `None`.

The `task` decorator also does one other important thing – it adds a special function **onto** the decorated function, which makes it possible to *schedule* the execution for a certain time in the future:

{decorated func}.schedule(args=None, kwargs=None, eta=None, delay=None, convert_utc

Use the special `schedule` function to schedule the execution of a queue task for a given time in the future:

```
import datetime

# get a datetime object representing one hour in the future
in_an_hour = datetime.datetime.now() + datetime.timedelta(seconds=3600)

# schedule "count_some_beans" to run in an hour
count_some_beans.schedule(args=(100000,), eta=in_an_hour)

# another way of doing the same thing...
count_some_beans.schedule(args=(100000,), delay=(60 * 60))
```

Parameters

- **args** – arguments to call the decorated function with
- **kwargs** – keyword arguments to call the decorated function with
- **eta** (*datetime*) – the time at which the function should be executed
- **delay** (*int*) – number of seconds to wait before executing function
- **convert_utc** – whether the `eta` should be converted from local time to UTC, defaults to `True`

Return type like calls to the decorated function, will return an *AsyncData* object if a result store is configured, otherwise returns `None`

{decorated func}.call_local

Call the `@task`-decorated function without enqueueing the call. Or, in other words, `call_local()` provides access to the actual function.

```
>>> count_some_beans.call_local(1337)
'Counted 1337 beans'
```

{decorated func}.task_class

Store a reference to the task class for the decorated function.

```
>>> count_some_beans.task_class
tasks.queuecmd_count_beans
```

periodic_task(*validate_datetime*)

Function decorator that marks the decorated function for processing by the consumer *at a specific interval*. Calls to functions decorated with `periodic_task` will execute normally, unlike `task()`, which enqueues tasks for execution by the consumer. Rather, the `periodic_task` decorator serves to **mark a function as needing to be executed periodically** by the consumer.

Note: By default, the consumer will execute `periodic_task` functions. To disable this, run the consumer with `-n` or `--no-periodic`.

The `validate_datetime` parameter is a function which accepts a datetime object and returns a boolean value whether or not the decorated function should execute at that time or not. The consumer will send a datetime to the function every minute, giving it the same granularity as the linux crontab, which it was designed to mimic.

For simplicity, there is a special function `crontab()`, which can be used to quickly specify intervals at which a function should execute. It is described below.

Here is an example of how you might use the `periodic_task` decorator and the `crontab` helper:

```
from huey import crontab
from huey import RedisHuey

huey = RedisHuey()

@huey.periodic_task(crontab(minute='*/5'))
def every_five_minutes():
    # this function gets executed every 5 minutes by the consumer
    print "It's been five minutes"
```

Note: Because functions decorated with `periodic_task` are meant to be executed at intervals in isolation, they should not take any required parameters nor should they be expected to return a meaningful value. This is the same regardless of whether or not you are using a result store.

Parameters **validate_datetime** – a callable which takes a `datetime` and returns a boolean whether the decorated function should execute at that time or not

Return type decorated function

Like `task()`, the periodic task decorator adds several helpers to the decorated function. These helpers allow you to "revoke" and "restore" the periodic task, effectively enabling you to pause it or prevent its execution.

{decorated_func}.revoke([revoke_until=None[, revoke_once=False]])

Prevent the given periodic task from executing. When no parameters are provided the function will not execute again.

This function can be called multiple times, but each call will overwrite the limitations of the previous.

Parameters

- **revoke_until** (*datetime*) – Prevent the execution of the task until the given datetime. If `None` it will prevent execution indefinitely.
- **revoke_once** (*bool*) – If `True` will only prevent execution the next time it would normally execute.

```
# skip the next execution
every_five_minutes.revoke(revoke_once=True)

# pause the command indefinitely
```

```
            every_five_minutes.revoke()

            # pause the command for 24 hours
            every_five_minutes.revoke(datetime.datetime.now() + datetime.timedelta(days=1))
```

{decorated_func}.is_revoked([dt=None])

Check whether the given periodic task is revoked. If `dt` is specified, it will check if the task is revoked for the given datetime.

