huey Documentation

Release 2.2.0

charles leifer

Mar 04, 2020

Contents

a lightweight alternative.

huey is:

- a task queue (**2019-04-01**: *version 2.0 released*)
- written in python (2.7+, 3.4+)
- clean and simple API
- redis, sqlite, or in-memory storage
- example code.

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 prioritization
- task result storage
- task locking
- task pipelines and chains

CHAPTER 1

At a glance

task() and *periodic_task()* decorators turn functions into tasks executed 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.task(retries=2, retry_delay=60)
def flaky_task(url):
    # This task might fail, in which case it will be retried up to 2 times
    # with a delay of 60s between retries.
    return this_might_fail(url)

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

Calling a `task`-decorated function will enqueue the function call for execution by the consumer. A special result handle is returned immediately, which can be used to fetch the result once the task is finished:

```
>>> from demo import add_numbers
>>> res = add_numbers(1, 2)
>>> res
<Result: task 6b6f36fc-da0d-4069-b46c-c0d4ccff1df6>

>>> res()
3
```

Tasks can be scheduled to run in the future:

```
>>> res = add_numbers.schedule((2, 3), delay=10)  # Will be run in ~10s.
>>> res(blocking=True)  # Will block until task finishes, in ~10s.
5
```

For much more, check out the *Guide* or take a look at the example code.

1.1 Running the consumer

Run the consumer with four worker processes:

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

To run the consumer with a single worker thread (default):

```
$ huey_consumer.py my_app.huey
```

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
```

For more information, see the *Consuming Tasks* document.

CHAPTER 2

Table of contents

2.1 Installing

huey can be installed from PyPI using pip.

```
$ pip install huey
```

huey has no dependencies outside the standard library, but redis-py is required to utilize Redis for your task storage:

```
$ 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
```

2.1.1 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

2.2 Guide

The purpose of this document is to present Huey using simple examples that cover the most common usage of the library. Detailed documentation can be found in the *API documentation*.

Example `task()` that adds two numbers:

```
# demo.py
from huey import SqliteHuey

huey = SqliteHuey(filename='/tmp/demo.db')

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

To test, run the consumer, specifying the import path to the `huey` object:

```
$ huey_consumer.py demo.huey
```

In a Python shell, we can call our `add` task:

```
>>> from demo import add
>>> r = add(1, 2)
>>> r()
3
```

Note: If you try to resolve the result (`r`) before the task has been executed, then `r()` will return `None`. You can avoid this by instructing the result to block until the task has finished and a result is ready:

```
>>> r = add(1, 2)
>>> r(blocking=True, timeout=5)  # Wait up to 5 seconds for result.
3
```

What happens when we call a task function?

1. When the `add()` function is called, a message representing the call is placed in a queue.
2. The function returns immediately without actually running, and returns a `Result` handle, which can be used to retrieve the result once the task has been executed by the consumer.
3. The consumer process sees that a message has arrived, and a worker will call the `add()` function and place the return value into the result store.
4. We can use the `Result` handle to read the return value from the result store.

For more information, see the `task()` decorator documentation.

2.2.1 Scheduling tasks

Tasks can be scheduled to execute at a certain time, or after a delay.

In the following example, we will schedule a call to `add()` to run in 10 seconds, and then will block until the result becomes available:

```
>>> r = add.schedule((3, 4), delay=10)
>>> r(blocking=True)  # Will block for ~10 seconds before returning.
7
```

If we wished to schedule the task to run at a particular time, we can use the `eta` parameter instead. The following example will run after a 10 second delay:

```
>>> eta = datetime.datetime.now() + datetime.timedelta(seconds=10)
>>> r = add.schedule((4, 5), eta=eta)
>>> r(blocking=True)  # Will block for ~10 seconds.
9
```

What happens when we schedule a task?

1. When we call *`schedule()`*, a message is placed on the queue instructing the consumer to call the `add()` function in 10 seconds.
2. The function returns immediately, and returns a *`Result`* handle.
3. The consumer process sees that a message has arrived, and will notice that the message is not yet ready to be executed, but should be run in ~10s.
4. The consumer adds the message to a schedule.
5. In ~10 seconds, the scheduler will pick-up the message and place it back into the queue for execution.
6. A worker will dequeue the message, execute the `add()` function, and place the return value in the result store.
7. The *`Result`* handle from step 2 will now be able to read the return value from the task.

For more details, see the *`schedule()`* API documentation.

2.2.2 Periodic tasks

Huey provides crontab-like functionality that enables functions to be executed automatically on a given schedule.

In the following example, we will declare a *`periodic_task()`* that executes every 3 minutes and prints a message on consumer process stdout:

```
from huey import SqliteHuey
from huey import crontab

huey = SqliteHuey(filename='/tmp/demo.db')

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

@huey.periodic_task(crontab(minute='*/3'))
def every_three_minutes():
    print('This task runs every three minutes')
```

Once a minute, the scheduler will check to see if any of the periodic tasks should be called. If so, the task will be enqueued for execution.

Note: Because periodic tasks are called independent of any user interaction, they do not accept any arguments.

Similarly, the return-value for periodic tasks is discarded, rather than being put into the result store. This is because there is not an obvious way for an application to obtain a *Result* handle to access the result of a given periodic task execution.

The *crontab()* function accepts the following arguments:

- minute
- hour
- day
- month
- day_of_week (0=Sunday, 6=Saturday)

Acceptable inputs:

- `*` - always true, e.g. if `hour='*'`, then the rule matches any hour.
- `*/n` - every *n* interval, e.g. `minute='*/15'` means every 15 minutes.
- `m-n` - run every time `m..n` inclusive.
- `m,n` - run on *m* and *n*.

Multiple rules can be expressed by separating the individual rules with a comma, for example:

```
# Runs every 10 minutes between 9a and 11a, and 4p-6p.
crontab(minute='*/10', hour='9-11,16-18')
```

For more information see the following API documentation:

- *periodic_task()*
- *crontab()*

2.2.3 Retrying tasks that fail

Sometimes we may have a task that we anticipate might fail from time to time, in which case we should retry it. Huey supports automatically retrying tasks a given number of times, optionally with a delay between attempts.

Here we'll declare a task that fails approximately half of the time. To configure this task to be automatically retried, use the `retries` parameter of the *task()* decorator:

```
import random

@huey.task(retries=2)  # Retry the task up to 2 times.
def flaky_task():
    if random.randint(0, 1) == 0:
        raise Exception('failing!')
    return 'OK'
```

What happens when we call this task?

1. Message is placed on the queue and a *Result* handle is returned to the caller.
2. Consumer picks up the message and attempts to run the task, but the call to `random.randint()` happens to return `0`, raising an `Exception`.

3. The consumer puts the error into the result store and the exception is logged. If the caller resolves the *Result* now, a *TaskException* will be raised which contains information about the exception that occurred in our task.
4. The consumer notices that the task can be retried 2 times, so it decrements the retry count and re-enqueues it for execution.
5. The consumer picks up the message again and runs the task. This time, the task succeeds! The new return value is placed into the result store ("OK").
6. We can reset our *Result* handle by calling *reset()* and then re-resolve it. The result handle will now give us the new value, "OK".

Should the task fail on the first invocation, it will be retried up-to two times. Note that it will be retried *immediately* after it returns.

To specify a delay between retry attempts, we can add a `retry_delay` argument. The task will be retried up-to two times, with a delay of 10 seconds between attempts:

```
@huey.task(retries=2, retry_delay=10)
def flaky_task():
    # ...
```

Note: Retries and retry delay arguments can also be specified for periodic tasks.

It is also possible to explicitly retry a task from within the task, by raising a *RetryTask* exception. When this exception is used, the task will be retried regardless of whether it was declared with `retries`. Similarly, the task's remaining retries (if they were declared) will not be affected by raising *RetryTask*.

For more information, see the following API documentation:

- *task()* and *periodic_task()*
- *Result*

2.2.4 Task priority

Note: Priority support for Redis requires Redis 5.0 or newer. To use task priorities with Redis, use the *PriorityRedisHuey* instead of *RedisHuey*.

Task prioritization is fully supported by *SqliteHuey* and the in-memory storage layer used when *Immediate mode* is enabled.

Huey tasks can be given a priority, allowing you to ensure that your most important tasks do not get delayed when the workers are busy.

Priorities can be assigned to a task function, in which case all invocations of the task will default to the given priority. Additionally, individual task invocations can be assigned a priority on a one-off basis.

Note: When no priority is given, the task will default to a priority of `0`.

To see how this works, lets define a task that has a priority (`10`):

```
@huey.task(priority=10)
def send_email(to, subj, body):
    return mailer.send(to, 'webmaster@myapp.com', subj, body)
```

When we invoke this task, it will be processed *before* any other pending tasks whose priority is less than 10. So we could imagine our queue looking something like this:

- `process_payment` - priority = 50
- `check_spam` - priority = 1
- `make_thumbnail` - priority = 0 (default)

Invoke the `send_email()` task:

```
send_email('new_user@foo.com', 'Welcome', 'blah blah')
```

Now the queue of pending tasks would be:

- `process_payment` - priority = 50
- `send_email` - priority = 10
- `check_spam` - priority = 1
- `make_thumbnail` - priority = 0

We can override the default priority by passing `priority=` as a keyword argument to the task function:

```
send_email('boss@mycompany.com', 'Important!', 'etc', priority=90)
```

Now the queue of pending tasks would be:

- `send_email` (to boss) - priority = 90
- `process_payment` - priority = 50
- `send_email` - priority = 10
- `check_spam` - priority = 1
- `make_thumbnail` - priority = 0

Task priority only affects the ordering of tasks as they are pulled from the queue of pending tasks. If there are periods of time where your workers are not able to keep up with the influx of tasks, Huey's `priority` feature can ensure that your most important tasks do not get delayed.

