Logutils Documentation

Release 0.3.6

Vinay Sajip

Nov 16, 2019

Contents

The logutils package provides a set of handlers for the Python standard library's logging package.

Some of these handlers are out-of-scope for the standard library, and so they are packaged here. Others are updated versions which have appeared in recent Python releases, but are usable with older versions of Python, and so are packaged here.

For recent changes, see whats-new.

There are a number of subcomponents to this package, relating to particular tasks you may want to perform:

CHAPTER 1

Configuring Libraries

When developing libraries, you'll probably need to use the *NullHandler* class.

N.B. This is part of the standard library since Python 2.7 / 3.1, so the version here is for use with earlier Python versions.

Typical usage:

```
import logging
try:
    from logging import NullHandler
except ImportError:
    from logutils import NullHandler

 # use this in all your library's subpackages/submodules
logger = logging.getLogger(__name__)

# use this just in your library's top-level package
logger.addHandler(NullHandler())
```

class logutils.**NullHandler**(*level=0*)

This handler does nothing. It's intended to be used to avoid the "No handlers could be found for logger XXX" one-off warning. This is important for library code, which may contain code to log events. If a user of the library does not configure logging, the one-off warning might be produced; to avoid this, the library developer simply needs to instantiate a NullHandler and add it to the top-level logger of the library module or package.

createLock()

Since this handler does nothing, it has no underlying I/O to protect against multi-threaded access, so this method returns *None*.

emit(*record*)

Emit a record. This does nothing and shouldn't be called during normal processing, unless you redefine *handle()*.

handle(*record*)

Handle a record. Does nothing in this class, but in other handlers it typically filters and then emits the record in a thread-safe way.

CHAPTER 2

Working with queues

This module contains classes which help you work with queues. A typical application is when you want to log from performance-critical threads, but where the handlers you want to use are slow (for example, `SMTPHandler`). In that case, you can create a queue, pass it to a *`QueueHandler`* instance and use that instance with your loggers. Elsewhere, you can instantiate a *`QueueListener`* with the same queue and some slow handlers, and call *`start()`* on it. This will start monitoring the queue on a separate thread and call all the configured handlers *on that thread*, so that your logging thread is not held up by the slow handlers.

Note that as well as in-process queues, you can use these classes with queues from the `multiprocessing` module.

N.B. This is part of the standard library since Python 3.2, so the version here is for use with earlier Python versions.

class `logutils.queue.`**QueueHandler**(*queue*)

This handler sends events to a queue. Typically, it would be used together with a multiprocessing Queue to centralise logging to file in one process (in a multi-process application), so as to avoid file write contention between processes.

Parameters **queue** – The queue to send *LogRecords* to.

emit(*record*)

Emit a record.

Writes the LogRecord to the queue, preparing it for pickling first.

Parameters **record** – The record to emit.

enqueue(*record*)

Enqueue a record.

The base implementation uses `put_nowait()`. You may want to override this method if you want to use blocking, timeouts or custom queue implementations.

Parameters **record** – The record to enqueue.

prepare(*record*)

Prepares a record for queuing. The object returned by this method is enqueued.

The base implementation formats the record to merge the message and arguments, and removes unpickleable items from the record in-place.

You might want to override this method if you want to convert the record to a dict or JSON string, or send a modified copy of the record while leaving the original intact.

Parameters **record** – The record to prepare.

class logutils.queue.**QueueListener**(*queue*, **handlers*, ***kwargs*)

This class implements an internal threaded listener which watches for LogRecords being added to a queue, removes them and passes them to a list of handlers for processing.

Parameters

- **record** – The queue to listen to.
- **handlers** – The handlers to invoke on everything received from the queue.

dequeue(*block*)

Dequeue a record and return it, optionally blocking.

The base implementation uses `get()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

Parameters **block** – Whether to block if the queue is empty. If *False* and the queue is empty, an `Empty` exception will be thrown.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses `put_nowait()`. You may want to override this method if you want to use timeouts or work with custom queue implementations.

handle(*record*)

Handle a record.

This just loops through the handlers offering them the record to handle.

Parameters **record** – The record to handle.

prepare(*record*)

Prepare a record for handling.

This method just returns the passed-in record. You may want to override this method if you need to do any custom marshalling or manipulation of the record before passing it to the handlers.

Parameters **record** – The record to prepare.

start()

Start the listener.

This starts up a background thread to monitor the queue for LogRecords to process.

stop()

Stop the listener.

This asks the thread to terminate, and then waits for it to do so. Note that if you don't call this before your application exits, there may be some records still left on the queue, which won't be processed.

CHAPTER 3

Working with Redis queues

QueueHandler and *QueueListener* classes are provided to facilitate interfacing with Redis.

class logutils.redis.**RedisQueueHandler**(*key='python.logging'*, *redis=None*, *limit=0*)

A QueueHandler implementation which pushes pickled records to a Redis queue using a specified key.

