huey Documentation

Release 1.10.5

charles leifer

Dec 19, 2018

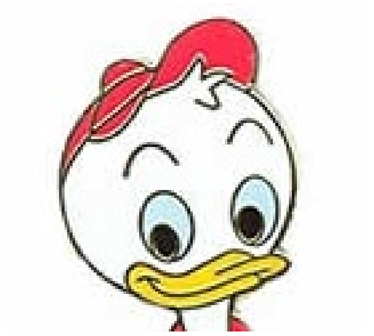
Contents

a lightweight alternative, huey is:

- written in python (2.7+, 3.4+)
- only dependency is the Python Redis client
- clean and simple APIs

huey supports:

- multi-process, multi-thread or greenlet task execution models
- schedule tasks to execute at a given time, or after a given delay
- schedule recurring tasks, like a crontab
- automatically retry tasks that fail
- task result storage
- task locking
- task pipelines and chains
- consumer publishes event stream, allowing high-fidelity monitoring



CHAPTER 1

At a glance

Use the *task()* and *periodic_task()* decorators to turn functions into tasks that will be run by the consumer:

```
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()
```

Here's how to run the consumer with four worker processes (good setup for CPU-intensive processing):

```
$ huey_consumer.py my_app.huey -k process -w 4
```

If your work-loads are mostly IO-bound, you can run the consumer with threads or greenlets instead. Because greenlets are so lightweight, you can run quite a few of them efficiently:

```
$ huey_consumer.py my_app.huey -k greenlet -w 32
```

CHAPTER 2

Redis

Huey's design and feature-set are, to a large extent, informed by the capabilities of the Redis database. Redis is a fantastic fit for a lightweight task queueing library like Huey: it's self-contained, versatile, and can be a multi-purpose solution for other web-application tasks like caching, event publishing, analytics, rate-limiting, and more.

Although Huey was designed with Redis in mind, the storage system implements a simple API and many other tools could be used instead of Redis if that's your preference. Huey ships with an alternative storage implementation that uses sqlite.

CHAPTER 3

Table of contents

3.1 Installing

huey can be installed from PyPI using pip.

```
$ pip install huey
```

If you want to enable SQLite and Redis backend automatically, use following command:

```
$ pip install huey[backends]
```

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

```
$ pip install redis
```

If your tasks are IO-bound rather than CPU-bound, you might consider using the `greenlet` worker type. To use the greenlet workers, you need to install `gevent`:

```
pip install gevent
```

3.1.1 Using git

If you want to run the very latest, you can clone the source repo and install the library:

```
$ 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
```

The source code is available online at https://github.com/coleifer/huey

3.2 Getting Started

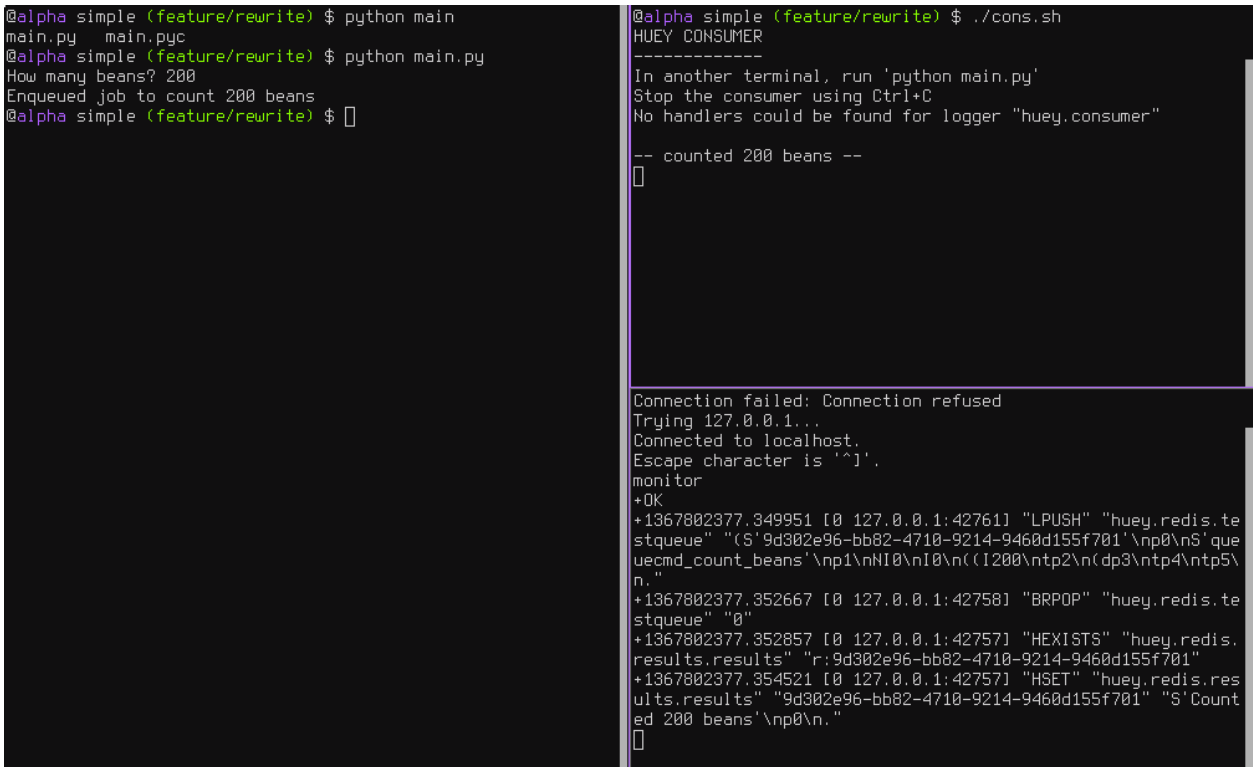
The goal of this document is to help you get running quickly and with as little fuss as possible.

3.2.1 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.

```
# config.py
from huey import RedisHuey

huey = RedisHuey()
```

The huey object encapsulates a queue. The queue is responsible for storing and retrieving messages, and the huey instance is used by your application code to coordinate function calls with a queue backend. We'll see how the huey object is used when looking at the actual function responsible for counting beans:

```
# 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. In a separate process, the consumer will see the new message and run the function.

Our 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, just return a value in your task function.

Note: If you are storing results but are not using them, that can waste significant space, especially if your task volume is high. To disable result storage, you can either return `None` or specify `result_store=False` when initializing your `Huey` instance.

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

```python
from config import huey


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

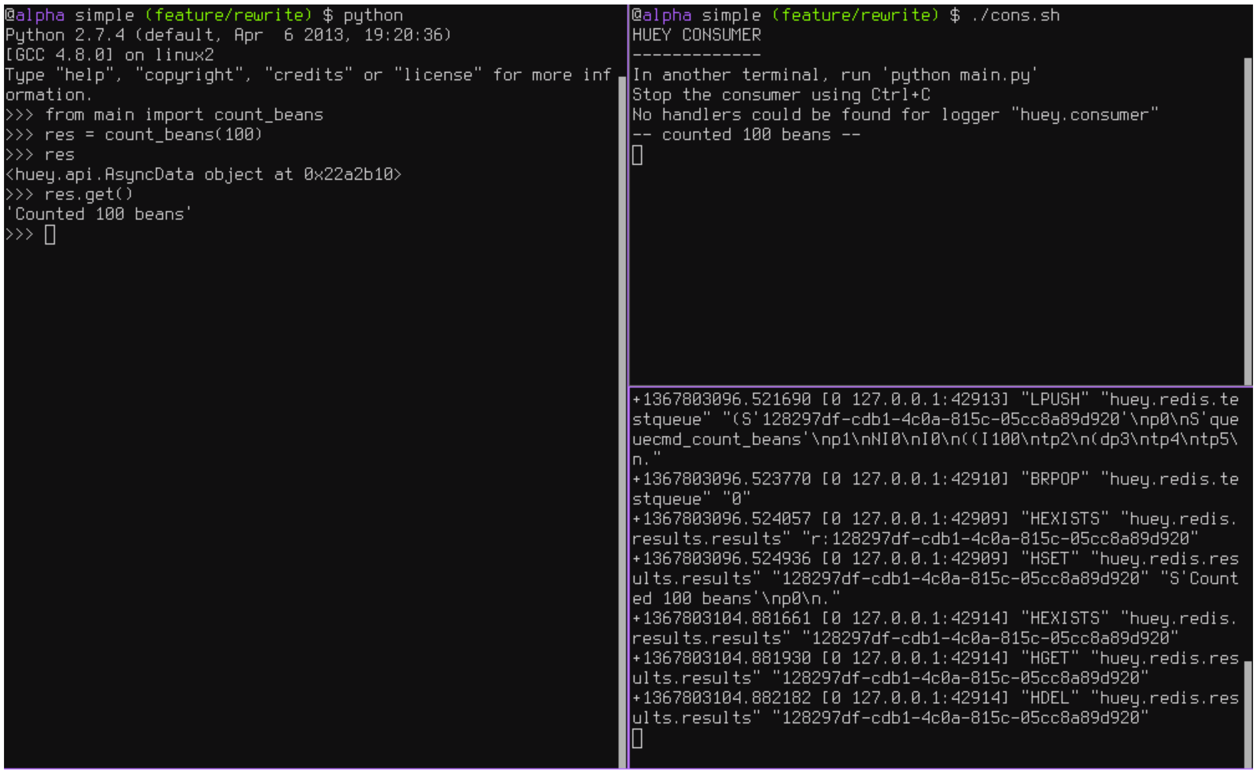
We're ready to fire up the consumer. Instead of simply executing the main program, though, we'll start an interpreter and run the following:

```
>>> from main import count_beans
>>> res = count_beans(100)
>>> print(res)                         # What is "res" ?
<huey.api.TaskResultWrapper object at 0xb7471a4c>

>>> res()                             # 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)
>>> print(res)
<huey.api.TaskResultWrapper object at 0xb72915ec>

>>> res()  # This returns None, no data is ready.

>>> res()  # A couple seconds later.

>>> res(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)
```

Note: By default, the Huey consumer runs in UTC-mode. The effect of this on scheduled tasks is that when using naive datetimes, they must be with respect to `datetime.utcnow()`.