Task-specific priority overrides can also be specified when scheduling a task to run in the future:

```
# Uses priority=10, since that was the default we used when
# declaring the send_email task:
send_email.schedule(('foo@bar.com', 'subj', 'msg'), delay=60)

# Override, specifying priority=50 for this task.
send_email.schedule(('bar@foo.com', 'subj', 'msg'), delay=60, priority=50)
```

Lastly, we can specify priority on *`periodic_task`*:

```
@huey.periodic_task(crontab(minute='0', hour='*/3'), priority=10)
def some_periodic_task():
    # ...
```

For more information:

- *PriorityRedisHuey* - Huey implementation that adds support for task priorities with the Redis storage layer. *Requires Redis 5.0 or newer*.
- *SqliteHuey* and the in-memory storage used when immediate-mode is enabled have full support for task priorities.
- *task()* and *periodic_task()*

2.2.5 Canceling or pausing tasks

Huey tasks can be cancelled dynamically at runtime. This applies to regular tasks, tasks scheduled to execute in the future, and periodic tasks.

Any task can be canceled ("revoked"), provided the task has not started executing yet. Similarly, a revoked task can be restored, provided it has not already been processed and discarded by the consumer.

Using the *Result.revoke()* and *Result.restore()* methods:

```
# Schedule a task to execute in 60 seconds.
res = add.schedule((1, 2), delay=60)

# Provided the 60s has not elapsed, the task can be canceled
# by calling the `revoke()` method on the result object.
res.revoke()

# We can check to see if the task is revoked.
res.is_revoked()  # -> True

# Similarly, we can restore the task, provided the 60s has
# not elapsed (at which point it would have been read and
# discarded by the consumer).
res.restore()
```

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

```
# Prevent all instances of the add() task from running.
add.revoke()

# We can check to see that all instances of the add() task
# are revoked:
add.is_revoked()  # -> True

# We can enqueue an instance of the add task, and then check
# to verify that it is revoked:
res = add(1, 2)
res.is_revoked()  # -> True

# To re-enable a task, we'll use the restore() method on
# the task function:
add.restore()

# Is the add() task enabled again?
add.is_revoked()  # -> False
```

Huey provides APIs to revoke / restore on both individual instances of a task, as well as all instances of the task. For more information, see the following API docs:

- *Result.revoke()* and *Result.restore()* for revoking individual instances of a task.

- *Result.is_revoked()* for checking the status of a task instance.
- *TaskWrapper.revoke()* and *TaskWrapper.restore()* for revoking all instances of a task.
- *TaskWrapper.is_revoked()* for checking the status of the task function itself.

2.2.6 Canceling or pausing periodic tasks

The `revoke()` and `restore()` methods support some additional options which may be especially useful for *periodic_task()*.

The *revoke()* method accepts two optional parameters:

- `revoke_once` - boolean flag, if set then only the next occurrence of the task will be revoked, after which it will be restored automatically.
- `revoke_until` - datetime, which specifies the time at which the task should be automatically restored.

For example, suppose we have a task that sends email notifications, but our mail server goes down and won't be fixed for a while. We can revoke the task for a couple of hours, after which time it will start executing again:

```
@huey.periodic_task(crontab(minute='0', hour='*'))
def send_notification_emails():
    # ... code to send emails ...
```

Here is how we might revoke the task for the next 3 hours:

```
>>> now = datetime.datetime.now()
>>> eta = now + datetime.timedelta(hours=3)
>>> send_notification_emails.revoke(revoke_until=eta)
```

Alternatively, we could use `revoke_once=True` to just skip the next execution of the task:

```
>>> send_notification_emails.revoke(revoke_once=True)
```

At any time, the task can be restored using the usual *restore()* method, and it's status can be checked using the *is_revoked()* method.

2.2.7 Task pipelines

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

To get started, let's review the usual way we execute tasks:

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

result = add(1, 2)
```

An equivalent, but more verbose, way is to use the *s()* method to create a *Task* instance and then enqueue it explicitly:

```
# Create a task representing the execution of add(1, 2).
task = add.s(1, 2)

# Enqueue the task instance, which returns a Result handle.
result = huey.enqueue(task)
```

So the following are equivalent:

```
result = add(1, 2)

# And:
result = huey.enqueue(add.s(1, 2))
```

The *TaskWrapper.s()* method is used to create a *Task* instance (which represents the execution of the given function). The `Task` is what gets serialized and sent to the consumer.

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

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

# Add additional tasks to pipeline by calling add_task.then().
pipeline = (add_task
            .then(add, 3)  # Call add() with previous result (1+2) and 3.
            .then(add, 4)  # Previous result ((1+2)+3) and 4.
            .then(add, 5)) # Etc.

# When a pipeline is enqueued, a ResultGroup is returned (which is
# comprised of individual Result instances).
result_group = huey.enqueue(pipeline)

# Print results of above pipeline.
print(result_group.get(blocking=True))
# [3, 6, 10, 15]

# Alternatively, we could have iterated over the result group:
for result in result_group:
    print(result.get(blocking=True))
# 3
# 6
# 10
# 15
```

When enqueueing a task pipeline, the return value will be a *ResultGroup*, which encapsulates the *Result* objects for the individual tasks. *ResultGroup* can be iterated over or you can use the *ResultGroup.get()* method to get all the task return values as a list.

Note that the return value from the parent task is passed to the next task in the pipeline, and so on.

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

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)
        .then(fib))
results = huey.enqueue(pipe)
```

(continues on next page)

(continued from previous page)

```
print(results(True))  # Resolve results, blocking until all are finished.
# [(2, 1), (3, 2), (5, 3), (8, 5)]
```

For more information, see the following API docs:

- *TaskWrapper.s()*
- *Task.then()*
- *ResultGroup* and *Result*

2.2.8 Locking tasks

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

This lock prevents multiple invocations of a task from running concurrently.

If a second invocation occurs and the lock cannot be acquired, then a special *TaskLockedException* is raised and the task will not be executed. If the task is configured to be retried, then it will be retried normally.

Examples:

```
@huey.periodic_task(crontab(minute='*/5'))
@huey.lock_task('reports-lock')  # Goes *after* the task decorator.
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()
```

See *Huey.lock_task()* for API documentation.

2.2.9 Signals

The `Consumer` sends *signals* as it processes tasks. The *Huey.signal()* method can be used to attach a callback to one or more signals, which will be invoked synchronously by the consumer when the signal is sent.

For a simple example, we can add a signal handler that simply prints the signal name and the ID of the related task.

```
@huey.signal()
def print_signal_args(signal, task, exc=None):
    if signal == SIGNAL_ERROR:
        print('%s - %s - exception: %s' % (signal, task.id, exc))
```

(continues on next page)

(continued from previous page)

```
    else:
        print('%s - %s' % (signal, task.id))
```

The *signal()* method is used to decorate the signal-handling function. It accepts an optional list of signals. If none are provided, as in our example, then the handler will be called for any signal.

The callback function (`print_signal_args`) accepts two required arguments, which are present on every signal: `signal` and `task`. Additionally, our handler accepts an optional third argument `exc` which is only included with `SIGNAL_ERROR`. `SIGNAL_ERROR` is only sent when a task raises an uncaught exception during execution.

Warning: Signal handlers are executed *synchronously* by the consumer, so it is typically a bad idea to introduce any slow operations into a signal handler.

For a complete list of Huey's signals and their meaning, see the *Signals* document, and the *Huey.signal()* API documentation.

2.2.10 Immediate mode

Note: Immediate mode replaces the *always eager* mode available prior to the release of Huey 2. It offers many improvements over always eager mode, which are described in the *Changes in 2.0* document.

Huey can be run in a special mode called *immediate* mode, which is very useful during testing and development. In immediate mode, Huey will execute task functions immediately rather than enqueueing them, while still preserving the APIs and behaviors one would expect when running a dedicated consumer process.

Immediate mode can be enabled in two ways:

```
huey = RedisHuey('my-app', immediate=True)

# Or at any time, via the "immediate" attribute:
huey = RedisHuey('my-app')
huey.immediate = True
```

To disable immediate mode:

```
huey.immediate = False
```

By default, enabling immediate mode will switch your Huey instance to using in-memory storage. This is to prevent accidentally reading or writing to live storage while doing development or testing. If you prefer to use immediate mode with live storage, you can specify `immediate_use_memory=False` when creating your *Huey* instance:

```
huey = RedisHuey('my-app', immediate_use_memory=False)
```

You can try out immediate mode quite easily in the Python shell. In the following example, everything happens within the interpreter – no separate consumer process is needed. In fact, because immediate mode switches to an in-memory storage when enabled, we don't even have to be running a Redis server:

```
>>> from huey import RedisHuey
>>> huey = RedisHuey()
>>> huey.immediate = True

```

(continues on next page)

(continued from previous page)

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

>>> result = add(1, 2)
>>> result()
3

>>> add.revoke(revoke_once=True)  # We can revoke tasks.
>>> result = add(2, 3)
>>> result() is None
True

>>> add(3, 4)()  # No longer revoked, was restored automatically.
7
```

What happens if we try to schedule a task for execution in the future, while using immediate mode?

```
>>> result = add.schedule((4, 5), delay=60)
>>> result() is None  # No result.
True
```

As you can see, the task was not executed. So what happened to it? The answer is that the task was added to the in-memory storage layer's schedule. We can check this by calling *Huey.scheduled()*:

```
>>> huey.scheduled()
[__main__.add: 8873...bcbd @2019-03-27 02:50:06]
```

Since immediate mode is fully synchronous, there is not a separate thread monitoring the schedule. The schedule can still be read or written to, but scheduled tasks will not automatically be executed.