Parameters

- **key** – The key to use for the queue. Defaults to "python.logging".
- **redis** – If specified, this instance is used to communicate with a Redis instance.
- **limit** – If specified, the queue is restricted to have only this many elements.

enqueue(*record*)

Enqueue a record.

The base implementation uses put_nowait(). You may want to override this method if you want to use blocking, timeouts or custom queue implementations.

Parameters **record** – The record to enqueue.

class logutils.redis.**RedisQueueListener**(**handlers*, ***kwargs*)

A QueueListener implementation which fetches pickled records from a Redis queue using a specified key.

Parameters

- **key** – The key to use for the queue. Defaults to "python.logging".
- **redis** – If specified, this instance is used to communicate with a Redis instance.

dequeue(*block*)

Dequeue and return a record.

enqueue_sentinel()

Writes a sentinel to the queue to tell the listener to quit. This implementation uses put_nowait(). You may want to override this method if you want to use timeouts or work with custom queue implementations.

CHAPTER 4

Unit testing

When developing unit tests, you may find the *TestHandler* and *Matcher* classes useful.

Typical usage:

```
import logging
from logutils.testing import TestHandler, Matcher
import unittest

class LoggingTest(unittest.TestCase):
    def setUp(self):
        self.handler = h = TestHandler(Matcher())
        self.logger = l = logging.getLogger()
        l.addHandler(h)

    def tearDown(self):
        self.logger.removeHandler(self.handler)
        self.handler.close()

    def test_simple(self):
        "Simple test of logging test harness."
        # Just as a demo, let's log some messages.
        # Only one should show up in the log.
        self.logger.debug("This won't show up.")
        self.logger.info("Neither will this.")
        self.logger.warning("But this will.")
        h = self.handler
        self.assertTrue(h.matches(levelno=logging.WARNING))
        self.assertFalse(h.matches(levelno=logging.DEBUG))
        self.assertFalse(h.matches(levelno=logging.INFO))

    def test_partial(self):
        "Test of partial matching in logging test harness."
        # Just as a demo, let's log some messages.
        # Only one should show up in the log.
        self.logger.debug("This won't show up.")
```

(continues on next page)

(continued from previous page)

```
        self.logger.info("Neither will this.")
        self.logger.warning("But this will.")
        h = self.handler
        self.assertTrue(h.matches(msg="ut th")) # from "But this will"
        self.assertTrue(h.matches(message="ut th")) # from "But this will"
        self.assertFalse(h.matches(message="either"))
        self.assertFalse(h.matches(message="won't"))

    def test_multiple(self):
        "Test of matching multiple values in logging test harness."
        # Just as a demo, let's log some messages.
        # Only one should show up in the log.
        self.logger.debug("This won't show up.")
        self.logger.info("Neither will this.")
        self.logger.warning("But this will.")
        self.logger.error("And so will this.")
        h = self.handler
        self.assertTrue(h.matches(levelno=logging.WARNING,
                                  message='ut thi'))
        self.assertTrue(h.matches(levelno=logging.ERROR,
                                  message='nd so wi'))
        self.assertFalse(h.matches(levelno=logging.INFO))
```

class logutils.testing.**Matcher**

This utility class matches a stored dictionary of `logging.LogRecord` attributes with keyword arguments passed to its *matches()* method.

match_value(*k*, *dv*, *v*)

Try to match a single stored value (dv) with a supplied value (v).

Return *True* if found, else *False*.

Parameters

- **k** – The key value (LogRecord attribute name).
- **dv** – The stored value to match against.
- **v** – The value to compare with the stored value.

matches(*d*, ***kwargs*)

Try to match a single dict with the supplied arguments.

Keys whose values are strings and which are in self._partial_matches will be checked for partial (i.e. substring) matches. You can extend this scheme to (for example) do regular expression matching, etc.

Return *True* if found, else *False*.

Parameters **kwargs** – A set of keyword arguments whose names are LogRecord attributes and whose values are what you want to match in a stored LogRecord.

class logutils.testing.**TestHandler**(*matcher*)

This handler collects records in a buffer for later inspection by your unit test code.

Parameters **matcher** – The *Matcher* instance to use for matching.

count

The number of records in the buffer.

emit(*record*)

Saves the *__dict__* of the record in the *buffer* attribute, and the formatted records in the *formatted* attribute.

Parameters **record** – The record to emit.

flush()
Clears out the *buffer* and *formatted* attributes.

matchall(*kwarglist*)
Accept a list of keyword argument values and ensure that the handler's buffer of stored records matches the list one-for-one.

Return *True* if exactly matched, else *False*.

Parameters **kwarglist** – A list of keyword-argument dictionaries, each of which will be passed to *matches()* with the corresponding record from the buffer.

matches(**kwargs*)
Look for a saved dict whose keys/values match the supplied arguments.

Return *True* if found, else *False*.

Parameters **kwargs** – A set of keyword arguments whose names are LogRecord attributes and whose values are what you want to match in a stored LogRecord.

shouldFlush()
Should the buffer be flushed?