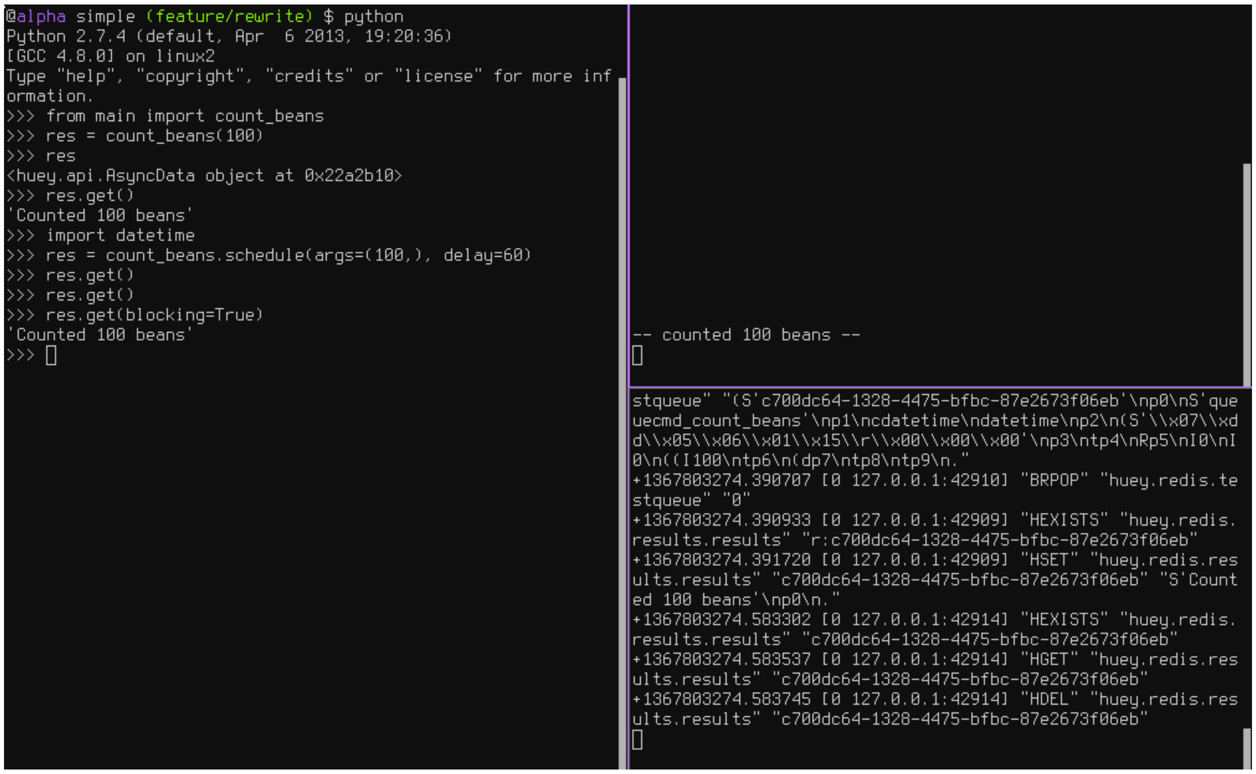
The reason we aren't using `utcnow()` in the example above is because the `schedule()` method takes a 3rd parameter, `convert_utc`, which defaults to `True`. So in the above code, the datetime is converted from localtime to UTC before being sent to the queue.

If you are running the consumer in localtime-mode (`-o`), then you should **always** specify `convert_utc=False` with `.schedule()`, including when you are specifying a `delay`.

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.

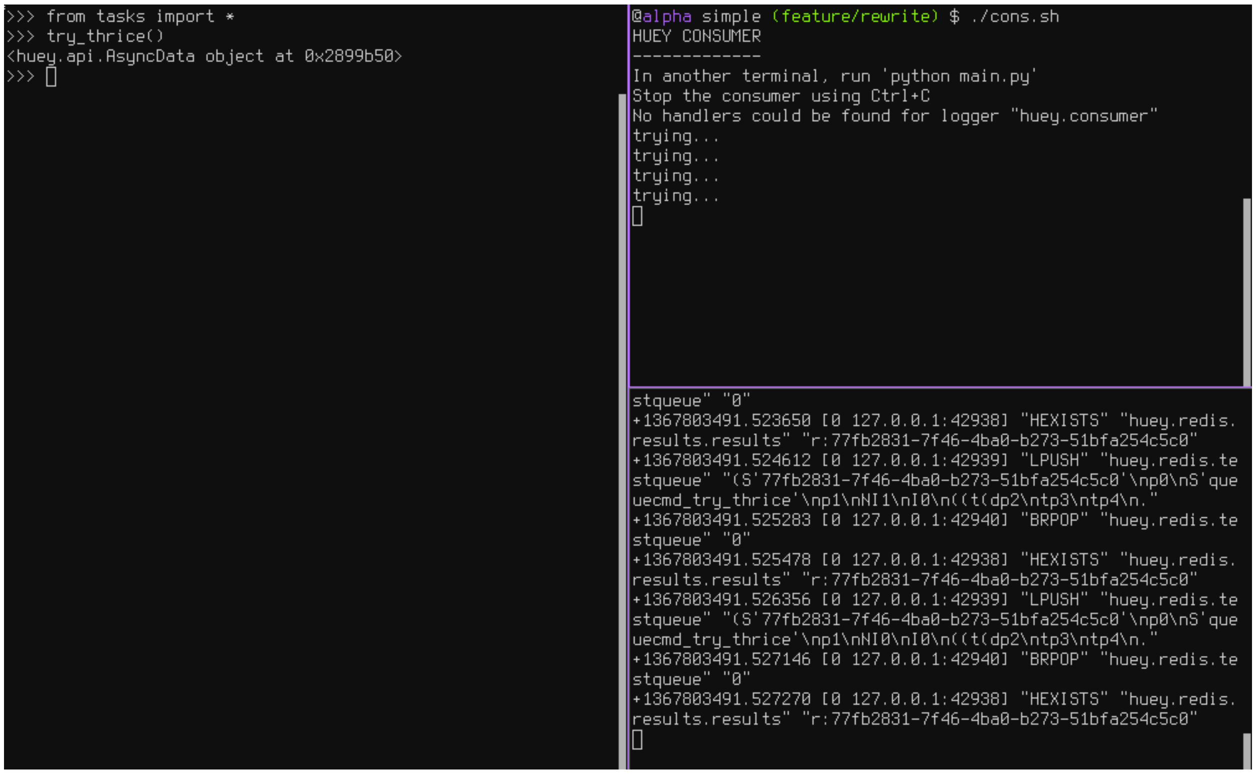
Here is a task that will be retried 3 times and will blow up every time:

```python
# 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"
```