2.2.11 Tips and tricks

To call a task-decorated function in its original form, you can use *call_local()*:

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

# Call the add() function in "un-decorated" form, skipping all
# the huey stuff:
add.call_local(3, 4)  # Returns 7.
```

It's also worth mentioning that python decorators are just syntactical sugar for wrapping a function with another function. Thus, the following two examples are equivalent:

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

# Equivalent to:
def _add(a, b):
    return a + b
```

(continues on next page)

(continued from previous page)

```
add = huey.task()(_add)
```

Task functions can be applied multiple times to a list (or iterable) of parameters using the *map()* method:

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

>>> params = [(i, i ** 2) for i in range(10)]
>>> result_group = add.map(params)
>>> result_group.get(blocking=True)
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90]
```

The Huey result-store can be used directly if you need a convenient way to cache arbitrary key/value data:

```
@huey.task()
def calculate_something():
    # By default, the result store treats get() like a pop(), so in
    # order to preserve the data so it can be read again, we specify
    # the second argument, peek=True.
    prev_results = huey.get('calculate-something.result', peek=True)
    if prev_results is None:
        # No previous results found, start from the beginning.
        data = start_from_beginning()
    else:
        # Only calculate what has changed since last time.
        data = just_what_changed(prev_results)

    # We can store the updated data back in the result store.
    huey.put('calculate-something.result', data)
    return data
```

See *Huey.get()* and *Huey.put()* for additional details.

Dynamic periodic tasks

To create periodic tasks dynamically we need to register them so that they are added to the in-memory schedule managed by the consumer's scheduler thread. Since this registry is in-memory, any dynamically defined tasks must be registered within the process that will ultimately schedule them: the consumer.

Warning: The following example will not work with the **process** worker-type option, since there is currently no way to interact with the scheduler process. When threads or greenlets are used, the worker threads share the same in-memory schedule as the scheduler thread, allowing modification to take place.

Example:

```
def dynamic_ptask(message):
    print('dynamically-created periodic task: "%s"' % message)

@huey.task()
def schedule_message(message, cron_minutes, cron_hours='*'):
```

(continues on next page)

(continued from previous page)

```
    # Create a new function that represents the application
    # of the "dynamic_ptask" with the provided message.
    def wrapper():
        dynamic_ptask(message)

    # The schedule that was specified for this task.
    schedule = crontab(cron_minutes, cron_hours)

    # Need to provide a unique name for the task. There are any number of
    # ways you can do this -- based on the arguments, etc. -- but for our
    # example we'll just use the time at which it was declared.
    task_name = 'dynamic_ptask_%s' % int(time.time())

    huey.periodic_task(schedule, name=task_name)(wrapper)
```

Assuming the consumer is running, we can now set up as many instances as we like of the "dynamic ptask" function:

```
>>> from demo import schedule_message
>>> schedule_message('I run every 5 minutes', '*/5')
<Result: task ...>
>>> schedule_message('I run between 0-15 and 30-45', '0-15,30-45')
<Result: task ...>
```

When the consumer executes the "schedule_message" tasks, our new periodic task will be registered and added to the schedule.

2.2.12 Reading more

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

- *Huey* - responsible for coordinating executable tasks and queue backends
- *Huey.task()* - decorator to indicate an executable task.
- *Result* - handle for interacting with a task.
- *Huey.periodic_task()* - decorator to indicate a task that executes at periodic intervals.
- *crontab()* - define what intervals to execute a periodic command.
- For information about managing shared resources like database connections, refer to the *shared resources* document.

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

2.3 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 it is 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.

2.3.1 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.

`-S, --simple` Use a simple log format consisting only of the time H:M:S and log message.

`-w, --workers` Number of worker threads/processes/greenlets, the default is `1` but most applications will 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 a large number of workers.
- 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.

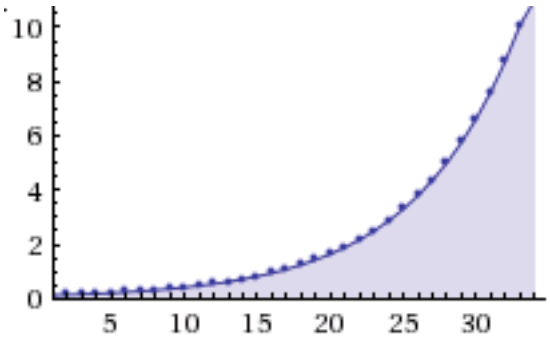
See the *Worker types* section for additional information.

-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, this is the number of seconds to wait when 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.

-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.

Examples

Running the consumer with 8 threads and a logfile for errors:

```
huey_consumer.py my.app.huey -l /var/log/app.huey.log -w 8 -q
```

Using multi-processing to run 4 worker processes.

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

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

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

Using greenlets to run 50 workers, with no health checking and a scheduler granularity of 60 seconds.

```
huey_consumer.py my.app.huey -w 50 -k greenlet -C -s 60
```

2.3.2 Worker types

The consumer consists of a main process, a scheduler, and one or more workers. These individual components all run concurrently, and Huey supports three different mechanisms to achieve this concurrency.

- *thread*, the default - uses OS threads. Due to Python's global interpreter lock, only one thread can be running at a time, but this is actually less of a limitation than it might sound. The Python runtime can intelligently switch the running thread when an I/O occurs or when a thread is idle. If the worker is CPU-bound, the runtime will pre-emptively switch threads after a given number of operations, ensuring each thread gets a chance to make progress. Threads provide a good balance of performance and memory efficiency.
- *process* - runs the scheduler and worker(s) in their own process. The main benefit over threads is the absence of the global interpreter lock, which allows CPU-bound workers to execute in parallel. Since each process maintains its own copy of the code in memory, it is likely that processes will require more memory than threads or greenlets. Processes are a good choice for tasks that perform CPU-intensive work.
- *greenlet* - runs the scheduler and worker(s) in greenlets. Requires gevent, a cooperative multi-tasking library. When a task performs an operation that would be blocking (read or write on a socket), the file descriptor is added to an event loop managed by gevent, and the scheduler will switch tasks. Since gevent uses cooperative multi-tasking, a task that is CPU-bound will not yield control to the gevent scheduler, limiting concurrency. For this reason, gevent is a good choice for tasks that perform lots of socket I/O, but may give worse performance for tasks that are CPU-bound (e.g., parsing large files, manipulating images, generating reports, etc).

When in doubt, the default setting (`thread`) is a safe choice.

Using gevent

Gevent works by monkey-patching various Python modules, such as `socket`, `ssl`, `time`, etc. In order for your application to be able to switch tasks reliably, you should apply the monkey-patch at the very beginning of your code – before anything else gets loaded.

Suppose we have defined an entrypoint for our application named `main.py`, which imports our *Huey* instance, our tasks, and the other essential parts of our application (the WSGI app, database connection, etc).

We would place the monkey-patch at the top of `main.py`, before all the other imports:

```
# main.py
from gevent import monkey; monkey.patch_all()  # Apply monkey-patch.

from .app import wsgi_app  # Import our WSGI app.
from .db import database  # Database connection.
from .queue import huey  # Huey instance for our app.
from .tasks import *  # Import all tasks, so they are discoverable.
```

To run the consumer:

```
$ huey_consumer.py main.huey -k greenlet -w 16
```

You should have a good understanding of how gevent works, its strengths and limitations, before using the greenlet worker type.

2.3.3 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 before the process exits.

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

2.3.4 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 thread/greenlet workers, to leak file descriptors. For more information, check out issue 374 and PEP 446.

2.3.5 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 (e.g a Redis list). At this point your application returns immediately, returning a *`Result`* 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 executes the task. If the task finishes, any results are stored in 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 between retries. 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.

2.3.6 Signals

The consumer will emit certain *Signals* as it executes tasks. User code can register signal handlers to respond to these events. For more information, see the *Signals* document.

2.4 Understanding how tasks are imported

Behind-the-scenes when you decorate a function with *task()* or *periodic_task()*, the function registers itself with an in-memory registry. When a task function is called, a reference is put into the queue, along with the arguments the function was called with, etc. The message is then read by the consumer, and the task function is 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 raise a *HueyException*.

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

2.4.1 Suggested organization of code

Generally, I structure things like this, which makes it very easy to avoid circular imports.

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

```
# config.py
from huey import RedisHuey

huey = RedisHuey('testing')
```

- `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 add(a, b):
    return a + b
```

- `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 add  # import any tasks / decorated functions


if __name__ == '__main__':
    result = add(1, 2)
    print('1 + 2 = %s' % result.get(blocking=True))
```

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
```

2.5 Managing shared resources

Tasks may need to make use of shared resources from the application, such as a database connection or an API client.

The simplest approach is to manage the resource explicitly. For example, Peewee database connections can be used as a context manager, so if we need to run some queries inside a task, we might write:

```
database = peewee.PostgresqlDatabase('my_app')
huey = RedisHuey()

@huey.task()
def check_comment_spam(comment_id):
    # Open DB connection at start of task, close upon exit.
    with database:
        comment = Comment.get(Comment.id == comment_id)

        if akismet.is_spam(comment.body):
            comment.is_spam = True
            comment.save()
```

Another option would be to write a decorator that acquires the shared resource before calling the task function, and then closes it after the task has finished. To make this a little simpler, Huey provides a special helper *Huey.context_task()* decorator that accepts an object implementing the context-manager API, and automatically wraps the task within the given context:

```
# Same as previous example, except we can omit the "with db" block.
@huey.context_task(db)
def check_comment_spam(comment_id):
    comment = Comment.get(Comment.id == comment_id)

    if akismet.is_spam(comment.body):
        comment.is_spam = True
        comment.save()
```

2.5.1 Startup hooks

The *Huey.on_startup()* decorator is used to register a callback that is executed once when each worker starts running. This hook provides a convenient way to initialize shared resources or perform other initializations which should happen within the context of the worker thread or process.

As an example, suppose many of our tasks will be executing queries against a Postgres database. Rather than opening and closing a connection for every task, we will instead open a connection when each worker starts. This connection may then be used by any tasks that are executed by that consumer:

```
import peewee

db = PostgresqlDatabase('my_app')

@huey.on_startup()
def open_db_connection():
    # If for some reason the db connection appears to already be open,
```

(continues on next page)

(continued from previous page)

```
    # close it first.
    if not db.is_closed():
        db.close()
    db.connect()

@huey.task()
def run_query(n):
    db.execute_sql('select pg_sleep(%s)', (n,))
    return n
```

Note: The above code works correctly because peewee stores connection state in a threadlocal. This is important if we are running the workers in threads (huey's default). Every thread will be sharing the same `PostgresqlDatabase` instance, but since the connection state is thread-local, each worker thread will see only its own connection.

2.5.2 Pre and post execute hooks

In addition to the *`on_startup()`* hook, Huey also provides decorators for registering pre- and post-execute hooks:

- *`Huey.pre_execute()`* - called right before a task is executed. The handler function should accept one argument: the task that will be executed. Pre-execute hooks have an additional feature: they can raise a special *`CancelExecution`* exception to instruct the consumer that the task should not be run.
- *`Huey.post_execute()`* - called after task has finished. The handler function should accept three arguments: the task that was executed, the return value, and the exception (if one occurred, otherwise is `None`).

