Logbook Documentation

Release 0.4

Armin Ronacher, Georg Brandl

September 14, 2015

Contents

Logbook is a logging sytem for Python that replaces the standard library's logging module. It was designed with both complex and simple applications and mind and the idea to make logging fun:

```
>>> from logbook import Logger
>>> log = Logger('Logbook')
>>> log.info('Hello, World!')
[2010-07-23 16:34] INFO: Logbook: Hello, World!
```

What makes it fun? What about getting log messages on your phone or desktop notification system? *Logbook can do that*.

This library is still under heavy development and the API is not fully finalized yet. Feedback is appreciated. The docs here only show a tiny, tiny feature set and are terribly incomplete. We will have better docs soon, but until then we hope this gives a sneak peak about how cool Logbook is. If you want more, have a look at the comprehensive testsuite.

CHAPTER 1

Documentation

1.1 What does it do?

Although the Python standard library provides a logging system, you should consider having a look at Logbook for your applications. Currently logbook is an alpha version and should be considered a developer preview.

But give it a try, we think it will work out for you and be fun to use :)

Furthermore because it was prototyped in a couple of days, it leverages some features of Python that are not available in older Python releases. Logbook currently requires Python 2.4 or higher including Python 3 (3.1 or higher, 3.0 is not supported).

1.1.1 Core Features

- Logbook is based on the concept of loggers that are extensible by the application.
- Each logger and handler, as well as other parts of the system, may inject additional information into the logging record that improves the usefulness of log entries.
- Handlers can be set on an application-wide stack as well as a thread-wide stack. Setting a handler does not replace existing handlers, but gives it higher priority. Each handler has the ability to prevent records from propagating to lower-priority handlers.
- Logbook comes with a useful default configuration that spits all the information to stderr in a useful manner.
- All of the built-in handlers have a useful default configuration applied with formatters that provide all the available information in a format that makes the most sense for the given handler. For example, a default stream handler will try to put all the required information into one line, whereas an email handler will split it up into nicely formatted ASCII tables that span multiple lines.
- Logbook has built-in handlers for streams, arbitrary files, files with time and size based rotation, a handler that delivers mails, a handler for the syslog daemon as well as the NT log file.
- There is also a special “fingers crossed” handler that, in combination with the handler stack, has the ability to accumulate all logging messages and will deliver those in case a severity level was exceeded. For example, it can withhold all logging messages for a specific request to a web application until an error record appears, in which case it will also send all withheld records to the handler it wraps. This way, you can always log lots of debugging records, but only get see them when they can actually tell you something of interest.
- It is possible to inject a handler for testing that records messages for assertions.
- Logbook was designed to be fast and with modern Python features in mind. For example, it uses context managers to handle the stack of handlers as well as new-style string formatting for all of the core log calls.

- Builtin support for ZeroMQ and other means to distribute log messages between heavily distributed systems and multiple processes.
- The Logbook system does not depend on log levels. In fact, custom log levels are not supported, instead we strongly recommend using logging subclasses or log processors that inject tagged information into the log record for this purpose.
- **PEP 8** naming and code style.

1.1.2 Advantages over Logging

If properly configured, Logbook's logging calls will be very cheap and provide a great performance improvement over an equivalent configuration of the standard library's logging module. While for some parts we are not quite at performance we desire, there will be some further performance improvements in the upcoming versions.

It also supports the ability to inject additional information for all logging calls happening in a specific thread or for the whole application. For example, this makes it possible for a web application to add request-specific information to each log record such as remote address, request URL, HTTP method and more.

The logging system is (besides the stack) stateless and makes unit testing it very simple. If context managers are used, it is impossible to corrupt the stack, so each test can easily hook in custom log handlers.

1.1.3 Cooperation

Logbook is an addon library to Python and working in an area where there are already a couple of contestants. First of all there is the standard library's `logging` module, secondly there is also the `warnings` module which is used internally in Python to warn about invalid uses of APIs and more. We know that there are many situations where you want to use either of them. Be it that they are integrated into a legacy system, part of a library outside of your control or just because they are a better choice.

Because of that, Logbook is two-way compatible with `logging` and one-way compatible with `warnings`. If you want, you can let all logging calls redirect to the logbook handlers or the other way round, depending on what your desired setup looks like. That way you can enjoy the best of both worlds.

1.1.4 It should be Fun

Logging should be fun. A good log setup makes debugging easier when things go rough. For good results you really have to start using logging before things actually break. Logbook comes with a couple of unusual log handlers to bring the fun back to logging. You can log to your personal twitter feed, you can log to mobile devices, your desktop notification system and more.