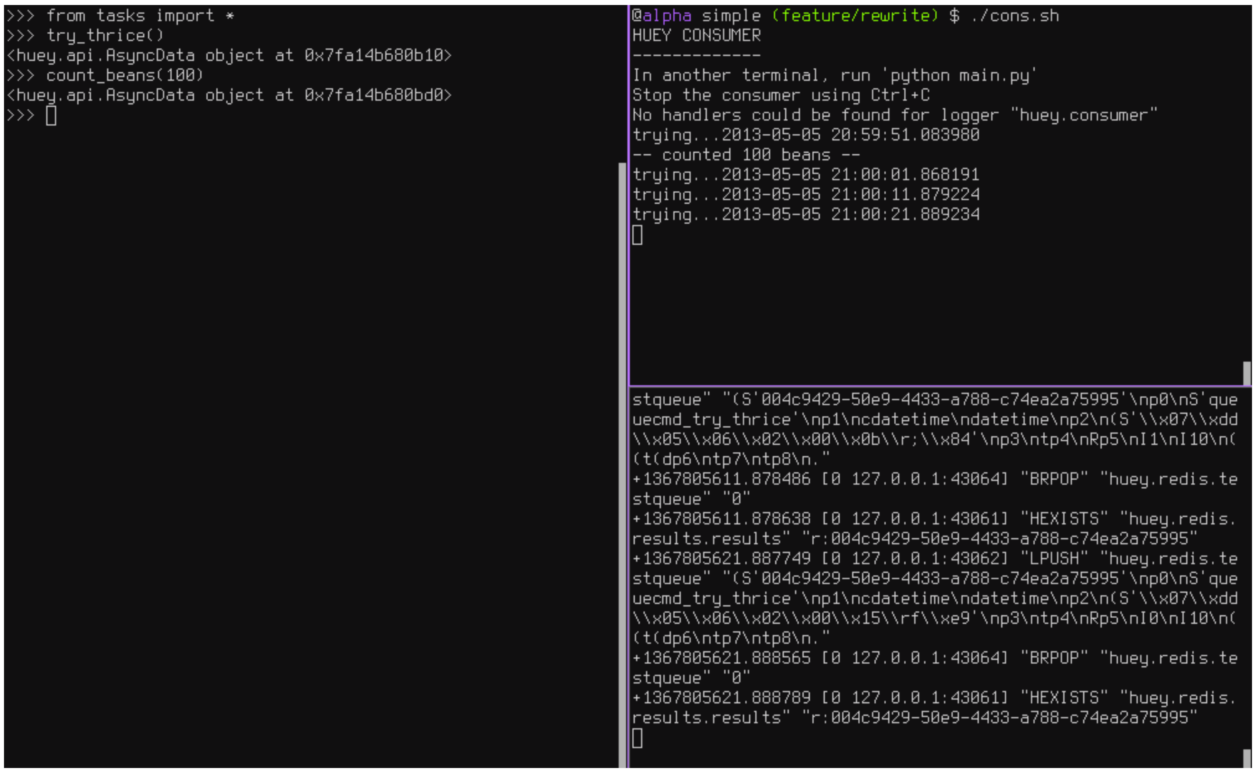
Oftentimes it is a good idea to wait a certain amount of time between retries. You can specify a *delay* between retries, in seconds, which is the minimum time before the task will be retried. Here we've modified the command to include a delay, and also to print the current time to show that its working.

```python
# 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.

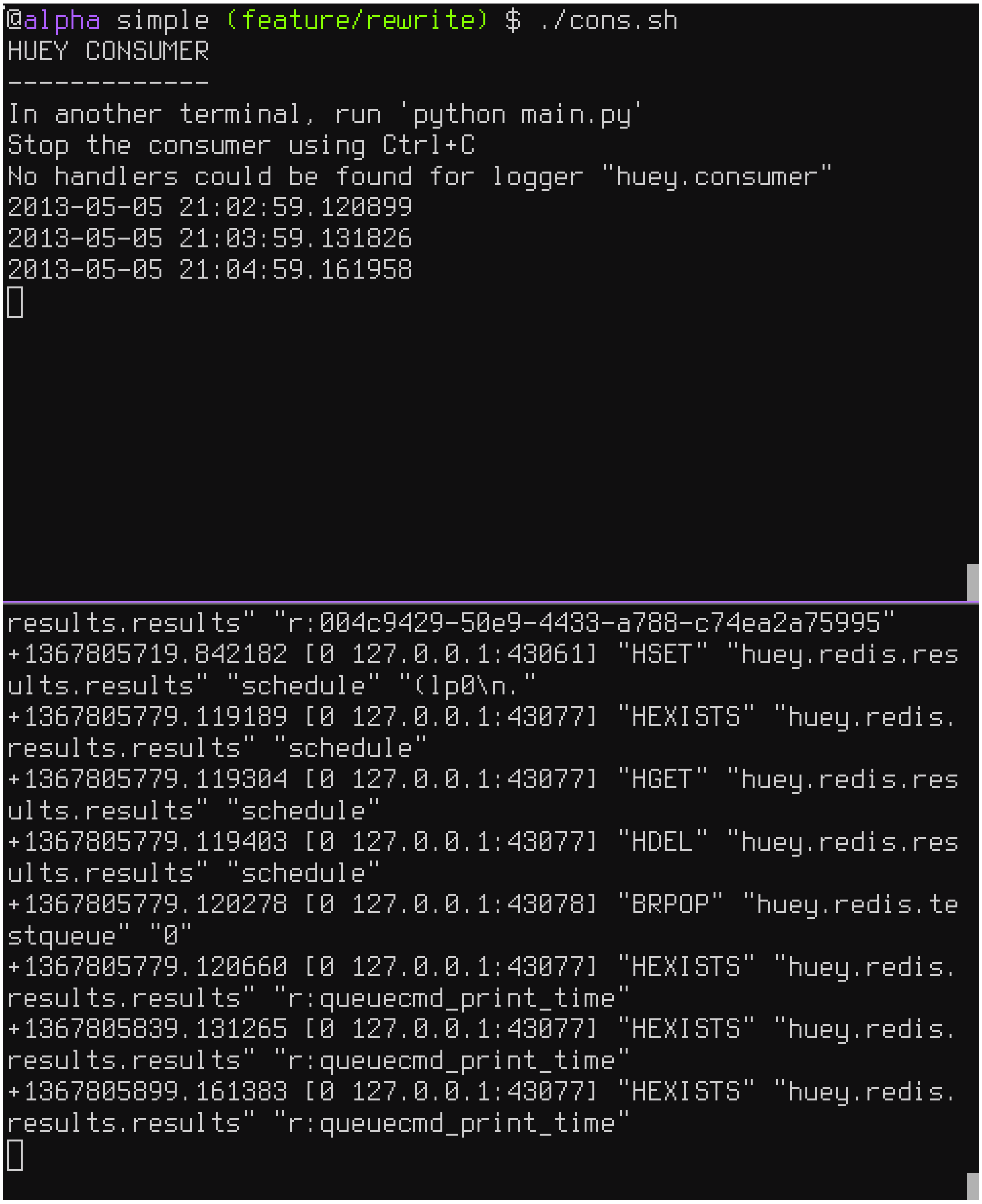
Let's add a new task that prints the time every minute – we'll use this to test that the consumer is executing the tasks on schedule.

```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"
```

Canceling or pausing tasks

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.

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 TaskResultWrapper 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()
```

To revoke all instances of a given task, use the `revoke()` and `restore()` methods on the task itself:

```
count_beans.revoke()
assert count_beans.is_revoked() is True

res = count_beans(100)
assert res.is_revoked() is True

count_beans.restore()
assert count_beans.is_revoked() is False
```

Canceling or pausing periodic tasks

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.periodic_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: When specifying the `revoke_until` setting, naive datetimes should be with respect to `datetime.utcnow()` if the consumer is running in UTC-mode (the default). Use `datetime.now()` if the consumers is running in localtime-mode (`-o`).

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()
assert print_time.is_revoked() is True
```

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

```
print_time.restore()
```

Task Pipelines

Huey supports pipelines (or chains) of one or more tasks that should be executed sequentially.

To get started with pipelines, let's first look behind-the-scenes at what happens when you invoke a `task`-decorated function:

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

result = add(1, 2)

# Is equivalent to:
task = add.s(1, 2)
result = huey.enqueue(task)
```

The *`TaskWrapper.s()`* method is used to create a *`QueueTask`* instance which represents the execution of the given function. The `QueueTask` is serialized and enqueued, then dequeued, deserialized and executed by the consumer.

To create a pipeline, we will use the *`TaskWrapper.s()`* method to create a *`QueueTask`* instance. We can then chain additional tasks using the *`QueueTask.then()`* method:

```
add_task = add.s(1, 2)  # Create QueueTask to represent task invocation.

# Add additional tasks to pipeline by calling QueueTask.then().
pipeline = (add_task
            .then(add, 3)  # Call add() with previous result and 3.
            .then(add, 4)  # etc...
            .then(add, 5))

results = huey.enqueue(pipeline)

# Print results of above pipeline.
print([result.get(blocking=True) for result in results])

# [3, 6, 10, 15]
```

When enqueueing a task pipeline, the return value will be a list of *`TaskResultWrapper`* objects, one for each task in the pipeline.

Note that the return value from the parent task is passed to the child task, and so-on.

If the value returned by the parent function is a `tuple`, then the tuple will be used to update the `*args` for the child function. Likewise, if the parent function returns a `dict`, then the dict will be used to update the `**kwargs` for the child function.

Example of chaining fibonacci calculations:

```
@huey.task()
def fib(a, b=1):
    a, b = a + b, a
    return (a, b)  # returns tuple, which is passed as *args

pipe = (fib.s(1)
        .then(fib)
        .then(fib))
results = huey.enqueue(pipe)

print([result.get(blocking=True) for result in results])
# [(2, 1), (3, 2), (5, 3)]
```

Here is an example of returning a dictionary to be passed in as keyword-arguments to the child function:

```
@huey.task()
def stateful(v1=None, v2=None, v3=None):
    state = {
        'v1': v1 + 1 if v1 is not None else 0,
        'v2': v2 + 2 if v2 is not None else 0,
        'v3': v3 + 3 if v3 is not None else 0}
    return state

pipe = (stateful
        .s()
        .then(stateful)
        .then(stateful))

results = huey.enqueue(pipe)
print([result.get(True) for result in results])

# Prints:
# [{'v1': 0, 'v2': 0, 'v3': 0},
#  {'v1': 1, 'v2': 2, 'v3': 3},
#  {'v1': 2, 'v2': 4, 'v3': 6}]
```

For more information, see the documentation on *`TaskWrapper.s()`* and *`QueueTask.then()`*.

Locking tasks

Task locking can be accomplished using the *`Huey.lock_task()`* method, which acts can be used as a context-manager or decorator.

This lock is designed to be used to prevent multiple invocations of a task from running concurrently. If using the lock as a decorator, place it directly above the function declaration.

If a second invocation occurs and the lock cannot be acquired, then a special exception is raised, which is handled by the consumer. The task will not be executed and an `EVENT_LOCKED` will be emitted. If the task is configured to be retried, then it will be retried normally, but the failure to acquire the lock is not considered an error.

Examples:

```python
@huey.periodic_task(crontab(minute='*/5'))
@huey.lock_task('reports-lock')
def generate_report():
    # If a report takes longer than 5 minutes to generate, we do
    # not want to kick off another until the previous invocation
    # has finished.
    run_report()


@huey.periodic_task(crontab(minute='0'))
def backup():
    # Generate backup of code
    do_code_backup()

    # Generate database backup. Since this may take longer than an
    # hour, we want to ensure that it is not run concurrently.
    with huey.lock_task('db-backup'):
        do_db_backup()
```

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
- *Huey.task()* - decorator to indicate an executable task
- *Huey.periodic_task()* - decorator to indicate a task that executes at periodic intervals
- *TaskResultWrapper.get()* - get the return value from a task
- *crontab()* - a function for defining what intervals to execute a periodic command

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

3.3 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.