Example:

```
from huey import CancelExecution

@huey.pre_execute()
def pre_execute_hook(task):
    # Pre-execute hooks are passed the task that is about to be run.

    # This pre-execute task will cancel the execution of every task if the
    # current day is Sunday.
    if datetime.datetime.now().weekday() == 6:
        raise CancelExecution('No tasks on sunday!')

@huey.post_execute()
def post_execute_hook(task, task_value, exc):
    # Post-execute hooks are passed the task, the return value (if the task
    # succeeded), and the exception (if one occurred).
    if exc is not None:
        print('Task "%s" failed with error: %s!' % (task.id, exc))
```

Note: Printing the error message is redundant, as the huey logger already logs any unhandled exceptions raised by a task, along with a traceback. These are just examples.

2.6 Signals

The consumer will send various signals as it processes tasks. Callbacks can be registered as signal handlers, and will be called synchronously by the consumer process.

The following signals are implemented by Huey:

- `SIGNAL_CANCELED`: task was canceled due to a pre-execute hook raising a *`CancelExecution`* exception.
- `SIGNAL_COMPLETE`: task has been executed successfully.
- `SIGNAL_ERROR`: task failed due to an unhandled exception.
- `SIGNAL_EXECUTING`: task is about to be executed.
- `SIGNAL_LOCKED`: failed to acquire lock, aborting task.
- `SIGNAL_RETRYING`: task failed, but will be retried.
- `SIGNAL_REVOKED`: task is revoked and will not be executed.
- `SIGNAL_SCHEDULED`: task is not yet ready to run and has been added to the schedule for future execution.

When a signal handler is called, it will be called with the following arguments:

- `signal`: the signal name, e.g. `'executing'`.
- `task`: the *`Task`* instance.

The following signals will include additional arguments:

- `SIGNAL_ERROR`: includes a third argument `exc`, which is the `Exception` that was raised while executing the task.

To register a signal handler, use the *`Huey.signal()`* method:

```
@huey.signal()
def all_signal_handler(signal, task, exc=None):
    # This handler will be called for every signal.
    print('%s - %s' % (signal, task.id))

@huey.signal(SIGNAL_ERROR, SIGNAL_LOCKED, SIGNAL_CANCELED, SIGNAL_REVOKED)
def task_not_executed_handler(signal, task, exc=None):
    # This handler will be called for the 4 signals listed, which
    # correspond to error conditions.
    print('[%s] %s - not executed' % (signal, task.id))

@huey.signal(SIGNAL_COMPLETE)
def task_success(signal, task):
    # This handle will be called for each task that completes successfully.
    pass
```

Signal handlers can be unregistered using *`Huey.disconnect_signal()`*.

```
# Disconnect the "task_success" signal handler.
huey.disconnect_signal(task_success)

# Disconnect the "task_not_executed_handler", but just from
# handling SIGNAL_LOCKED.
huey.disconnect_signal(task_not_executed_handler, SIGNAL_LOCKED)
```

2.6.1 Examples

We'll use the following tasks to illustrate how signals may be sent:

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

@huey.task(retries=2, retry_delay=10)
def flaky_task():
    if random.randint(0, 1) == 0:
        raise ValueError('uh-oh')
    return 'OK'
```

Here is a simple example of a task execution we would expect to succeed:

```
>>> result = add(1, 2)
>>> result.get(blocking=True)
```

The consumer would send the following signals:

- SIGNAL_EXECUTING - the task has been dequeued and will be executed.
- SIGNAL_COMPLETE - the task has finished successfully.

Here is an example of scheduling a task for execution after a short delay:

```
>>> result = add.schedule((2, 3), delay=10)
>>> result(True)  # same as result.get(blocking=True)
```

The following signals would be sent:

- SIGNAL_SCHEDULED - the task is not yet ready to run, so it has been added to the schedule.
- After 10 seconds, the consumer will run the task and send the SIGNAL_EXECUTING signal.
- SIGNAL_COMPLETE.

Here is an example that may fail, in which case it will be retried automatically with a delay of 10 seconds.

```
>>> result = flaky_task()
>>> try:
...     result.get(blocking=True)
... except TaskException:
...     result.reset()
...     result.get(blocking=True)  # Try again if first time fails.
...
```

Assuming the task failed the first time and succeeded the second time, we would see the following signals being sent:

- SIGNAL_EXECUTING - the task is being executed.
- SIGNAL_ERROR - the task raised an unhandled exception.
- SIGNAL_RETRYING - the task will be retried.
- SIGNAL_SCHEDULED - the task has been added to the schedule for execution in ~10 seconds.
- SIGNAL_EXECUTING - second try running task.
- SIGNAL_COMPLETE - task succeeded.

What happens if we revoke the add() task and then attempt to execute it:

```
>>> add.revoke()
>>> res = add(1, 2)
```

The following signal will be sent:

- `SIGNAL_REVOKED` - this is sent before the task enters the "executing" state. When a task is revoked, no other signals will be sent.

Performance considerations

Signal handlers are executed **synchronously** by the consumer as it processes tasks. It is important to use care when implementing signal handlers, as one slow signal handler can impact the overall responsiveness of the consumer.

For example, if you implement a signal handler that posts some data to REST API, everything might work fine until the REST API goes down or stops being responsive – which will cause the signal handler to block, which then prevents the consumer from moving on to the next task.

Another consideration is the *management of shared resources* that may be used by signal handlers, such as database connections or open file handles. Signal handlers are called by the consumer workers, which (depending on how you are running the consumer) may be separate processes, threads or greenlets. As a result, care should be taken to ensure proper initialization and cleanup of any resources you plan to use in signal handlers.

2.7 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.py` module.

2.7.1 Huey types

Implementations of *Huey* which handle task and result persistence.

Note: See the documentation for *Huey* for the list of initialization parameters common to all Huey implementations.

---

class RedisHuey

Huey that utilizes redis for queue and result storage. Requires redis-py.

Commonly-used keyword arguments for storage configuration:

Parameters

- **blocking** (*bool*) – Use blocking-pop when reading from the queue (as opposed to polling). Default is true.
- **connection_pool** – a redis-py `ConnectionPool` instance.
- **url** – url for Redis connection.
- **host** – hostname of the Redis server.
- **port** – port number.

- **password** – password for Redis.
- **db** (*int*) – Redis database to use (typically 0-15, default is 0).

The redis-py documentation contains the complete list of arguments supported by the Redis client.

Note: RedisHuey does not support task priorities. If you wish to use task priorities with Redis, use *PriorityRedisHuey*.

RedisHuey uses a Redis LIST to store the queue of pending tasks. Redis lists are a natural fit, as they offer O(1) append and pop from either end of the list. Redis also provides blocking-pop commands which allow the consumer to react to a new message as soon as it is available without resorting to polling.

See also:

RedisStorage

class PriorityRedisHuey

Huey that utilizes redis for queue and result storage. Requires redis-py. Accepts the same arguments as *RedisHuey*.

PriorityRedisHuey supports *task priorities*, and requires Redis **5.0 or newer**.

PriorityRedisHuey uses a Redis SORTED SET to store the queue of pending tasks. Sorted sets consist of a unique value and a numeric score. In addition to being sorted by numeric score, Redis also orders the items within the set lexicographically. Huey takes advantage of these two characteristics to implement the priority queue. Redis 5.0 added a new command, ZPOPMIN, which pops the lowest-scoring item from the sorted set (and BZPOPMIN, the blocking variety).

class RedisExpireHuey

Identical to *RedisHuey* except for the way task result values are stored. RedisHuey keeps all task results in a Redis hash, and whenever a task result is read (via the result handle), it is also removed from the result hash. This is done to prevent the task result storage from growing without bound. Additionally, using a Redis hash for all results helps avoid cluttering up the Redis keyspace and utilizes less RAM for storing the keys themselves.

`RedisExpireHuey` uses a different approach: task results are stored in ordinary Redis keys with a special prefix. Result keys are then given a time-to-live, and will be expired automatically by the Redis server. This removes the necessity to remove results from the result store after they are read once.

Commonly-used keyword arguments for storage configuration:

Parameters

- **expire_time** (*int*) – Expire time in seconds, default is 86400 (1 day).
- **blocking** (*bool*) – Use blocking-pop when reading from the queue (as opposed to polling). Default is true.
- **connection_pool** – a redis-py `ConnectionPool` instance.
- **url** – url for Redis connection.
- **host** – hostname of the Redis server.
- **port** – port number.
- **password** – password for Redis.
- **db** (*int*) – Redis database to use (typically 0-15, default is 0).

class SqliteHuey

Huey that utilizes sqlite3 for queue and result storage. Only requirement is the standard library `sqlite3` module.

Commonly-used keyword arguments:

Parameters

- **filename** (*str*) – filename for database, defaults to 'huey.db'.
- **cache_mb** (*int*) – megabytes of memory to allow for sqlite page-cache.
- **fsync** (*bool*) – use durable writes. Slower but more resilient to corruption in the event of sudden power loss. Defaults to false.

SqliteHuey fully supports task priorities.

See also:

SqliteStorage

class MemoryHuey

Huey that uses in-memory storage. Only should be used when testing or when using `immediate` mode. MemoryHuey fully supports task priorities.

class FileHuey

Huey that uses the file-system for storage. Should not be used in high-throughput, highly-concurrent environments, as the *FileStorage* utilizes exclusive locks around all file-system operations.

Parameters

- **path** (*str*) – base-path for huey data (queue tasks, schedule and results will be stored in sub-directories of this path).
- **levels** (*int*) – number of levels in result-file directory structure to ensure the results directory does not contain an unmanageable number of files.
- **use_thread_lock** (*bool*) – use the standard lib `threading.Lock` instead of a lock-file for file-system operations. This should only be enabled when using the greenlet or thread consumer worker models.

FileHuey fully supports task priorities.

2.7.2 Huey object

class Huey(*name='huey'*, *results=True*, *store_none=False*, *utc=True*, *immediate=False*, *serializer=None*, *compression=False*, *use_zlib=False*, *immediate_use_memory=True*, *storage_kwargs*)

Parameters

- **name** (*str*) – the name of the task queue, e.g. your application's name.
- **results** (*bool*) – whether to store task results.
- **store_none** (*bool*) – whether to store `None` in the result store.
- **utc** (*bool*) – use UTC internally, convert naive datetimes from local time to UTC (if local time is other than UTC).
- **immediate** (*bool*) – useful for debugging; causes tasks to be executed synchronously in the application.
- **serializer** (*Serializer*) – serializer implementation for tasks and result data. The default implementation uses `pickle`.
- **compression** (*bool*) – compress tasks and result data.
- **use_zlib** (*bool*) – use zlib for compression instead of gzip.

- **immediate_use_memory** (*bool*) – automatically switch to a local in-memory storage backend whenever immediate-mode is enabled.
- **storage_kwargs** – arbitrary keyword arguments that will be passed to the storage backend for additional configuration.

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.

Example usage:

```
# demo.py
from huey import RedisHuey

# Create a huey instance.
huey = RedisHuey('my-app')

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

@huey.periodic_task(crontab(minute='0', hour='2'))
def nightly_report():
    generate_nightly_report()
```

To run the consumer with 4 worker threads:

```
$ huey_consumer.py demo.huey -w 4
```

To add two numbers, the "huey" way:

```
>>> from demo import add_numbers
>>> res = add_numbers(1, 2)
>>> res(blocking=True)  # Blocks until result is available.
3
```

To test huey without using a consumer, you can use "immediate" mode. Immediate mode follows all the same code paths as Huey does when running the consumer process, but does so synchronously within the application.