Parameters **dt** (*datetime*) – If provided, checks whether task is revoked at the given datetime

{decorated_func}.restore()

Clears any revoked status and run the task normally

If you want access to the underlying task class, it is stored as an attribute on the decorated function:

{decorated_func}.task_class

Store a reference to the task class for the decorated function.

crontab(*month='*', day='*', day_of_week='*', hour='*', minute='*'*)

Convert a "crontab"-style set of parameters into a test function that will return `True` when a given `datetime` matches the parameters set forth in the crontab.

Acceptable inputs:

- "*" = every distinct value
- "*/n" = run every "n" times, i.e. hours='*/4' == 0, 4, 8, 12, 16, 20
- "m-n" = run every time m..n
- "m,n" = run on m and n

Return type a test function that takes a `datetime` and returns a boolean

1.8.2 AsyncData

class AsyncData(*huey, task*)

Although you will probably never instantiate an `AsyncData` object yourself, they are returned by any calls to *`task()`* decorated functions (provided that "huey" is configured with a result store). The `AsyncData` talks to the result store and is responsible for fetching results from tasks. Once the consumer finishes executing a task, the return value is placed in the result store, allowing the producer to retrieve it.

Working with the `AsyncData` class is very simple:

```
>>> from main import count_some_beans
>>> res = count_some_beans(100)
>>> res  # what is "res" ?
<huey.queue.AsyncData object at 0xb7471a4c>

>>> res.get()  # get the result of this task, assuming it executed
'Counted 100 beans'
```

What happens when data isn't available yet? Let's assume the next call takes about a minute to calculate:

```
>>> res = count_some_beans(10000000) # let's pretend this is slow
>>> res.get()  # data is not ready, so returns None

>>> res.get() is None  # data still not ready
True
```

```
>>> res.get(blocking=True, timeout=5)  # block for 5 seconds
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/charles/tmp/huey/src/huey/huey/queue.py", line 46, in get
    raise DataStoreTimeout
huey.exceptions.DataStoreTimeout

>>> res.get(blocking=True)  # no timeout, will block until it gets data
'Counted 10000000 beans'
```

get([*blocking=False*[, *timeout=None*[, *backoff=1.15*[, *max_delay=1.0*[, *revoke_on_timeout=False*]]]]])

Attempt to retrieve the return value of a task. By default, it will simply ask for the value, returning `None` if it is not ready yet. If you want to wait for a value, you can specify `blocking = True` – this will loop, backing off up to the provided `max_delay` until the value is ready or until the `timeout` is reached. If the `timeout` is reached before the result is ready, a `DataStoreTimeout` exception will be raised.

Parameters

- **blocking** – boolean, whether to block while waiting for task result
- **timeout** – number of seconds to block for (used with *blocking=True*)
- **backoff** – amount to backoff delay each time no result is found
- **max_delay** – maximum amount of time to wait between iterations when attempting to fetch result.
- **revoke_on_timeout** (*bool*) – if a timeout occurs, revoke the task

revoke()

Revoke the given task. Unless it is in the process of executing, it will be revoked and the task will not run.

```
in_an_hour = datetime.datetime.now() + datetime.timedelta(seconds=3600)

# run this command in an hour
res = count_some_beans.schedule(args=(100000,), eta=in_an_hour)

# oh shoot, I changed my mind, do not run it after all
res.revoke()
```

restore()

Restore the given task. Unless it has already been skipped over, it will be restored and run as scheduled.

1.8.3 Queues and DataStores

Huey communicates with two types of data stores – queues and datastores. Thinking of them as python datatypes, a queue is sort of like a `list` and a datastore is sort of like a `dict`. Queues are FIFOs that store tasks – producers put tasks in on one end and the consumer reads and executes tasks from the other. DataStores are key-based stores that can store arbitrary results of tasks keyed by task id. DataStores can also be used to serialize task schedules so in the event your consumer goes down you can bring it back up and not lose any tasks that had been scheduled.