3.3.1 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 the logfile will grow indefinitely, so you may wish to configure a tool like `logrotate`.

Alternatively, you can attach your own handler to `huey.consumer`.

The default loglevel is `INFO`.

-v, --verbose Verbose logging (loglevel=DEBUG). If no logfile is specified and verbose is set, then the consumer will log to the console.

Note: due to conflicts, when using Django this option is renamed to use `-V`, `--huey-verbose`.

-q, --quiet Minimal logging, only errors and their tracebacks will be logged.

-w, --workers Number of worker threads/processes/greenlets, the default is `1` but some applications may want to increase this number for greater throughput. Even if you have a small workload, you will typically want to increase this number to at least 2 just in case one worker gets tied up on a slow task. If you have a CPU-intensive workload, you may want to increase the number of workers to the number of CPU cores (or 2x CPU cores). Lastly, if you are using the `greenlet` worker type, you can easily run tens or hundreds of workers as they are extremely lightweight.

-k, --worker-type Choose the worker type, `thread`, `process` or `greenlet`. The default is `thread`.

Depending on your workload, one worker type may perform better than the others:

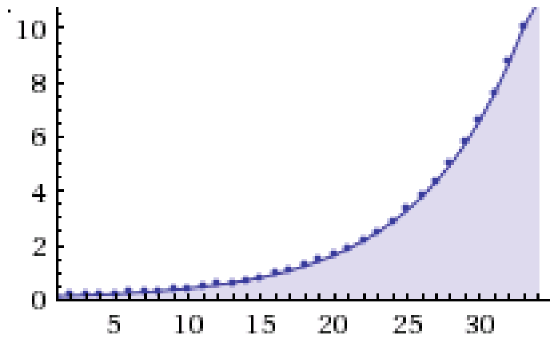
- CPU heavy loads: use "process". Python's global interpreter lock prevents multiple threads from running simultaneously, so to leverage multiple CPU cores (and reduce thread contention) run each worker as a separate process.
- IO heavy loads: use "greenlet". For example, tasks that crawl websites or which spend a lot of time waiting to read/write to a socket, will get a huge boost from using the greenlet worker model. Because greenlets are so cheap in terms of memory, you can easily run tens or hundreds of them.
- Anything else: use "thread". You get the benefits of pre-emptive multi-tasking without the overhead of multiple processes. A safe choice and the default.

-n, --no-periodic Indicate that this consumer process should *not* enqueue periodic tasks. If you do not plan on using the periodic task feature, feel free to use this option to save a few CPU cycles.

-d, --delay When using a "polling"-type queue backend, the amount of time to wait between polling the backend. Default is 0.1 seconds. For example, when the consumer starts up it will begin polling every 0.1 seconds. If no tasks are found in the queue, it will multiply the current delay (0.1) by the backoff parameter. When a task is received, the polling interval will reset back to this value.

-m, --max-delay The maximum amount of time to wait between polling, if using weighted backoff. Default is 10 seconds. If your huey consumer doesn't see a lot of action, you can increase this number to reduce CPU usage and Redis traffic.

-b, --backoff The amount to back-off when polling for results. Must be greater than one. Default is 1.15. This parameter controls the rate at which the interval increases after successive attempts return no tasks. Here is how the defaults, 0.1 initial and 1.15 backoff, look:



-c, --health-check-interval This parameter specifies how often huey should check on the status of the workers, restarting any that died for some reason. I personally run a dozen or so huey consumers at any given time and have never encountered an issue with the workers, but I suppose anything's possible and better safe than sorry.

-C, --disable-health-check This option **disables** the worker health checks. Until version 1.3.0, huey had no concept of a "worker health check" because in my experience the workers simply always stayed up and responsive. But if you are using huey for critical tasks, you may want the insurance of having additional monitoring to make sure your workers stay up and running. At any rate, I feel comfortable saying that it's perfectly fine to use this option and disable worker health checks.

-s, --scheduler-interval The frequency with which the scheduler should run. By default this will run every second, but you can increase the interval to as much as 60 seconds.

-u, --utc Indicates that the consumer should use UTC time for crontabs. Default is True, so it is not actually necessary to use this option.

-o, --localtime Indicates that the consumer should use localtime for crontabs. The default behavior is to use UTC everywhere.

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 1.05 -m 1.0
```

Running single-threaded with periodict task support disabled. Additionally, logging records are written to stdout.

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

Using multi-processing to run 4 worker processes.

```
huey_consumer.py my.app.huey -w 4 -k process
```

Using greenlets to run 100 workers, with no health checking and a scheduler granularity of 60 seconds.

```
huey_consumer.py my.app.huey -w 100 -k greenlet -C -s 60
```

3.3.2 Consumer shutdown

The huey consumer supports graceful shutdown via `SIGINT`. When the consumer process receives `SIGINT`, workers are allowed to finish up whatever task they are currently executing.

Alternatively, you can shutdown the consumer using `SIGTERM` and any running tasks will be interrupted, ensuring the process exits quickly.

3.3.3 Consumer restart

To cleanly restart the consumer, including all workers, send the `SIGHUP` signal. When the consumer receives the hang-up signal, any tasks being executed will be allowed to finish before the restart occurs.

Note: If you are using Python 2.7 and either the thread or greenlet worker model, it is strongly recommended that you use a process manager (such as systemd or supervisor) to handle running and restarting the consumer. The reason has to do with the potential of Python 2.7, when mixed with threaded/greenlet workers, to leak file descriptors. For more information, check out issue 374 and PEP 446.

3.3.4 Consumer Internals

This section will attempt to explain what happens when you call a `task`-decorated function in your application. To do this, we will go into the implementation of the consumer. The code for the consumer itself is actually quite short (couple hundred lines), and I encourage you to check it out.

The consumer is composed of three components: a master process, the scheduler, and the worker(s). Depending on the worker type chosen, the scheduler and workers will be run in their threads, processes or greenlets.

These three components coordinate the receipt, scheduling, and execution of your tasks, respectively.

1. You call a function – huey has decorated it, which triggers a message being put into the queue (Redis by default). At this point your application returns immediately, returning a *`TaskResultWrapper`* object.
2. In the consumer process, the worker(s) will be listening for new messages and one of the workers will receive your message indicating which task to run, when to run it, and with what parameters.
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 (provided you have not disabled the result store). If the task fails, the consumer checks to see if the task can be retried. Then, if the task is to be retried, the consumer checks to see if the task is configured to wait a number of seconds before retrying. Depending on the configuration, huey will either re-enqueue the task for execution, or tell the scheduler when to re-enqueue it based on the delay.

While all the above is going on with the Worker(s), the Scheduler is looking at its schedule to see if any tasks are ready to be executed. If a task is ready to run, it is enqueued and will be processed by the next available worker.

If you are using the Periodic Task feature (cron), then every minute, the scheduler will check through the various periodic tasks to see if any should be run. If so, these tasks are enqueued.

Warning: SIGINT is used to perform a graceful shutdown.

When the consumer is shutdown using SIGTERM, any workers still involved in the execution of a task will be interrupted mid-task.

3.3.5 Events

As the consumer processes tasks, it can be configured to emit events. For information on consumer-sent events, check out the *Consumer Events* documentation.

3.4 Consumer Events

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 always provided:

- `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`).

If an error occurred, then the following data is also provided:

- `error`: A boolean value indicating if there was an error.
- `traceback`: A string traceback of the error, if one occurred.

When an event includes other keys, those will be noted below.

The following events are emitted by the consumer. I've listed the event name, and in parentheses the process that emits the event and any non-standard metadata it includes.

- `EVENT_CHECKING_PERIODIC` (Scheduler, `timestamp`): emitted every minute when the scheduler checks for periodic tasks to execute.
- `EVENT_FINISHED` (Worker, `duration`): emitted when a task executes successfully and cleanly returns.
- `EVENT_RETRYING` (Worker): emitted after a task failure, when the task will be retried.
- `EVENT_REVOKED` (Worker, `timestamp`): emitted when a task is pulled from the queue but is not executed due to having been revoked.
- `EVENT_LOCKED` (Worker, `duration`): emitted when a task could not be executed because a lock was unable to be acquired.
- `EVENT_SCHEDULED` (Worker): emitted when a task specifies a delay or ETA and is not yet ready to run. This can also occur when a task is being retried and specifies a retry delay. The task is added to the schedule for later execution.
- `EVENT_SCHEDULING_PERIODIC` (Schedule, `timestamp`): emitted when a periodic task is scheduled for execution.
- `EVENT_STARTED` (Worker, `timestamp`): emitted when a worker begins executing a task.

Error events:

- `EVENT_ERROR_DEQUEUEING` (Worker): emitted if an error occurs reading from the backend queue.
- `EVENT_ERROR_ENQUEUEING` (Schedule, Worker): emitted if an error occurs enqueueing a task for execution. This can occur when the scheduler attempts to enqueue a task that had been delayed, or when a worker attempts to retry a task after an error.
- `EVENT_ERROR_INTERNAL` (Worker): emitted if an unspecified error occurs. An example might be the Redis server being offline.
- `EVENT_ERROR_SCHEDULING` (Worker): emitted when an exception occurs enqueueing a task.

- `EVENT_ERROR_STORING_RESULT` (Worker, `duration`): emitted when an exception occurs attempting to store the result of a task. In this case the task ran to completion, but the result could not be stored.
- `EVENT_ERROR_TASK` (Worker, `duration`): emitted when an unspecified error occurs in the user's task code.

3.4.1 Listening to events

The easiest way to listen for events is by iterating over the `huey.storage` object.