```
>>> from demo import add_numbers, huey
>>> huey.immediate = True  # Tasks executed immediately.
>>> res = add_numbers(2, 3)
>>> res()
5
```

immediate

The `immediate` property is used to enable and disable *immediate mode*. When immediate mode is enabled, task-decorated functions are executed synchronously by the caller, making it very useful for development and testing. Calling a task function still returns a *Result* handle, but the task itself is executed immediately.

By default, when immediate mode is enabled, Huey will switch to using in-memory storage. This is to help prevent accidentally writing to a live Redis server while testing. To disable this functionality, specify `immediate_use_memory=False` when initializing *Huey*.

Enabling immediate mode:

```
huey = RedisHuey()

# Enable immediate mode. Tasks now executed synchronously.
# Additionally, huey will now use in-memory storage.
huey.immediate = True

# Disable immediate mode. Tasks will now be enqueued in a Redis
# queue.
huey.immediate = False
```

Immediate mode can also be specified when your Huey instance is created:

```
huey = RedisHuey(immediate=True)
```

task(*retries=0*, *retry_delay=0*, *priority=None*, *context=False*, *name=None*, ***kwargs*)

Parameters

- **retries** (*int*) – number of times to retry the function if an unhandled exception occurs when it is executed.
- **retry_delay** (*int*) – number of seconds to wait between retries.
- **priority** (*int*) – priority assigned to task, higher numbers are processed first by the consumer when there is a backlog.
- **context** (*bool*) – when the task is executed, include the *Task* instance as a keyword argument.
- **name** (*str*) – name for this task. If not provided, Huey will default to using the module name plus function name.
- **kwargs** – arbitrary key/value arguments that are passed to the *TaskWrapper* instance.

Returns a *TaskWrapper* that wraps the decorated function and exposes a number of APIs for enqueueing the task.

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. Return a *Result* handle, which can be used to check the result of the task function, revoke the task (assuming it hasn't started yet), reschedule the task, and more.

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

For more information, see the *immediate mode* section of the guide.

The `task()` decorator returns a *TaskWrapper*, which implements special methods for enqueueing the decorated function. These methods are used to *schedule()* the task to run in the future, chain tasks to form a pipeline, and more.

Example:

```
from huey import RedisHuey

huey = RedisHuey()

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

Whenever the `add()` function is called, the actual execution will occur when the consumer dequeues the message.

```
>>> res = add(1, 2)
>>> res
<Result: task 6b6f36fc-da0d-4069-b46c-c0d4ccff1df6>
>>> res()
3
```

To further illustrate this point:

```
@huey.task()
def slow(n):
    time.sleep(n)
    return n
```

Calling the `slow()` task will return immediately. We can, however, block until the task finishes by waiting for the result:

```
>>> res = slow(10)  # Returns immediately.
>>> res(blocking=True)  # Block until task finishes, ~10s.
10
```

Note: The return value of any calls to the decorated function depends on whether the *Huey* instance is configured to store the results of tasks (`results=True` is the default). When the result store is disabled, calling a task-decorated function will return `None` instead of a result handle.

In some cases, it may be useful to receive the *Task* instance itself as an argument.

```
@huey.task(context=True)  # Include task as an argument.
def print_a_task_id(message, task=None):
    print('%s: %s' % (message, task.id))


print_a_task_id('hello')
print_a_task_id('goodbye')
```

This would cause the consumer would print something like:

```
hello: e724a743-e63e-4400-ac07-78a2fa242b41
goodbye: 606f36fc-da0d-4069-b46c-c0d4ccff1df6
```

Note: When using other decorators on task functions, make sure that you understand when they will be evaluated. In the following example the decorator `a` will be evaluated in the calling process, while `b` will be evaluated in the worker process.

```
@a
@huey.task()
@b
def task():
    pass
```

For more information, see *TaskWrapper*, *Task*, and *Result*.

periodic_task(*validate_datetime*, *retries=0*, *retry_delay=0*, *priority=None*, *context=False*, *name=None*, ***kwargs*)

Parameters

- **validate_datetime** (*function*) – function which accepts a `datetime` instance and returns whether the task should be executed at the given time.
- **retries** (*int*) – number of times to retry the function if an unhandled exception occurs when it is executed.
- **retry_delay** (*int*) – number of seconds to wait in-between retries.
- **priority** (*int*) – priority assigned to task, higher numbers are processed first by the consumer when there is a backlog.
- **context** (*bool*) – when the task is executed, include the *Task* instance as a parameter.
- **name** (*str*) – name for this task. If not provided, Huey will default to using the module name plus function name.
- **kwargs** – arbitrary key/value arguments that are passed to the *TaskWrapper* instance.

Returns a *TaskWrapper* that wraps the decorated function and exposes a number of APIs for enqueueing the task.

The `periodic_task()` decorator marks a function for automatic execution by the consumer *at a specific interval*, like `cron`.

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 once per minute, giving it the same granularity as the `cron`.

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. The following task will be executed every three hours, on the hour:

```
from huey import crontab
from huey import RedisHuey

huey = RedisHuey()

@huey.periodic_task(crontab(minute='0', hour='*/3'))
def update_feeds():
    for feed in my_list_of_feeds:
        fetch_feed_data(feed)
```

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.

Like *task()*, the periodic task decorator adds helpers to the decorated function. These helpers allow you to *revoke()* and *restore()* the periodic task, enabling you to pause it or prevent its execution. For more information, see *TaskWrapper*.

Note: The result (return value) of a periodic task is not stored in the result store. This is primarily due to the fact that there is not an obvious way one would read such results, since the invocation of the periodic task happens inside the consumer scheduler. As such, there is no task result handle which the user could use to read the result. To store the results of periodic tasks, you will need to use your own storage or use the storage APIs directly:

```
@huey.periodic_task(crontab(minute='*/10'))
def my_task():
    # do some work...
    do_something()

    # Manually store some data in the result store.
    huey.put('my-task', some_data_to_store)
```

More info:

- *Huey.put()*
- *Huey.get()*

---

context_task(*obj*, *retries=0*, *retry_delay=0*, *context=False*, *name=None*, ***kwargs*)

Parameters

- **obj** – object that implements the context-manager APIs.
- **as_argument** (*bool*) – pass the context-manager object into the decorated task as the first argument.
- **retries** (*int*) – number of times to retry the function if an unhandled exception occurs when it is executed.
- **retry_delay** (*int*) – number of seconds to wait in-between retries.
- **context** (*bool*) – when the task is executed, include the *Task* instance as a parameter.
- **name** (*str*) – name for this task. If not provided, Huey will default to using the module name plus function name.
- **kwargs** – arbitrary key/value arguments that are passed to the *TaskWrapper* instance.

Returns a *TaskWrapper* that wraps the decorated function and exposes a number of APIs for enqueueing the task.

This is an extended implementation of the *Huey.task()* decorator, which wraps the decorated task in a `with obj:` block. Roughly equivalent to:

```
db = PostgresqlDatabase(...)

@huey.task()
def without_context_task(n):
    with db:
        do_something(n)
```

(continues on next page)

(continued from previous page)

```
@huey.context_task(db)
def with_context_task(n):
    return do_something(n)
```

pre_execute(*name=None*)

Parameters **name** – (optional) name for the hook.

Returns a decorator used to wrap the actual pre-execute function.

Decorator for registering a pre-execute hook. The callback will be executed before the execution of every task. 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.

Usage:

```
@huey.pre_execute()
def my_pre_execute_hook(task):
    if datetime.datetime.now().weekday() == 6:
        raise CancelExecution('Sunday -- no work will be done.')
```

unregister_pre_execute(*name_or_fn*)

Parameters **name_or_fn** – the name given to the pre-execute hook, or the function object itself.

Returns boolean

Unregister the specified pre-execute hook.

post_execute(*name=None*)

Parameters **name** – (optional) name for the hook.

Returns a decorator used to wrap the actual post-execute function.

Register a post-execute hook. The callback will be executed after the execution of every task. 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.

Usage:

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

unregister_post_execute(*name_or_fn*)

Parameters **name_or_fn** – the name given to the post-execute hook, or the function object itself.

Returns boolean

Unregister the specified post-execute hook.

on_startup(*name=None*)

Parameters **name** – (optional) name for the hook.

Returns a decorator used to wrap the actual on-startup function.

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 shared 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.