Huey, like just about a zillion other projects, uses a "pluggable backend" approach, where the interface is defined on a couple classes *BaseQueue* and *BaseDataStore*, and you can write an implementation for any datastore you like. The project ships with backends that talk to redis, a fast key-based datastore, but the sky's the limit when it comes to what you want to interface with. Below is an outline of the methods that must be implemented on each class.

Base classes

class BaseQueue(*name*, ***connection*)
Queue implementation – any connections that must be made should be created when instantiating this class.

Parameters

- **name** – A string representation of the name for this queue
- **connection** – Connection parameters for the queue

blocking = False
Whether the backend blocks when waiting for new results. If set to `False`, the backend will be polled at intervals, if `True` it will read and wait.

write(*data*)
Write data to the queue - has no return value.

Parameters data – a string

read()
Read data from the queue, returning None if no data is available – an empty queue should not raise an Exception!

Return type a string message or `None` if no data is present

remove(*data*)
Remove all instances of given data from queue, returning number removed

Parameters data (*string*) –

Return type number of instances removed

flush()
Optional: Delete everything in the queue – used by tests

__len__()
Optional: Return the number of items in the queue – used by tests

class BaseDataStore(*name*, ***connection*)
Data store implementation – any connections that must be made should be created when instantiating this class.

Parameters

- **name** – A string representation of the name for this data store
- **connection** – Connection parameters for the data store

put(*key*, *value*)
Store the `value` using the `key` as the identifier

peek(*key*)
Retrieve the value stored at the given `key`, returns a special value `EmptyData` if nothing exists at the given key.

get(*key*)
Retrieve the value stored at the given `key`, returns a special value `EmptyData` if no data exists at the given key. This is to differentiate between "no data" and a stored `None` value.

Warning: After a result is fetched it will be removed from the store!

flush()
Remove all keys

class **BaseSchedule**(*name*, ***connection*)
Schedule tasks, should be able to efficiently find tasks that are ready for execution.

add(*data*, *timestamp*)
Add the timestamped data (a serialized task) to the task schedule.

read(*timestamp*)
Return all tasks that are ready for execution at the given timestamp.

flush()
Remove all tasks from the schedule.

class **BaseEventEmitter**(*channel*, ***connection*)
A send-and-forget event emitter that is used for sending real-time updates for tasks in the consumer.

emit(*data*)
Send the data on the specified channel.

Redis implementation

All the following use the python redis driver written by Andy McCurdy.

class **RedisQueue**(*name*, ***connection*)
Does a simple `RPOP` to pull messages from the queue, meaning that it polls.

Parameters

- **name** – the name of the queue to use
- **connection** – a list of values passed directly into the `redis.Redis` class

class **RedisBlockingQueue**(*name*, *read_timeout=None*, ***connection*)
Does a `BRPOP` to pull messages from the queue, meaning that it blocks on reads. By default Huey will block forever waiting for a message, but if you want, you can specify a timeout in seconds. This may prevent the consumer from getting hung waiting on tasks in the event of network disruptions or similar quirks.

Parameters

- **name** – the name of the queue to use
- **read_timeout** (*int*) – limit blocking pop to `read_timeout` seconds.
- **connection** – a list of values passed directly into the `redis.Redis` class

class **RedisDataStore**(*name*, ***connection*)
Stores results in a redis hash using `HSET`, `HGET` and `HDEL`

Parameters

- **name** – the name of the data store to use
- **connection** – a list of values passed directly into the `redis.Redis` class

class **RedisSchedule**(*name*, ***connection*)

Uses sorted sets to efficiently manage a schedule of timestamped tasks.

param name the name of the data store to use

param connection a list of values passed directly into the `redis.Redis` class

class **RedisEventEmitter**(*channel*, ***connection*)
Uses Redis pubsub to emit json-serialized updates about tasks in real-time.

Parameters

- **`channel`** – the channel to send messages on.
- **`connection`** – values passed directly to the `redis.Redis` class.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search