```
from huey.consumer import EVENT_FINISHED

for event in huey.storage:
    # Do something with the result of the task.
    if event['status'] == EVENT_FINISHED:
        result = huey.result(event['id'])
        process_result(event, result)
```

You can also achieve the same result with a simple loop like this:

```
pubsub = huey.storage.listener()
for message in pubsub.listen():
    event = message['data']  # Actual event data is stored in 'data' key.
    # Do something with `event` object.
    process_event(event)
```

3.4.2 Ordering of events

For the execution of a simple task, the events emitted would be:

- `EVENT_STARTED`
- `EVENT_FINISHED`

If a task was scheduled to be executed in the future, the events would be:

- `EVENT_SCHEDULED`
- `EVENT_STARTED`
- `EVENT_FINISHED`

If an error occurs and the task is configured to be retried, the events would be:

- `EVENT_STARTED`
- `EVENT_ERROR_TASK` (includes traceback)
- `EVENT_RETRYING`
- `EVENT_SCHEDULED` (if there is a retry delay, it will go onto the schedule)
- `EVENT_STARTED`
- `EVENT_FINISHED` if task succeeds, otherwise go back to `EVENT_ERROR_TASK`.

3.5 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 (along with the arguments the function was called with, etc), 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**.

3.5.1 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, both the `huey` instance **and** the task functions are imported in a centralized location.

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

3.6 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 TaskRegistry" 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 not accidentally specified `result_store=False` when instantiating your *`Huey`* object.

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 itself runs in UTC.

Cronjobs are not being run The consumer and scheduler run in UTC by default. To run the consumer using the server's localtime, specify `--localtime` or `-o` when running the consumer.

Greenlet workers seem stuck If you wish to use the Greenlet worker type, you need to be sure to monkey-patch in your application's entrypoint. At the top of your `main` module, you can add the following code: `from gevent import monkey; monkey.patch_all()`. Furthermore, if your tasks are CPU-bound, `gevent` can appear to lock up because it only supports cooperative multi-tasking (as opposed to pre-emptive multi-tasking when using threads). For Django, it is necessary to apply the patch inside the `manage.py` script. See the Django docs section for the code.

Testing projects using Huey If you don't have, or want, a redis server for running tests you can set `always_eager` to `True` at your Huey settings.

3.7 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.

3.7.1 Function decorators and helpers

class Huey(*name*[, *result_store=True*[, *events=True*[, *store_none=False*[, *always_eager=False*[, *store_errors=True*[, *blocking=False*[, ***storage_kwargs*]]]]]]])

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

- **name** – the name of the huey instance or application.
- **result_store** (*bool*) – whether the results of tasks should be stored.
- **events** (*bool*) – whether events should be emitted by the consumer.
- **store_none** (*bool*) – Flag to indicate whether tasks that return `None` should store their results in the result store.
- **always_eager** (*bool*) – Useful for testing, this will execute all tasks immediately, without enqueueing them.
- **store_errors** (*bool*) – whether task errors should be stored.
- **blocking** (*bool*) – whether the queue will block (if False, then the queue will poll).
- **storage_kwargs** – arbitrary kwargs to pass to the storage implementation.

Example usage:

```python
from huey import RedisHuey, crontab

huey = RedisHuey('my-app')

@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 a *`TaskResultWrapper`* 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.

With a result store:

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

Without a result store:

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

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.

Returns A callable *TaskWrapper* instance.

Return type *TaskWrapper*

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 *TaskResultWrapper* 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 methods **onto** the decorated function, which makes it possible to *schedule* the execution for a certain time in the future, create task pipelines, etc. For more information, see:

- *TaskWrapper.schedule()*
- *TaskWrapper.s()*
- *TaskWrapper.revoke()*
- *TaskWrapper.is_revoked()*

- *TaskWrapper.restore()*

periodic_task(*validate_datetime*)

Function decorator that marks the decorated function for processing by the consumer *at a specific interval*. The `periodic_task` decorator serves to **mark a function as needing to be executed periodically** by the consumer.

Note: By default, the consumer will schedule and enqueue periodic task functions. To disable the enqueueing of periodic tasks, 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

Returns A callable *TaskWrapper* instance.

Return type `PeriodicQueueTask`

Like *task()*, the periodic task decorator adds 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. For more information, see *TaskWrapper*.

enqueue(*task*)

Enqueue the given task. When the result store is enabled (on by default), the return value will be a *TaskResultWrapper* which provides access to the result (among other things).

If the task specifies another task to run on completion (see *QueueTask.then()*), then the return value will be a `list` of *TaskResultWrapper* objects, one for each task in the pipeline.

Note: Unless you are executing a pipeline of tasks, it should not typically be necessary to use the *Huey.enqueue()* method. Calling (or scheduling) a `task`-decorated function will automatically enqueue a task for execution.

When you create a task pipeline, however, it is necessary to enqueue the pipeline once it has been set up.

Parameters **`task`** (`QueueTask`) – a *`QueueTask`* instance.

Returns A *`TaskResultWrapper`* object (if result store enabled).

`register_pre_execute`(*name*, *fn*)

Register a pre-execute hook. The callback will be executed before the execution of all tasks. Execution of the task can be cancelled by raising a `CancelExecution` exception. Uncaught exceptions will be logged but will not cause the task itself to be cancelled.

The callback function should accept a single task instance, the return value is ignored.

Hooks are executed in the order in which they are registered (which may be implicit if registered using the decorator).

Parameters

- **`name`** – Name for the hook.
- **`fn`** – Callback function that accepts task to be executed.

`unregister_pre_execute`(*name*)

Unregister the specified pre-execute hook.

`pre_execute`([*name=None*])

Decorator for registering a pre-execute hook.

Usage:

```
@huey.pre_execute()
def my_pre_execute_hook(task):
    do_something()
```

`register_post_execute`(*name*, *fn*)

Register a post-execute hook. The callback will be executed after the execution of all tasks. Uncaught exceptions will be logged but will have no other effect on the overall operation of the consumer.

The callback function should accept:

- a task instance
- the return value from the execution of the task (which may be None)
- any exception that was raised during the execution of the task (which will be None for tasks that executed normally).

The return value of the callback itself is ignored.

Hooks are executed in the order in which they are registered (which may be implicit if registered using the decorator).

Parameters

- **`name`** – Name for the hook.
- **`fn`** – Callback function that accepts task that was executed and the tasks return value (or None).

`unregister_post_execute`(*name*)

Unregister the specified post-execute hook.

post_execute([*name=None*])
Decorator for registering a post-execute hook.

Usage:

```
@huey.post_execute()
def my_post_execute_hook(task, task_value, exc):
    do_something()
```

register_startup(*name, fn*)
Register a startup hook. The callback will be executed whenever a worker comes online. Uncaught exceptions will be logged but will have no other effect on the overall operation of the worker.

The callback function must not accept any parameters.

This API is provided to simplify setting up global resources that, for whatever reason, should not be created as import-time side-effects. For example, your tasks need to write data into a Postgres database. If you create the connection at import-time, before the worker processes are spawned, you'll likely run into errors when attempting to use the connection from the child (worker) processes. To avoid this problem, you can register a startup hook which executes once when the worker starts up.

Parameters

- **name** – Name for the hook.
- **fn** – Callback function.

unregister_startup(*name*)
Unregister the specified startup hook.

on_startup([*name=None*])
Decorator for registering a startup hook. See `register_startup()` for information about start hooks.

Usage:

```
db_connection = None

@huey.on_startup()
def setup_db_connection():
    global db_connection
    db_connection = psycopg2.connect(database='my_db')

@huey.task()
def write_data(rows):
    cursor = db_connection.cursor()
    # ...
```

revoke(*task*[, *revoke_until=None*[, *revoke_once=False*]])
Prevent the given task **instance** from being executed by the consumer after it has been enqueued. To understand this method, you need to know a bit about how the consumer works. When you call a function decorated by the `Huey.task()` method, calls to that function will enqueue a message to the consumer indicating which task to execute, what the parameters are, etc. If the task is not scheduled to execute in the future, and there is a free worker available, the task starts executing immediately. Otherwise if workers are busy, it will wait in line for the next free worker.

When you revoke a task, when the worker picks up the revoked task to start executing it, it will instead just throw it away and get the next available task. So, revoking a task only has affect between the time you call the task and the time the worker actually starts executing the task.

Warning: This method only revokes a given **instance** of a task. Therefore, this method cannot be used with periodic tasks. To revoke **all** instances of a given task (including periodic tasks), see the `revoke_all()` method.

This function can be called multiple times, but each call will supercede any previous revoke settings.

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.

`restore`(*task*)

Takes a previously revoked task **instance** and restores it, allowing normal execution. If the revoked task was already consumed and discarded by a worker, then restoring will have no effect.

Note: If the task class itself has been revoked, restoring a given instance will not have any effect.

`revoke_by_id`(*task_id*[, *revoke_until=None*[, *revoke_once=False*]])

Exactly the same as `revoke()`, except it accepts a task instance ID instead of the task instance itself.

`restore_by_id`(*task_id*)

Exactly the same as `restore()`, except it accepts a task instance ID instead of the task instance itself.

`revoke_all`(*task_class*[, *revoke_until=None*[, *revoke_once=False*]])

Prevent any instance of the given task from being executed by the consumer.

Warning: This method affects all instances of a given task.

This function can be called multiple times, but each call will supercede any previous revoke settings.

Parameters

- **`revoke_until`** (*`datetime`*) – Prevent 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.

`restore_all`(*task_class*)

Takes a previously revoked task class and restores it, allowing normal execution. Restoring a revoked task class does not have any effect on individually revoked instances of the given task.