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()
    # ...
```

unregister_on_startup(*name_or_fn*)

Parameters **name_or_fn** – the name given to the on-startup hook, or the function object itself.

Returns boolean

Unregister the specified on-startup hook.

on_shutdown(*name=None*)

Parameters **name** – (optional) name for the hook.

Returns a decorator used to wrap the actual on-shutdown function.

Register a shutdown hook. The callback will be executed by a worker immediately before it goes offline. Uncaught exceptions will be logged but will have no other effect on the overall shutdown of the worker.

The callback function must not accept any parameters.

This API is provided to simplify cleaning-up shared resources.

unregister_on_shutdown(*name_or_fn*)

Parameters **name_or_fn** – the name given to the on-shutdown hook, or the function object itself.

Returns boolean

Unregister the specified on-shutdown hook.

signal(**signals*)

Parameters **signals** – zero or more signals to handle.

Returns a decorator used to wrap the actual signal handler.

Attach a signal handler callback, which will be executed when the specified signals are sent by the consumer. If no signals are listed, then the handler will be bound to **all** signals. The list of signals and additional information can be found in the *Signals* documentation.

Example:

```
from huey.signals import SIGNAL_ERROR, SIGNAL_LOCKED

@huey.signal(SIGNAL_ERROR, SIGNAL_LOCKED)
def task_not_run_handler(signal, task, exc=None):
    # Do something in response to the "ERROR" or "LOCEKD" signals.
    # Note that the "ERROR" signal includes a third parameter,
    # which is the unhandled exception that was raised by the task.
    # Since this parameter is not sent with the "LOCKED" signal, we
    # provide a default of ``exc=None``.
    pass
```

disconnect_signal(*receiver*, **signals*)

Parameters

- **receiver** – the signal handling function to disconnect.
- **signals** – zero or more signals to stop handling.

Disconnect the signal handler from the provided signals. If no signals are provided, then the handler is disconnected from any signals it may have been connected to.

enqueue(*task*)

Parameters **task** (*Task*) – task instance to enqueue.

Returns *Result* handle for the given task.

Enqueue the given task. When the result store is enabled (default), the return value will be a *Result* handle which provides access to the result of the task execution (as well as other things).

If the task specifies another task to run on completion (see *Task.then()*), the return value will be a *ResultGroup*, which encapsulates a list of individual *Result* instances for the given pipeline.

Note: Unless you are executing a pipeline of tasks, it should not be necessary to use the *enqueue()* method directly. 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.

---

revoke(*task*, *revoke_until=None*, *revoke_once=False*)

See also:

Use *Result.revoke()* instead.

revoke_by_id(*task_id*, *revoke_until=None*, *revoke_once=False*)

Parameters

- **task_id** (*str*) – task instance id.
- **revoke_until** (*datetime*) – optional expiration date for revocation.
- **revoke_once** (*bool*) – revoke once and then re-enable.

Revoke a *Task* instance using the task id.

revoke_all(*task_class*, *revoke_until=None*, *revoke_once=False*)

See also:

Use *TaskWrapper.revoke()* instead.

restore(*task*)

See also:

Use *Result.restore()* instead.

restore_by_id(*task_id*)

Parameters **task_id** (*str*) – task instance id.

Returns boolean indicating success.

Restore a *Task* instance using the task id. Returns boolean indicating if the revocation was successfully removed.

restore_all(*task_class*)

See also:

Use *TaskWrapper.restore()* instead.

is_revoked(*task*, *timestamp=None*)

See also:

For task instances, use *Result.is_revoked()*.

For task functions, use *TaskWrapper.is_revoked()*.

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

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*) – when set to `True`, this parameter ensures that the task result will be preserved after having been successfully retrieved. Ordinarily, Huey will discard results after they have been read, to prevent the result store from growing without bounds.

Attempts 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 the result, 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 *HueyException* will be raised.

See also:

Result - the *result()* method is simply a wrapper that creates a `Result` object and calls its *get()* method.

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 *Result.get()* methods). If you intend to access the task result multiple times, you must specify `preserve=True` when calling these methods.

lock_task(*lock_name*)

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

Returns `TaskLock` instance, which can be used as a decorator or context-manager.

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 *TaskLockedException* is raised, which is handled by the consumer. The task will not be executed and a `SIGNAL_LOCKED` will be sent. If the task is configured to be retried, then it will be retried normally.

Examples:

```
@huey.periodic_task(crontab(minute='*/5'))
@huey.lock_task('reports-lock')  # Goes *after* the task decorator.
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()
```

put(*key*, *value*)

Parameters

- **key** – key for data
- **value** – arbitrary data to store in result store.

Store a value in the result-store under the given key.

get(*key*, *peek=False*)

Parameters

- **key** – key to read
- **peek** (*bool*) – non-destructive read

Read a value from the result-store at the given key. By default reads are destructive. To preserve the value for additional reads, specify `peek=True`.

pending(*limit=None*)

Parameters **limit** (*int*) – optionally limit the number of tasks returned.

Returns a list of *Task* instances waiting to be run.

scheduled(*limit=None*)

Parameters **limit** (*int*) – optionally limit the number of tasks returned.

Returns a list of *Task* instances that are scheduled to execute at some time in the future.

all_results()

Returns a dict of task-id to the serialized result data for all key/value pairs in the result store.

__len__()

Return the number of items currently in the queue.

class TaskWrapper(*huey*, *func*, *retries=None*, *retry_delay=None*, *context=False*, *name=None*, *task_base=None*, ***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.
- **context** (*bool*) – when the task is executed, include the *Task* instance as a parameter.
- **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 *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. The *Huey.task()* and *Huey.periodic_task()* decorators will create the appropriate TaskWrapper automatically.

schedule(*args=None*, *kwargs=None*, *eta=None*, *delay=None*)

Parameters

- **args** (*tuple*) – arguments for decorated function.
- **kwargs** (*dict*) – keyword arguments for decorated function.
- **eta** (*datetime*) – the time at which the function should be executed.
- **delay** (*int*) – number of seconds to wait before executing function.

Returns a *Result* handle for the task.

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

```
import datetime

one_hour = datetime.datetime.now() + datetime.timedelta(hours=1)

# Schedule the task to be run in an hour. It will be called with
# three arguments.
res = check_feeds.schedule(args=(url1, url2, url3), eta=one_hour)

# Equivalent, but uses delay rather than eta.
res = check_feeds.schedule(args=(url1, url2, url3), delay=3600)
```

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

Parameters

- **revoke_until** (*datetime*) – Automatically restore the task after the given datetime.
- **revoke_once** (*bool*) – Revoke the next execution of the task and then automatically restore.

Revoking a task will prevent any instance of the given task from executing. When no parameters are provided the function will not execute again until *TaskWrapper.restore()* is called.

This function can be called multiple times, but each call will supercede any restrictions from the previous revocation.

```
# Skip the next execution
send_emails.revoke(revoke_once=True)

# Prevent any invocation from executing.
send_emails.revoke()

# Prevent any invocation for 24 hours.
tomorrow = datetime.datetime.now() + datetime.timedelta(days=1)
send_emails.revoke(revoke_until=tomorrow)
```

is_revoked(*timestamp=None*)

Parameters **timestamp** (*datetime*) – If provided, checks whether task is revoked with respect to the given timestamp.

Returns bool indicating whether task is revoked.

Check whether the given task is revoked.

restore()

Returns bool indicating whether a previous revocation rule was found and removed successfully.

Removes a previous task revocation, if one was configured.

call_local()

Call the `@task`-decorated function, bypassing all Huey-specific logic. In other words, `call_local()` provides access to the underlying user-defined function.

```
>>> add.call_local(1, 2)
3
```

s(**args*, ***kwargs*)

Parameters

- **args** – Arguments for task function.
- **kwargs** – Keyword arguments for task function.
- **priority** (*int*) – assign priority override to task, higher numbers are processed first by the consumer when there is a backlog.

Returns a *Task* instance representing the execution of the task function with the given arguments.

Create a *Task* instance representing the invocation of the task function with the given arguments and keyword-arguments.

Note: The returned 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)
```

Typically, one will only 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.
```

(continues on next page)

(continued from previous page)

```
print(results.get(blocking=True))

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

map(*it*)

Parameters **it** – a list, tuple or generic iterable that contains the arguments for a number of individual task executions.

Returns a *ResultGroup* encapsulating the individual *Result* handlers for the task executions.

Note: The iterable should be a list of argument tuples which will be passed to the task function.

Example:

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

rg = add.map([(i, i * i) for i in range(10)])

# Resolve all results.
rg.get(blocking=True)

# [0, 2, 6, 12, 20, 30, 42, 56, 72, 90]
```

class Task(*args=None*, *kwargs=None*, *id=None*, *eta=None*, *retries=None*, *retry_delay=None*, *on_complete=None*, *on_error=None*)

Parameters

- **args** (*tuple*) – arguments for the function call.
- **kwargs** (*dict*) – keyword arguments for the function call.
- **id** (*str*) – unique id, defaults to a UUID if not provided.
- **eta** (*datetime*) – time at which task should be executed.
- **retries** (*int*) – automatic retry attempts.
- **retry_delay** (*int*) – seconds to wait before retrying a failed task.
- **priority** (*int*) – priority assigned to task, higher numbers are processed first by the consumer when there is a backlog.
- **on_complete** (Task) – Task to execute upon completion of this task.
- **on_error** (Task) – Task to execute upon failure / error.

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

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

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 Task 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))

result_group = huey.enqueue(pipeline)

print(result_group.get(blocking=True))

# [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))
result_group = huey.enqueue(pipe)

print(result_group.get(blocking=True))
# [(2, 1), (3, 2), (5, 3)]
```

error(*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 *error()* method is similar to the *then()* method, which is used to construct a task pipeline, except the `error()` task will only be called in the event of an unhandled exception in the parent task.

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.

2.7.3 Result

class Result(*huey*, *task*)

Although you will probably never instantiate an `Result` object yourself, they are returned whenever you execute a task-decorated function, or schedule a task for execution. The `Result` object 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 original caller to retrieve it.

Getting results from tasks is very simple:

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

>>> res = add(1, 2)
>>> res  # what is "res" ?
<Result: task 6b6f36fc-da0d-4069-b46c-c0d4ccff1df6>
```

(continues on next page)

(continued from previous page)

```
>>> res()  # Fetch the result of this task.
3
```

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

```
>>> res = add(100, 200)  # Imagine this is very slow.
>>> res.get()  # Data is not ready, so None is returned.

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

>>> res(blocking=True)  # No timeout, will block until it gets data.
300
```

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',)
```

id

Returns the unique id of the corresponding task.

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

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.

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 *HueyException* exception will be raised.

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

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

revoke(*revoke_once=True*)

Parameters **revoke_once** (*bool*) – revoke only once.