This returns *False* - you'll need to flush manually, usually after your unit test code checks the buffer contents against your expectations.

CHAPTER 5

Dictionary-based Configuration

This module implements dictionary-based configuration according to PEP 391.

N.B. This is part of the standard library since Python 2.7 / 3.2, so the version here is for use with earlier Python versions.

class logutils.dictconfig.**DictConfigurator**(*config*)
Configure logging using a dictionary-like object to describe the configuration.

configure()
Do the configuration.

logutils.dictconfig.**dictConfig**(*config*)
Configure logging using a dictionary.

CHAPTER 6

Working with Logger adapters

N.B. This is part of the standard library since Python 2.6 / 3.1, so the version here is for use with earlier Python versions.

The class was enhanced for Python 3.2, so you may wish to use this version with earlier Python versions.

However, note that the `LoggerAdapter` class will **not** work with Python 2.4 or earlier, as it uses the *extra* keyword argument which was added in later Python versions.

class logutils.adapter.**LoggerAdapter**(*logger*, *extra*)

An adapter for loggers which makes it easier to specify contextual information in logging output.

critical(*msg*, **args*, ***kwargs*)

Delegate a critical call to the underlying logger.

debug(*msg*, **args*, ***kwargs*)

Delegate a debug call to the underlying logger.

error(*msg*, **args*, ***kwargs*)

Delegate an error call to the underlying logger.

exception(*msg*, **args*, ***kwargs*)

Delegate an exception call to the underlying logger.

getEffectiveLevel()

Get the effective level for the underlying logger.

hasHandlers()

See if the underlying logger has any handlers.

info(*msg*, **args*, ***kwargs*)

Delegate an info call to the underlying logger.

isEnabledFor(*level*)

Is this logger enabled for level 'level'?

log(*level*, *msg*, **args*, ***kwargs*)

Delegate a log call to the underlying logger, after adding contextual information from this adapter instance.

process(*msg*, *kwargs*)
Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

setLevel(*level*)
Set the specified level on the underlying logger.

warn(*msg*, **args*, ***kwargs*)
Delegate a warning call to the underlying logger.

warning(*msg*, **args*, ***kwargs*)
Delegate a warning call to the underlying logger.

CHAPTER 7

Working with web sites

N.B. The *HTTPHandler* class has been present in the logging package since the first release, but was enhanced for Python 3.2 to add options for secure connections and user credentials. You may wish to use this version with earlier Python releases.

class logutils.http.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*)

A class which sends records to a Web server, using either GET or POST semantics.

Parameters

- **host** – The Web server to connect to.
- **url** – The URL to use for the connection.
- **method** – The HTTP method to use. GET and POST are supported.
- **secure** – set to True if HTTPS is to be used.
- **credentials** – Set to a username/password tuple if desired. If set, a Basic authentication header is sent. WARNING: if using credentials, make sure *secure* is *True* to avoid sending usernames and passwords in cleartext over the wire.

emit(*record*)

Emit a record.

Send the record to the Web server as a percent-encoded dictionary

Parameters **record** – The record to be emitted.

mapLogRecord(*record*)

Default implementation of mapping the log record into a dict that is sent as the CGI data. Overwrite in your class. Contributed by Franz Glasner.

Parameters **record** – The record to be mapped.

CHAPTER 8

Colorizing Console Streams

`ColorizingStreamHandler` is a handler which allows colorizing of console streams, described here in more detail.

class logutils.colorize.**ColorizingStreamHandler**(*stream=None*)

A stream handler which supports colorizing of console streams under Windows, Linux and Mac OS X.

Parameters **strm** – The stream to colorize - typically `sys.stdout` or `sys.stderr`.

colorize(*message*, *record*)

Colorize a message for a logging event.

This implementation uses the `level_map` class attribute to map the LogRecord's level to a colour/intensity setting, which is then applied to the whole message.

Parameters

- **message** – The message to colorize.
- **record** – The `LogRecord` for the message.

emit(*record*)

Emit a record.

If a formatter is specified, it is used to format the record. The record is then written to the stream with a trailing newline. If exception information is present, it is formatted using traceback.print_exception and appended to the stream. If the stream has an 'encoding' attribute, it is used to determine how to do the output to the stream.

format(*record*)

Formats a record for output.

This implementation colorizes the message line, but leaves any traceback unolorized.

is_tty

Returns true if the handler's stream is a terminal.

level_map = {10: (None, 'blue', False), 20: (None, 'black', False), 30: (None, 'yel

Maps levels to colour/intensity settings.

output_colorized(*message*)
Output a colorized message.

On Linux and Mac OS X, this method just writes the already-colorized message to the stream, since on these platforms console streams accept ANSI escape sequences for colorization. On Windows, this handler implements a subset of ANSI escape sequence handling by parsing the message, extracting the sequences and making Win32 API calls to colorize the output.

Parameters **message** – The message to colorize and output.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

N

O

P

Q

R

S

T

W