Note: Restoring a revoked task class does not have any effect on individually revoked instances of the given task.

`is_revoked`(*task*[, *dt=None*])

Returns a boolean indicating whether the given task instance/class is revoked. If the `dt` parameter is specified, then the result will indicate whether the task is revoked at that particular datetime.

Note: If a task class is specified, the return value will indicate only whether all instances of that task are revoked.

If a task instance/ID is specified, the return value will indicate whether the given instance **or** the task class itself has been revoked.

Parameters **`task`** – Either a task class, task instance or task ID.

Returns Boolean indicating whether the aforementioned task is revoked.

`result`(*task_id*[, *blocking=False*[, *timeout=None*[, *backoff=1.15*[, *max_delay=1.0*[, *revoke_on_timeout=False*[, *preserve=False*]]]]]])

Attempt to retrieve the return value of a task. By default, `result()` will simply check 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 the `timeout` is reached. If the `timeout` is reached before the result is ready, a `DataStoreTimeout` exception will be raised.

Note: If the task failed with an exception, then a `TaskException` that wraps the original exception will be raised.

Warning: By default the result store will delete a task's return value after the value has been successfully read (by a successful call to the `result()` or `TaskResultWrapper.get()` methods). If you need to use the task result multiple times, you must specify `preserve=True` when calling these methods.

Parameters

- **`task_id`** – the task's unique identifier.
- **`blocking`** (*`bool`*) – whether to block while waiting for task result
- **`timeout`** – number of seconds to block (if `blocking=True`)
- **`backoff`** – amount to backoff delay each iteration of loop
- **`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, thereby preventing it from running if it is has not started yet.
- **`preserve`** (*`bool`*) – see the above warning. When set to `True`, this parameter ensures that the task result should be preserved after having been successfully retrieved.

`lock_task`(*lock_name*)

Utilize the Storage key/value APIs to implement simple locking.

This lock is designed to be used to prevent multiple invocations of a task from running concurrently. Can be used as either a context-manager or as a task decorator. If using as a decorator, place it directly above the function declaration.

If a second invocation occurs and the lock cannot be acquired, then a special exception is raised, which is handled by the consumer. The task will not be executed and an `EVENT_LOCKED` will be emitted. If the task is configured to be retried, then it will be retried normally, but the failure to acquire the lock is not considered an error.

Examples:

```
@huey.periodic_task(crontab(minute='*/5'))
@huey.lock_task('reports-lock')
def generate_report():
    # If a report takes longer than 5 minutes to generate, we do
    # not want to kick off another until the previous invocation
    # has finished.
    run_report()

@huey.periodic_task(crontab(minute='0'))
def backup():
    # Generate backup of code
    do_code_backup()

    # Generate database backup. Since this may take longer than an
    # hour, we want to ensure that it is not run concurrently.
    with huey.lock_task('db-backup'):
        do_db_backup()
```

Parameters **`lock_name`** (*`str`*) – Name to use for the lock.

Returns Decorator or context-manager.

`pending`([*limit=None*])
Return all unexecuted tasks currently in the queue.

`scheduled`([*limit=None*])
Return all unexecuted tasks currently in the schedule.

`all_results`()
Return a mapping of task-id to pickled result data for all executed tasks whose return values have not been automatically removed.

`class TaskWrapper`(*huey*, *func*[, *retries=0*[, *retry_delay=0*[, *retries_as_argument=False*[, *include_task=False*[, *name=None*[, *task_base=None*[, ***task_settings*]]]]]]])

Parameters

- **`huey`** (`Huey`) – A huey instance.
- **`func`** – User function.
- **`retries`** (*`int`*) – Upon failure, number of times to retry the task.
- **`retry_delay`** (*`int`*) – Number of seconds to wait before retrying after a failure/exception.
- **`retries_as_argument`** (*`bool`*) – Pass the number of remaining retries as an argument to the user function.
- **`include_task`** (*`bool`*) – Pass the task object itself as an argument to the user function.
- **`name`** (*`str`*) – Name for task (will be determined based on task module and function name if not provided).
- **`task_base`** – Base-class for task, defaults to *`QueueTask`*.
- **`task_settings`** – Arbitrary settings to pass to the task class constructor.

Wrapper around a user-defined function that converts function calls into tasks executed by the consumer. The wrapper, which decorates the function, replaces the function in the scope with a *`TaskWrapper`* instance.

The wrapper class, when called, will enqueue the requested function call for execution by the consumer.

Note: You should not need to create *TaskWrapper* instances directly. Instead, use the *Huey.task()* and *Huey.periodic_task()* decorators.

The wrapper class also has several helper methods for managing and enqueueing tasks, which are described below.

schedule([*args=None*[, *kwargs=None*[, *eta=None*[, *delay=None*[, *convert_utc=True*]]]]])
Use the `schedule` method to schedule the execution of the queue task for a given time in the future:

```python
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. See note below on how to correctly specify the `eta` whether the consumer is running in UTC- or localtime-mode.
- **delay** (*int*) – number of seconds to wait before executing function
- **convert_utc** – whether the `eta` or `delay` should be converted from local time to UTC. Defaults to `True`. See note below.

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

Note: It can easily become confusing when/how to use the `convert_utc` parameter when scheduling tasks. Similarly, if you are using naive datetimes, whether the ETA should be based around `datetime.utcnow()` or `datetime.now()`.

If you are running the consumer in UTC-mode (the default):

- When specifying a `delay`, `convert_utc=True`.
- When specifying an `eta` with respect to `datetime.now()`, `convert_utc=True`.
- When specifying an `eta` with respect to `datetime.utcnow()`, `convert_utc=False`.

If you are running the consumer in localtime-mode (`-o`):

- When specifying a `delay`, `convert_utc=False`.
- When specifying an `eta`, it should always be with respect to `datetime.now()` with `convert_utc=False`.

In other words, for consumers running in UTC-mode, the only time `convert_utc=False` is when you are passing an `eta` that is already a naive datetime with respect to `utcnow()`.

Similarly for localtime-mode consumers, `convert_utc` should always be `False` and when specifying an `eta` it should be with respect to `datetime.now()`.

call_local()

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

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

revoke([*revoke_until=None*[, *revoke_once=False*]])

Prevent any instance of the given task from executing. When no parameters are provided the function will not execute again until explicitly restored.

This function can be called multiple times, but each call will supercede any limitations placed on the previous revocation.

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 of the next invocation of the task.

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

# prevent any invocation from executing.
count_some_beans.revoke()

# prevent any invocation for 24 hours.
count_some_beans.revoke(datetime.datetime.now() + datetime.timedelta(days=1))
```

is_revoked([*dt=None*])

Check whether the given task is revoked. If `dt` is specified, it will check if the task is revoked with respect to the given datetime.

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

restore()

Clears any revoked status and allows the task to run normally.

s([**args*[, ***kwargs*]])

Create a task instance representing the invocation of the user function with the given arguments and keyword-arguments. The resulting task instance is **not** enqueued automatically.

To illustrate the distinction, when you call a `task()`-decorated function, behind-the-scenes, Huey is doing something like this:

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

result = add(1, 2)

# Is equivalent to:
task = add.s(1, 2)
result = huey.enqueue(task)
```

Parameters

- **args** – Arguments for user-defined function.
- **kwargs** – Keyword arguments for user-defined function.

Returns a *QueueTask* instance representing the execution of the user-defined function with the given arguments.

Typically, one will use the *TaskWrapper.s()* helper when creating task execution pipelines.

For example:

```
add_task = add.s(1, 2)  # Represent task invocation.
pipeline = (add_task
            .then(add, 3)  # Call add() with previous result and 3.
            .then(add, 4)  # etc...
            .then(add, 5))

results = huey.enqueue(pipeline)

# Print results of above pipeline.
print([result.get(blocking=True) for result in results])

# [3, 6, 10, 15]
```

task_class

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

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

class QueueTask([*data=None*[, *task_id=None*[, *execute_time=None*[, *retries=None*[, *retry_delay=None*[, *on_complete=None*]]]]]])

The `QueueTask` class represents the execution of a function. Instances of the class are serialized and enqueued for execution by the consumer, which deserializes them and executes the function.

Note: You should not need to create instances of *QueueTask* directly, but instead use either the *Huey.task()* decorator or the *TaskWrapper.s()* method.

Parameters

- **data** – Data specific to this execution of the task. For `task()`-decorated functions, this will be a tuple of the `(args, kwargs)` the function was invoked with.
- **task_id** (*str*) – The task's ID, defaults to a UUID if not provided.
- **execute_time** (*datetime*) – Time at which task should be executed.
- **retries** (*int*) – Number of times to retry task upon failure/exception.
- **retry_delay** (*int*) – Number of seconds to wait before retrying a failed task.
- **on_complete** (`QueueTask`) – Task to execute upon completion of this task.

Here's a refresher on how tasks work:

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

ret = add(1, 2)
print(ret.get(blocking=True))  # "3".

# The above two lines are equivalent to:
task_instance = add.s(1, 2)  # Create a QueueTask instance.
ret = huey.enqueue(task_instance)  # Enqueue the queue task.
print(ret.get(blocking=True))  # "3".
```

then(*task*[, **args*[, ***kwargs*]])

Parameters

- **task** (`TaskWrapper`) – A `task()`-decorated function.
- **args** – Arguments to pass to the task.
- **kwargs** – Keyword arguments to pass to the task.

Returns The parent task.

The `then()` method is used to create task pipelines. A pipeline is a lot like a unix pipe, such that the return value from the parent task is then passed (along with any parameters specified by `args` and `kwargs`) to the child task.