Revoke the given task. Unless it is in the process of executing, the task will be discarded without being executed.

```
one_hour = datetime.datetime.now() + datetime.timedelta(hours=1)

# Run this command in an hour
res = add.schedule((1, 2), eta=one_hour)

# 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, via a call to *TaskWrapper.revoke()*, 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:

TaskWrapper.is_revoked().

reschedule(*eta=None*, *delay=None*)

Parameters

- **eta** (*datetime*) – execute function at the given time.
- **delay** (*int*) – execute function after specified delay in seconds.

Returns *Result* handle for the new task.

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.

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).

class ResultGroup
A `ResultGroup` will be returned when you enqueue a task pipeline or if you use the *TaskWrapper.map()* method. It is a simple wrapper around a number of individual *Result()* instances, and provides a convenience API for fetching the results in bulk.

get(***kwargs*)
Call `get()` on each individual `Result()` instance in the group and returns a list of return values. Any keyword arguments are passed along.

2.7.4 Serializer

class Serializer(*compression=False*, *compression_level=6*, *use_zlib=False*)

Parameters

- **compression** (*bool*) – use gzip compression
- **compression_level** (*int*) – 0 for least, 9 for most.
- **use_zlib** (*bool*) – use zlib for compression instead of gzip.

The Serializer class implements a simple interface that can be extended to provide your own serialization format. The default implementation uses `pickle`.

To override, the following methods should be implemented. Compression is handled transparently elsewhere in the API.

_serialize(*data*)

Parameters **data** – arbitrary Python object to serialize.

Rtype bytes

_deserialize(*data*)

Parameters **data** (*bytes*) – serialized data.

Returns the deserialized object.

2.7.5 Exceptions

class HueyException
General exception class.

class ConfigurationError
Raised when Huey encounters a configuration problem.

class TaskLockedException
Raised by the consumer when a task lock cannot be acquired.

class CancelExecution
Should be raised by user code within a `pre_execute()` hook to signal to the consumer that the task shall be cancelled.

class RetryTask
Raised by user code from within a `task()` function to force a retry. When this exception is raised, the task will be retried irrespective of whether it is configured with automatic retries.

class TaskException
General exception raised by `Result` handles when reading the result of a task that failed due to an error.

2.7.6 Storage

Huey comes with several built-in storage implementations:

class RedisStorage(*name='huey'*, *blocking=True*, *read_timeout=1*, *connection_pool=None*, *url=None*, *client_name=None*, ***connection_params*)

Parameters

- **blocking** (*bool*) – Use blocking-pop when reading from the queue (as opposed to polling). Default is true.
- **read_timeout** – Timeout to use when performing a blocking pop, default is 1 second.
- **connection_pool** – a redis-py `ConnectionPool` instance.
- **url** – url for Redis connection.
- **client_name** – name used to identify Redis clients used by Huey.

Additional keyword arguments will be passed directly to the Redis client constructor. See the redis-py documentation for the complete list of arguments supported by the Redis client.

class RedisExpireStorage(*name='huey'*, *expire_time=86400*, *blocking=True*, *read_timeout=1*, *connection_pool=None*, *url=None*, *client_name=None*, ***connection_params*)

Parameters **expire_time** (*int*) – TTL for results of individual tasks.

Subclass of *RedisStorage* that implements the result store APIs using normal Redis keys with a TTL, so that unread results will automatically be cleaned-up. *RedisStorage* uses a *HASH* for the result store, which has the benefit of keeping the Redis keyspace orderly, but which comes with the downside that unread task results can build up over time. This storage implementation trades keyspace sprawl for automatic clean-up.

class PriorityRedisStorage(*name='huey'*, *blocking=True*, *read_timeout=1*, *connection_pool=None*, *url=None*, *client_name=None*, ***connection_params*)

Parameters

- **blocking** (*bool*) – Use blocking-zpopmin when reading from the queue (as opposed to polling). Default is true.
- **read_timeout** – Timeout to use when performing a blocking pop, default is 1 second.
- **connection_pool** – a redis-py `ConnectionPool` instance.
- **url** – url for Redis connection.
- **client_name** – name used to identify Redis clients used by Huey.

Redis storage that uses a different data-structure for the task queue in order to support task priorities.

Additional keyword arguments will be passed directly to the Redis client constructor. See the redis-py documentation for the complete list of arguments supported by the Redis client.

> **Warning:** This storage engine requires Redis 5.0 or newer.

class PriorityRedisExpireStorage(*name='huey'*, *expire_time=86400*, *...*)

Parameters **expire_time** (*int*) – TTL for results of individual tasks.

Combination of *PriorityRedisStorage*, which supports task priorities, and *RedisExpireStorage*, which stores task results as top-level Redis keys in order set a TTL so that unread results are automatically cleaned-up.

class SqliteStorage(*filename='huey.db'*, *name='huey'*, *cache_mb=8*, *fsync=False*, *timeout=5*, *strict_fifo=False*, ***kwargs*)

Parameters

- **filename** (*str*) – sqlite database filename.
- **cache_mb** (*int*) – sqlite page-cache size in megabytes.
- **fsync** (*bool*) – if enabled, all writes to the Sqlite database will be synchonized. This provides greater safety from database corruption in the event of sudden power-loss.
- **journal_mode** (*str*) – sqlite journaling mode to use. Defaults to using write-ahead logging, which enables readers to coexist with a single writer.
- **timeout** (*int*) – busy timeout (in seconds), amount of time to wait to acquire the write lock when another thread / connection holds it.
- **strict_fifo** (*bool*) – ensure that the task queue behaves as a strict FIFO. By default, Sqlite may reuse rowids for deleted tasks, which can cause tasks to be run in a different order than the order in which they were enqueued.
- **kwargs** – Additional keyword arguments passed to the `sqlite3` connection constructor.

class FileStorage(*name*, *path*, *levels=2*, *use_thread_lock=False*)

Parameters

- **name** (*str*) – (unused by the file storage API)
- **path** (*str*) – directory path used to store task results. Will be created if it does not exist.
- **levels** (*int*) – number of levels in cache-file directory structure to ensure a given directory does not contain an unmanageable number of files.
- **use_thread_lock** (*bool*) – use the standard lib `threading.Lock` instead of a lockfile. Note: this should only be enabled when using the greenlet or thread consumer worker models.

The *FileStorage* implements a simple file-system storage layer. This storage class should not be used in high-throughput, highly-concurrent environments, as it utilizes exclusive locks around all file-system operations. This is done to prevent race-conditions when reading from the file-system.

class MemoryStorage

In-memory storage engine for use when testing or developing. Designed for use with *immediate mode*.

class BlackHoleStorage

Storage class that discards all data written to it, and thus always appears to be empty. Intended for testing only.

class BaseStorage(*name='huey'*, ***storage_kwargs*)

enqueue(*data*, *priority=None*)

dequeue()

queue_size()

enqueued_items(*limit=None*)

flush_queue()

add_to_schedule(*data*, *timestamp*)

read_schedule(*timestamp*)

schedule_size()

scheduled_items(*limit=None*)

flush_schedule()

put_data(*key*, *value*)

peek_data(*key*)

pop_data(*key*)

put_if_empty(*key*, *value*)

has_data_for_key(*key*)

result_store_size()

result_items()

flush_results()

2.8 Huey Extensions

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

2.8.1 Mini-Huey

`MiniHuey` provides a very lightweight huey-like API that may be useful for certain applications. The `MiniHuey` consumer runs inside a greenlet in your main application process. This means there is no separate consumer process to manage, nor 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.

MiniHuey may be useful if:

- Your application is a WSGI application.
- Your tasks do stuff like check for spam, send email, make requests to web-based APIs, query a database server.
- You do not need automatic retries, persistence for your message queue, dynamic task revocation.
- You wish to keep things nice and simple and don't want the overhead of additional process(es) to manage.

MiniHuey may be a bad choice if:

- Your application is incompatible with gevent (e.g. uses asyncio).
- Your tasks do stuff like process large files, crunch numbers, parse large XML or JSON documents, or other CPU or disk-intensive work.
- You need a persistent store for messages and results, so the consumer can be restarted without losing any unprocessed messages.

If you are not sure, then you should probably not use *MiniHuey*. Use the regular *Huey* instead.

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.

task([*validate_func=None*])

Task decorator similar to *Huey.task()* or *Huey.periodic_task()*. For tasks that should be scheduled automatically at regular intervals, simply provide a suitable *crontab()* definition.

The decorated task will gain a `schedule()` method which can be used like the *TaskWrapper.schedule()* method.

Examples task declarations:

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

huey = MiniHuey()

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

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

Example usage. Running tasks and getting results work about the same as regular Huey:

```
# Executes the task asynchronously in a new greenlet.
result = fetch_url('https://google.com/')

# Wait for the task to finish.
html = result.get()
```

Scheduling a task for execution:

```
# Fetch in ~30s.
result = fetch_url.schedule(('https://google.com',), delay=30)

# Wait until result is ready, takes ~30s.
html = result.get()
```

start()

Start the scheduler in a new green thread. The scheduler is needed if you plan to schedule tasks for execution using the `schedule()` method, or if you want to run periodic tasks.

Typically this method should be called when your application starts up. For example, a WSGI application might do something like:

```
# Always apply gevent monkey-patch before anything else!
from gevent import monkey; monkey.patch_all()

from my_app import app  # flask/bottle/whatever WSGI app.
from my_app import mini_huey

# Start the scheduler. Returns immediately.
mini_huey.start()

# Run the WSGI server.
from gevent.pywsgi import WSGIServer
WSGIServer(('127.0.0.1', 8000), app).serve_forever()
```

stop()
Stop the scheduler.

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

Note: Tasks enqueued for immediate execution will be run regardless of whether the scheduler is running. You only need to start the scheduler if you plan to schedule tasks in the future or run periodic tasks.

2.8.2 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 = {
    'huey_class': 'huey.RedisHuey',  # Huey implementation to use.
    'name': settings.DATABASES['default']['NAME'],  # Use db name for huey.
    'results': True,  # Store return values of tasks.
    'store_none': False,  # If a task returns None, do not save to results.
    'immediate': settings.DEBUG,  # If DEBUG=True, run synchronously.
```

(continues on next page)

(continued from previous page)

```
    'utc': True,  # Use UTC for all times internally.
    'blocking': True,  # Perform blocking pop rather than poll Redis.
    '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.
        '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.
        '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.
    },
}
```

The following `huey_class` implementations are provided out-of-the-box:

- `huey.RedisHuey` - default.
- `huey.PriorityRedisHuey` - uses Redis but adds support for *Task priority*. Requires redis server 5.0 or newer.
- `huey.SqliteHuey` - uses Sqlite, full support for task priorities.