1.1.5 Logbook in a Nutshell

This is how easy it is to get started with Logbook:

```
from logbook import warn
warn('This is a warning')
```

That will use the default logging channel. But you can create as many as you like:

```
from logbook import Logger
log = Logger('My Logger')
log.warn('This is a warning')
```

1.1.6 Roadmap

Here a list of things you can expect in upcoming versions:

- c implementation of the internal stack management and record dispatching for higher performance.
- a ticketing log handler that creates tickets in trac and redmine.
- a web frontend for the ticketing database handler.

1.2 Quickstart

Logbook makes it very easy to get started with logging. Just import the logger class, create yourself a logger and you are set:

```
>>> from logbook import Logger
>>> log = Logger('My Awesome Logger')
>>> log.warn('This is too cool for stdlib')
[2010-07-23 16:34] WARNING: My Awesome Logger: This is too cool for stdlib
```

A logger is a so-called *RecordDispatcher*, which is commonly referred to as a "logging channel". The name you give such a channel is up to you and need not be unique although it's a good idea to keep it unique so that you can filter by it if you want.

The basic interface is similar to what you may already know from the standard library's `logging` module.

There are several logging levels, available as methods on the logger. The levels – and their suggested meaning – are:

- `critical` – for errors that lead to termination
- `error` – for errors that occur, but are handled
- `warning` – for exceptional circumstances that might not be errors
- `notice` – for non-error messages you usually want to see
- `info` – for messages you usually don't want to see
- `debug` – for debug messages

Each of these levels is available as method on the `Logger`. Additionally the `warning` level is aliased as `warn()`.

Alternatively, there is the `log()` method that takes the logging level (string or integer) as an argument.

1.2.1 Handlers

Each call to a logging method creates a log *record* which is then passed to *handlers*, which decide how to store or present the logging info. There are a multitude of available handlers, and of course you can also create your own:

- `StreamHandler` for logging to arbitrary streams
- `StderrHandler` for logging to stderr
- `FileHandler`, `MonitoringFileHandler`, `RotatingFileHandler` and `TimedRotatingFileHandler` for logging to files
- `MailHandler` for logging via e-mail
- `SyslogHandler` for logging to the syslog daemon
- `NTEventLogHandler` for logging to the Windows NT event log

On top of those there are a couple of handlers for special use cases:

- `logbook.FingersCrossedHandler` for logging into memory and delegating information to another handler when a certain level was exceeded, otherwise discarding all buffered records.
- *`logbook.more.TaggingHandler`* for dispatching log records that are tagged (used in combination with a *`logbook.more.TaggingLogger`*)
- *`logbook.queues.ZeroMQHandler`* for logging to ZeroMQ
- *`logbook.queues.MultiProcessingHandler`* for logging from a child process to a handler from the outer process.
- *`logbook.queues.ThreadedWrapperHandler`* for moving the actual handling of a handler into a background thread and using a queue to deliver records to that thread.
- *`logbook.notifiers.GrowlHandler`* and *`logbook.notifiers.LibNotifyHandler`* for logging to the OS X Growl or the linux notification daemon.
- *`logbook.notifiers.BoxcarHandler`* for logging to boxcar.
- *`logbook.more.TwitterHandler`* for logging to twitter.
- *`logbook.more.ExternalApplicationHandler`* for logging to an external application such as the OS X `say` command.
- *`logbook.ticketing.TicketingHandler`* for creating tickets from log records in a database or other data store.

1.2.2 Registering Handlers

So how are handlers registered? If you are used to the standard Python logging system, it works a little bit differently here. Handlers can be registered for a thread or for a whole process or individually for a logger. However, it is strongly recommended not to add handlers to loggers unless there is a very good use case for that.