Here's an example of chaining some addition operations:

```
add_task = add.s(1, 2)  # Represent task invocation.
pipeline = (add_task
            .then(add, 3)  # Call add() with previous result and 3.
            .then(add, 4)  # etc...
            .then(add, 5))

results = huey.enqueue(pipeline)

# Print results of above pipeline.
print([result.get(blocking=True) for result in results])

# [3, 6, 10, 15]
```

If the value returned by the parent function is a `tuple`, then the tuple will be used to update the `*args` for the child function. Likewise, if the parent function returns a `dict`, then the dict will be used to update the `**kwargs` for the child function.

Example of chaining fibonacci calculations:

```
@huey.task()
def fib(a, b=1):
    a, b = a + b, a
    return (a, b)  # returns tuple, which is passed as *args

pipe = (fib.s(1)
        .then(fib)
        .then(fib))
results = huey.enqueue(pipe)

```

(continues on next page)

(continued from previous page)

```
print([result.get(blocking=True) for result in results])
# [(2, 1), (3, 2), (5, 3)]
```

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.

Day-of-week uses 0=Sunday and 6=Saturday.

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

Note: It is currently not possible to run periodic tasks with an interval less than once per minute. If you need to run tasks more frequently, you can create a periodic task that runs once per minute, and from that task, schedule any number of sub-tasks to run after the desired delays.

3.7.2 TaskResultWrapper

class TaskResultWrapper(*huey, task*)

Although you will probably never instantiate an `TaskResultWrapper` object yourself, they are returned by any calls to `task()` decorated functions (provided that *huey* is configured with a result store). The `TaskResultWrapper` 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.

Note: By default, the data is removed from the result store after being read, but this behavior can be disabled.

Getting results from tasks is very simple:

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

>>> res()  # Fetch the result of this task.
'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 None is returned.

>>> res() is None  # We can omit ".get", it works the same way.
```

(continues on next page)

(continued from previous page)

```
True

>>> res(blocking=True, timeout=5)  # Block for up to 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(blocking=True)  # No timeout, will block until it gets data.
'Counted 10000000 beans'
```

If the task failed with an exception, then a `TaskException` will be raised when reading the result value:

```
>>> @huey.task()
... def fails():
...     raise Exception('I failed')

>>> res = fails()
>>> res()  # raises a TaskException!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/charles/tmp/huey/src/huey/huey/api.py", line 684, in get
    raise TaskException(result.metadata)
huey.exceptions.TaskException: Exception('I failed',)
```

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

Attempt to retrieve the return value of a task. By default, *get()* will simply check 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 the `timeout` is reached. If the `timeout` is reached before the result is ready, a `DataStoreTimeout` exception will be raised.

Warning: By default the result store will delete a task's return value after the value has been successfully read (by a successful call to the *result()* or *TaskResultWrapper.get()* methods). If you need to use the task result multiple times, you must specify `preserve=True` when calling these methods.

Note: Instead of calling `.get()`, you can simply call the *TaskResultWrapper* object directly. Both methods accept the same parameters.

Parameters

- **blocking** (*bool*) – whether to block while waiting for task result
- **timeout** – number of seconds to block (if `blocking=True`)
- **backoff** – amount to backoff delay each iteration of loop
- **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, thereby preventing it from running if it is has not started yet.

- **preserve** (*bool*) – see the above warning. When set to `True`, this parameter ensures that the task result should be preserved after having been successfully retrieved.

`__call__`(***kwargs*)
Identical to the `get()` method, provided as a shortcut.

`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 instance. Unless the task instance has already been dequeued and discarded, it will be restored and run as scheduled.

Warning: If the task class itself has been revoked, then this method has no effect.

`is_revoked`()
Return a boolean value indicating whether this particular task instance **or** the task class itself has been revoked.

See also: `Huey.is_revoked()`.

`reschedule`([*eta=None*[, *delay=None*[, *convert+utc=True*]]])
Reschedule the given task. The original task instance will be revoked, but **no checks are made** to verify that it hasn't already been executed.

If neither an `eta` nor a `delay` is specified, the task will be run as soon as it is received by a worker.

Parameters

- **eta** (*datetime*) – the time at which the function should be executed. See note below on how to correctly specify the `eta` whether the consumer is running in UTC- or
- **delay** (*int*) – number of seconds to wait before executing function
- **convert_utc** – whether the `eta` or `delay` should be converted from local time to UTC. Defaults to `True`. See the note in the `schedule()` method of `Huey.task()` for more information.

Return type `TaskResultWrapper` object for the new task.

`reset`()
Reset the cached result and allow re-fetching a new result for the given task (i.e. after a task error and subsequent retry).

3.7.3 Storage

Huey

class `BaseStorage`([*name='huey'*[, ***storage_kwargs*]])

3.8 Huey Extensions

The `huey.contrib` package contains modules that provide extra functionality beyond the core APIs.

3.8.1 Mini-Huey

`MiniHuey` provides a very lightweight huey-like API that may be useful for certain classes of applications. Unlike *Huey*, the `MiniHuey` consumer runs inside a greenlet in your main application process. This means there is no separate consumer process to run, not is there any persistence for the enqueued/scheduled tasks; whenever a task is enqueued or is scheduled to run, a new greenlet is spawned to execute the task.

Usage and task declaration:

class MiniHuey([*name='huey'*[, *interval=1*[, *pool_size=None*]]])

Parameters

- **name** (*str*) – Name given to this huey instance.
- **interval** (*int*) – How frequently to check for scheduled tasks (seconds).
- **pool_size** (*int*) – Limit number of concurrent tasks to given size.

```
from huey import crontab
from huey.contrib.minimal import MiniHuey


huey = MiniHuey()

@huey.task()
def fetch_url(url):
    return urllib2.urlopen(url).read()

@huey.task(crontab(minute='0', hour='4'))
def run_backup():
    pass
```

Note: There is not a separate decorator for *periodic*, or *crontab*, tasks. Just use `MiniHuey.task()` and pass in a validation function.

When your application starts, be sure to start the `MiniHuey` scheduler:

```
from gevent import monkey; monkey.patch_all()

huey.start()  # Kicks off scheduler in a new greenlet.
start_wsgi_server()  # Or whatever your main application is doing...
```

Warning: Tasks enqueued manually for immediate execution will be run regardless of whether the scheduler is running. If you want to be able to schedule tasks in the future or run periodic tasks, you will need to call `start()`.

Calling tasks and getting results works about the same as regular huey:

```
async_result = fetch_url('https://www.google.com/')
html = async_result.get()  # Block until task is executed.

# Fetch the Yahoo! page in 30 seconds.
async_result = fetch_url.schedule(args=('https://www.yahoo.com/',),
                                  delay=30)
html = async_result.get()  # Blocks for ~30s.
```

3.8.2 SQLite Storage

The `SqliteHuey` and the associated `SqliteStorage` can be used instead of the default `RedisHuey`. `SqliteHuey` is implemented in such a way that it can safely be used with a multi-process, multi-thread, or multi-greenlet consumer.

Using `SqliteHuey` is almost exactly the same as using `RedisHuey`. Begin by instantiating the `Huey` object, passing in the name of the queue **and the filename** of the SQLite database:

```
from huey.contrib.sqlitedb import SqliteHuey

huey = SqliteHuey('my_app', filename='/var/www/my_app/huey.db')
```

Note: The SQLite storage engine depends on peewee. For information on installing peewee, see the peewee installation documentation, or simply run: `pip install peewee`.

3.8.3 Simple Server

Huey supports a simple client/server database that can be used for development and testing. The server design is inspired by redis and implements commands that map to the methods described by the storage API. If you'd like to read a technical post about the implementation, check out this blog post.

The server can optionally use gevent, but if gevent is not available you can use threads (use `-t` for threads).

To obtain the simple server, you can clone the `simpledb` repository:

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

Running the simple server

Usage:

```
Usage: simpledb.py [options]

Options:
  -h, --help            show this help message and exit
  -d, --debug           Log debug messages.
  -e, --errors          Log error messages only.
  -t, --use-threads     Use threads instead of gevent.
  -H HOST, --host=HOST  Host to listen on.
  -m MAX_CLIENTS, --max-clients=MAX_CLIENTS
```

(continues on next page)

(continued from previous page)

```
                        Maximum number of clients.
  -p PORT, --port=PORT  Port to listen on.
  -l LOG_FILE, --log-file=LOG_FILE
                        Log file.
  -x EXTENSIONS, --extensions=EXTENSIONS
                        Import path for Python extension module(s).
```

By default the server will listen on localhost, port 31337.

Example (with logging):

```
$ python simpledb.py --debug --log-file=/var/log/huey-simple.log
```

Using simple server with Huey

To use the simple server with Huey, use the `SimpleHuey` class:

```
from huey.contrib.simple_storage import SimpleHuey


huey = SimpleHuey('my-app')

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

The `SimpleHuey` class relies on a `SimpleStorage` storage backend, which in turn, uses the `simple.Client` client class.

3.8.4 Django

Huey comes with special integration for use with the Django framework. The integration provides:

1. Configuration of huey via the Django settings module.
2. Running the consumer as a Django management command.
3. Auto-discovery of `tasks.py` modules to simplify task importing.
4. Properly manage database connections.

Supported Django versions are the officially supported at https://www.djangoproject.com/download/#supported-versions

Setting things up

To use huey with Django, the first step is to add an entry to your project's `settings.INSTALLED_APPS`:

```
# settings.py
# ...
INSTALLED_APPS = (
    # ...
    'huey.contrib.djhuey',  # Add this to the list.
    # ...
)
```

The above is the bare minimum needed to start using huey's Django integration. If you like, though, you can also configure both Huey and the *consumer* using the settings module.