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*.

-A, --disable-autoload Disable automatic loading of tasks modules.

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:

```
#!/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
```

(continues on next page)

(continued from previous page)

```
@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')
```

The `huey.contrib.djhuey` module exposes a number of additional helpers:

- *lock_task()*
- *enqueue()*
- *restore()*, *restore_all()*, *restore_by_id()*
- *revoke()*, *revoke_all()*, *revoke_by_id()*
- *is_revoked()*
- *on_startup()*
- *pre_execute()*
- *post_execute()*
- *signal()* and *disconnect_signal()*

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 you prefer to use a live storage engine when `DEBUG` is enabled, you can specify `immediate_use_memory=False` - which still runs Huey in immediate mode, but using a live storage API. To completely disable immediate mode when `DEBUG` is set, specify `immediate=False` in your settings.

```
# settings.py
HUEY = {
    'name': 'my-app',

    # To run Huey in "immediate" mode with a live storage API, specify
    # immediate_use_memory=False.
    'immediate_use_memory': False,

    # OR:
    # To run Huey in "live" mode regardless of whether DEBUG is enabled,
    # specify immediate=False.
    'immediate': 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')
```

2.9 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 *`HueyException`* "XXX not found in TaskRegistry" errors.

"HueyException: 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 *import documentation*.

"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 `results=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. Huey uses UTC internally by default, and naive datetimes will be converted from local time to UTC (if local time happens to not be UTC).

Cronjobs are not being run The consumer and scheduler run in UTC by default.

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 Use `immediate=True`:

```
test_mode = os.environ.get('TEST_MODE')

# When immediate=True, Huey will default to using an in-memory
# storage layer.
huey = RedisHuey(immediate=test_mode)

# Alternatively, you can set the `immediate` attribute:
huey.immediate = True if test_mode else False
```

2.10 Changes in 2.0

The 2.0 release of Huey is mostly API-compatible with previous versions, but there are a number of things that have been altered or improved in this release.

Warning: The serialization format for tasks has changed. An attempt has been made to provide backward compatibility when reading messages enqueued by an older version of Huey, but this is not guaranteed to work.

2.10.1 Summary

The `always_eager` mode has been renamed *Immediate mode*. As the new name implies, tasks are run immediately instead of being enqueued. Immediate mode is designed to be used during testing and development. When immediate mode is enabled, Huey switches to using in-memory storage by default, so as to avoid accidental writes to a live storage. Immediate mode improves greatly on `always_eager` mode, as it no longer requires special-casing and follows the same code-paths used when Huey is in live mode. See *Immediate mode* for more details.

Previously, the Huey consumer accepted options to run in UTC or local-time. Various APIs, particularly around scheduling and task revocation, needed to be compatible with however the consumer was configured, and it could easily get confusing. As of 2.0, UTC-vs-localtime is specified when instantiating Huey, and all conversion happens internally, hopefully making things easier to think about – that is, you don't have to think about it.

The events APIs have been removed and replaced by a *Signals* system. Signal handlers are executed synchronously by the worker(s) as they run, so it's a bit different, but hopefully a lot easier to actually utilize, as the events API required a dedicated listener thread if you were to make any use of it (since it used a pub/sub approach). Events could be built on-top of the signals, but currently I have no plans for this.

Errors are no longer stored in a separate list. Should a task fail due to an unhandled exception, the exception will be placed in the result store, and can be introspected using the task's *`Result`* handle.

Huey now supports *Task priority*. To use priorities with Redis, you need to be running Redis 5.0 or newer, and should use *`PriorityRedisHuey`*. The original *`RedisHuey`* continues to support older versions of Redis. *`SqliteHuey`* and the in-memory storage used for dev/testing provide full support for task priorities.

2.10.2 Details

Changes when initializing *`Huey`*:

- `result_store` parameter has been renamed to `results`.
- `events` parameter is removed. Events have been replaced by *Signals*.
- `store_errors` parameter is removed. Huey no longer maintains a separate list of recent errors. Unhandled errors that occur when running a task are stored in the result store. Also the `max_errors` parameter of the Redis storage engine is removed.
- `global_registry` parameter is removed. Tasks are no longer registered to a global registry - tasks are registered to the Huey instance with which they are decorated.
- `always_eager` has been renamed `immediate`.

New initialization arguments:

- Boolean `utc` parameter (defaults to true). This setting is used to control how Huey interprets datetimes internally. Previously, this logic was spread across a number of APIs and a consumer flag.
- Serializer parameter accepts an (optional) object implementing the *`Serializer`* interface. Defaults to using `pickle`.
- Accepts option to use gzip `compression` when serializing data.

Other changes to *`Huey`*:

- Immediate mode can be enabled or disabled at runtime by setting the *`immediate`* property.

- Event emitter has been replaced by *Signals*, so all event-related APIs have been removed.
- Special classes of exceptions for the various storage operations have been removed. For more information see *Exceptions*.
- The `Huey.errors()` method is gone. Errors are no longer tracked separately.

Changes to the *task()* and *periodic_task()* decorators:

- Previously these decorators accepted two optional keyword arguments, `retries_as_argument` and `include_task`. Since the remaining retries are stored as an attribute on the task itself, the first is redundant. In 2.0 these are replaced by a new keyword argument `context`, which, if `True`, will pass the task instance to the decorated function as a keyword argument.
- Enqueueing a task pipeline will now return a *ResultGroup* instead of a list of individual *Result* instances.

Changes to the *Result* handle (previous called `TaskResultWrapper`):

- The `task_id` property is renamed to `id`.
- Task instances that are revoked via *Result.revoke()* will default to using `revoke_once=True`.
- The *reschedule()* method no longer requires a delay or eta. Leaving both empty will reschedule the task immediately.

Changes to *crontab()*:

- The order of arguments has been changed to match the order used on linux crontab. The order is now minute, hour, day, month, day of week.

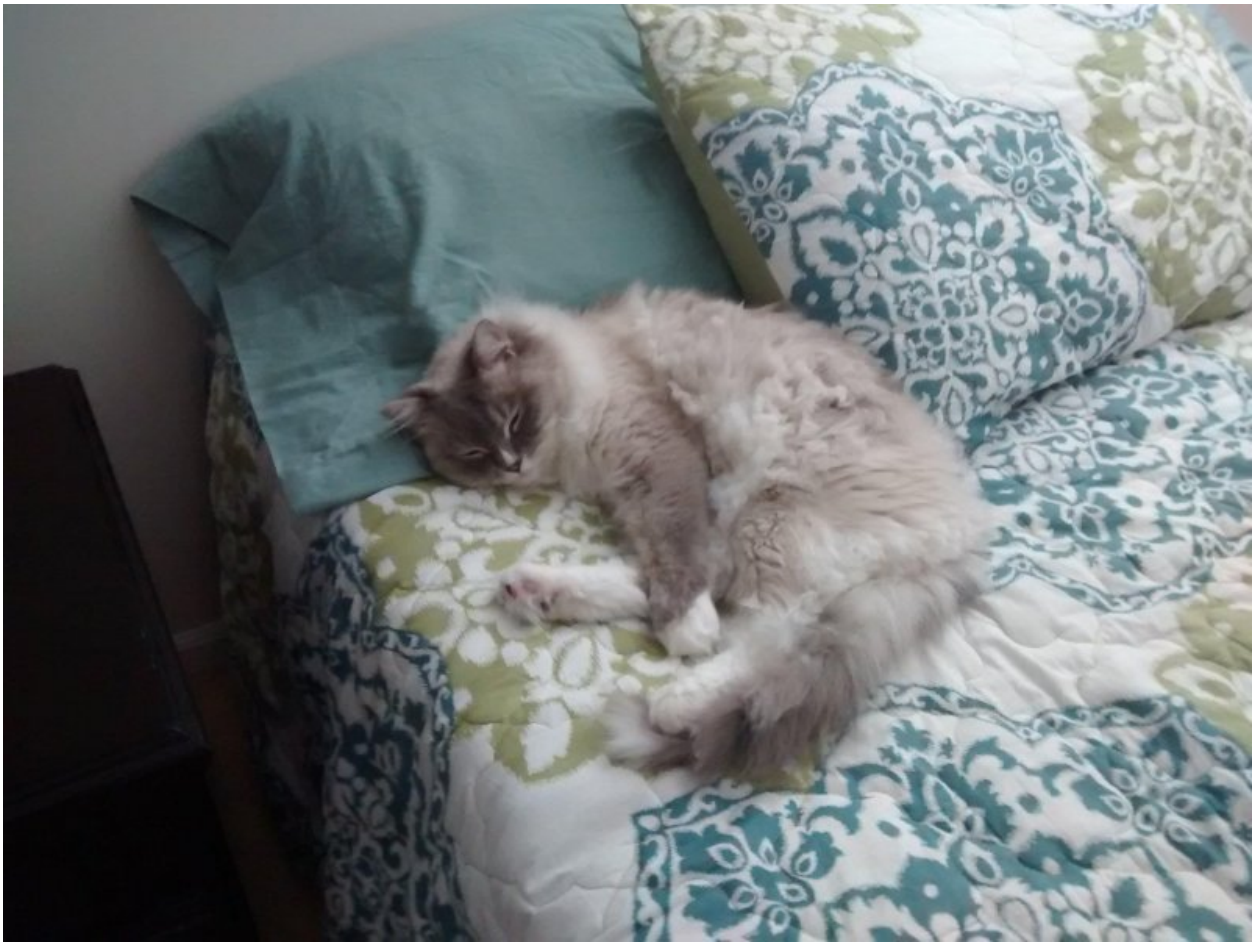
Miscellaneous:

- Huey no longer uses a global registry for task functions. Task functions are only visible to the huey instance they are decorated by.
- `RedisHuey` defaults to using a blocking pop on the queue, which should improve latency and reduce chatter. To go back to the old polling default, specify `blocking=False` when creating your huey instance.
- `SqliteHuey` no longer has any third-party dependencies and has been moved into the main `huey` module.
- The `MiniHuey` contrib module has been renamed to `huey.contrib.mini`.
- The `SimpleStorage` contrib module has been removed.

Django-specific:

- The `backend_class` setting has been renamed to `huey_class` (used to specify import-path to Huey implementation, e.g. `huey.RedisHuey`).

Huey is named in honor of my cat



CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index