Note: Huey settings are optional. If not provided, Huey will default to using Redis running on localhost:6379 (standard setup).

Configuration is kept in `settings.HUEY`, which can be either a dictionary or a `Huey` instance. Here is an example that shows all of the supported options with their default values:

```
# settings.py
HUEY = {
    'name': settings.DATABASES['default']['NAME'],  # Use db name for huey.
    'result_store': True,  # Store return values of tasks.
    'events': True,  # Consumer emits events allowing real-time monitoring.
    'store_none': False,  # If a task returns None, do not save to results.
    'always_eager': settings.DEBUG,  # If DEBUG=True, run synchronously.
    'store_errors': True,  # Store error info if task throws exception.
    'blocking': False,  # Poll the queue rather than do blocking pop.
    'backend_class': 'huey.RedisHuey',  # Use path to redis huey by default,
    'connection': {
        'host': 'localhost',
        'port': 6379,
        'db': 0,
        'connection_pool': None,  # Definitely you should use pooling!
        # ... tons of other options, see redis-py for details.

        # huey-specific connection parameters.
        'read_timeout': 1,  # If not polling (blocking pop), use timeout.
        'max_errors': 1000,  # Only store the 1000 most recent errors.
        'url': None,  # Allow Redis config via a DSN.
    },
    'consumer': {
        'workers': 1,
        'worker_type': 'thread',
        'initial_delay': 0.1,  # Smallest polling interval, same as -d.
        'backoff': 1.15,  # Exponential backoff using this rate, -b.
        'max_delay': 10.0,  # Max possible polling interval, -m.
        'utc': True,  # Treat ETAs and schedules as UTC datetimes.
        'scheduler_interval': 1,  # Check schedule every second, -s.
        'periodic': True,  # Enable crontab feature.
        'check_worker_health': True,  # Enable worker health checks.
        'health_check_interval': 1,  # Check worker health every second.
    },
}
```

Alternatively, you can simply set `settings.HUEY` to a `Huey` instance and do your configuration directly. In the example below, I've also shown how you can create a connection pool:

```
# settings.py -- alternative configuration method
from huey import RedisHuey
from redis import ConnectionPool

pool = ConnectionPool(host='my.redis.host', port=6379, max_connections=20)
HUEY = RedisHuey('my-app', connection_pool=pool)
```

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 using both the django settings module and/or by specifying options from the command-line.

Note: Options specified on the command line take precedence over those specified in the settings module.

To start the consumer, you simply run:

```
$ ./manage.py run_huey
```

In addition to the `HUEY.consumer` setting dictionary, the management command supports all the same options as the standalone consumer. These options are listed and described in the *Options for the consumer* section.

For quick reference, the most important command-line options are briefly listed here.

-w, --workers Number of worker threads/processes/greenlets. Default is 1, but most applications should use at least 2.

-k, --worker-type Worker type, must be "thread", "process" or "greenlet". The default is *thread*, which provides good all-around performance. For CPU-intensive workloads, *process* is likely to be more performant. The *greenlet* worker type is suited for IO-heavy workloads. When using *greenlet* you can specify tens or hundreds of workers since they are extremely lightweight compared to threads/processes. *See note below on using gevent/greenlet.*

Note: Due to a conflict with Django's base option list, the "verbose" option is set using `-V` or `--huey-verbose`. When enabled, huey logs at the DEBUG level.

For more information, read the *Options for the consumer* section.

Using gevent

When using worker type *greenlet*, it's necessary to apply a monkey-patch before any libraries or system modules are imported. Gevent monkey-patches things like `socket` to provide non-blocking I/O, and if those modules are loaded before the patch is applied, then the resulting code will execute synchronously.

Unfortunately, because of Django's design, the only way to reliably apply this patch is to create a custom bootstrap script that mimics the functionality of `manage.py`. Here is the patched `manage.py` code:

```python
#!/usr/bin/env python
import os
import sys

# Apply monkey-patch if we are running the huey consumer.
if 'run_huey' in sys.argv:
    from gevent import monkey
    monkey.patch_all()

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "conf")
    from django.core.management import execute_from_command_line
    execute_from_command_line(sys.argv)
```

How to create tasks

The `task()` and `periodic_task()` decorators can be imported from the `huey.contrib.djhuey` module. Here is how you might define two tasks:

```
from huey import crontab
from huey.contrib.djhuey import 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 import crontab
from huey.contrib.djhuey import 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.
```

DEBUG and Synchronous Execution

When `settings.DEBUG = True`, tasks will be executed **synchronously** just like regular function calls. The purpose of this is to avoid running both Redis and an additional consumer process while developing or running tests. If, however, you would like to enqueue tasks regardless of whether `DEBUG = True`, then explicitly specify `always_eager=False` in your huey settings:

```
# settings.py
HUEY = {
    'name': 'my-app',
    # Other settings ...
    'always_eager': False,
}
```

Configuration Examples

This section contains example `HUEY` configurations.

```
# Redis running locally with four worker threads.
HUEY = {
    'name': 'my-app',
    'consumer': {'workers': 4, 'worker_type': 'thread'},
}
```

```
# Redis on network host with 64 worker greenlets and connection pool
# supporting up to 100 connections.
from redis import ConnectionPool

pool = ConnectionPool(
    host='192.168.1.123',
    port=6379,
    max_connections=100)

HUEY = {
    'name': 'my-app',
    'connection': {'connection_pool': pool},
    'consumer': {'workers': 64, 'worker_type': 'greenlet'},
}
```

It is also possible to specify the connection using a Redis URL, making it easy to configure this setting using a single environment variable:

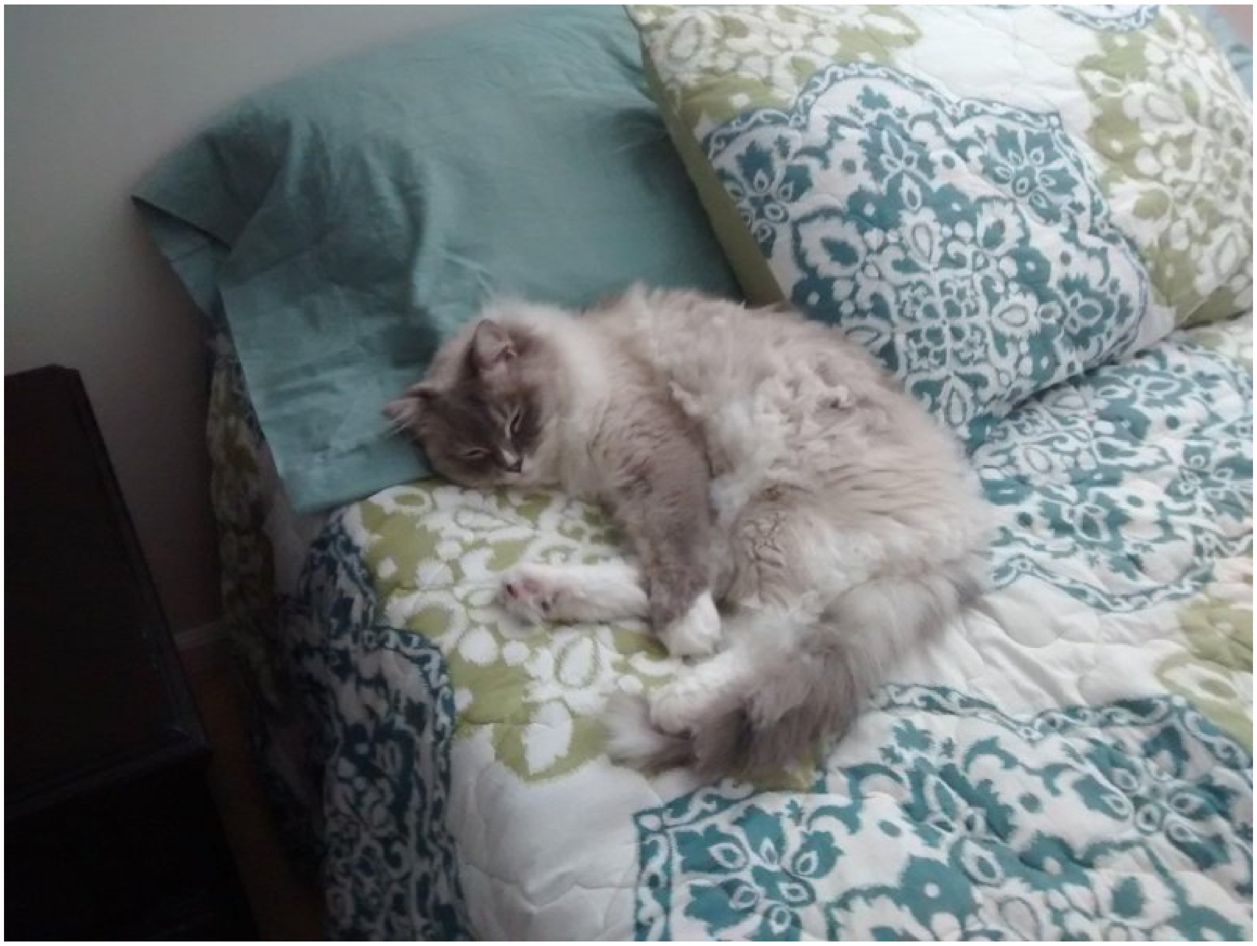
```
HUEY = {
    'name': 'my-app',
    'url': os.environ.get('REDIS_URL', 'redis://localhost:6379/?db=1')
}
```

Alternatively, you can just assign a *Huey* instance to the `HUEY` setting:

```
from huey import RedisHuey

HUEY = RedisHuey('my-app')
```

Huey is named in honor of my cat



CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index