If you want errors to go to syslog, you can set up logging like this:

```
from logbook import SyslogHandler

error_handler = SyslogHandler('logbook example', level='ERROR')
with error_handler.applicationbound():
    # whatever is executed here and an error is logged to the
    # error handler
    ...
```

This will send all errors to the syslog but warnings and lower record levels still to stderr. This is because the handler is not bubbling by default which means that if a record is handled by the handler, it will not bubble up to a higher handler. If you want to display all records on stderr, even if they went to the syslog you can enable bubbling by setting *bubble* to `True`:

```
from logbook import SyslogHandler

error_handler = SyslogHandler('logbook example', level='ERROR', bubble=True)
with error_handler.applicationbound():
    # whatever is executed here and an error is logged to the
    # error handler but it will also bubble up to the default
    # stderr handler.
    ...
```

So what if you want to only log errors to the syslog and nothing to stderr? Then you can combine this with a `NullHandler`:

```
from logbook import SyslogHandler, NullHandler

error_handler = SyslogHandler('logbook example', level='ERROR')
null_handler = NullHandler()

with null_handler.applicationbound():
    with error_handler.applicationbound():
        # errors now go to the error_handler and everything else
        # is swallowed by the null handler so nothing ends up
        # on the default stderr handler
        ...
```

1.2.3 Record Processors

What makes logbook interesting is the ability to automatically process log records. This is handy if you want additional information to be logged for everything you do. A good example use case is recording the IP of the current request in a web application. Or, in a daemon process you might want to log the user and working directory of the process.

A context processor can be injected at two places: you can either bind a processor to a stack like you do with handlers or you can override the override the *RecordDispatcher.process_record()* method.

Here an example that injects the current working directory into the *extra* dictionary of a log record:

```
import os
from logbook import Processor

def inject_cwd(record):
    record.extra['cwd'] = os.getcwd()

with my_handler.applicationbound():
    with Processor(inject_cwd).applicationbound():
        # everything logged here will have the current working
        # directory in the log record.
        ...
```

The alternative is to inject information just for one logger in which case you might want to subclass it:

```
import os

class MyLogger(logbook.Logger):

    def process_record(self, record):
        logbook.Logger.process_record(self, record)
        record.extra['cwd'] = os.getcwd()
```

1.2.4 Configuring the Logging Format

All handlers have a useful default log format you don't have to change to use logbook. However if you start injecting custom information into log records, it makes sense to configure the log formatting so that you can see that information.

There are two ways to configure formatting: you can either just change the format string or hook in a custom format function.

All the handlers that come with logbook and that log into a string use the `StringFormatter` by default. Their constructors accept a format string which sets the `logbook.Handler.format_string` attribute. You can override this attribute in which case a new string formatter is set:

```
>>> from logbook import StderrHandler
>>> handler = StderrHandler()
>>> handler.format_string = '{record.channel}: {record.message}'
>>> handler.formatter
<logbook.handlers.StringFormatter object at 0x100641b90>
```

Alternatively you can also set a custom format function which is invoked with the record and handler as arguments:

```
>>> def my_formatter(record, handler):
...  return record.message
...
>>> handler.formatter = my_formatter
```

The format string used for the default string formatter has one variable called *record* available which is the log record itself. All attributes can be looked up using the dotted syntax, and items in the *extra* dict looked up using brackets. Note that if you are accessing an item in the extra dict that does not exist, an empty string is returned.

Here is an example configuration that shows the current working directory from the example in the previous section:

```
handler = StderrHandler(format_string=
    '{record.channel}: {record.message) [{record.extra[cwd]}]')
```

In the *more* module there is a formatter that uses the Jinja2 template engine to format log records, especially useful for multi-line log formatting such as mails (*JinjaFormatter*).

1.3 Common Logbook Setups

This part of the documentation shows how you can configure Logbook for different kinds of setups.

1.3.1 Desktop Application Setup

If you develop a desktop application (command line or GUI), you probably have a line like this in your code:

```
if __name__ == '__main__':
    main()
```

This is what you should wrap with a `with` statement that sets up your log handler:

```
from logbook import FileHandler
log_handler = FileHandler('application.log')

if __name__ == '__main__':
    with log_handler.applicationbound():
        main()
```

Alternatively you can also just push a handler in there:

```
from logbook import FileHandler
log_handler = FileHandler('application.log')
log_handler.push_application()

if __name__ == '__main__':
    main()
```

Please keep in mind that you will have to pop the handlers in reverse order if you want to remove them from the stack, so it is recommended to use the context manager API if you plan on reverting the handlers.

1.3.2 Web Application Setup

Typical modern web applications written in Python have two separate contexts where code might be executed: when the code is imported, as well as when a request is handled. The first case is easy to handle, just push a global file handler that writes everything into a file.

But Logbook also gives you the ability to improve upon the logging. For example, you can easily create yourself a log handler that is used for request-bound logging that also injects additional information.

For this you can either subclass the logger or you can bind to the handler with a function that is invoked before logging. The latter has the advantage that it will also be triggered for other logger instances which might be used by a different library.

Here is a simple WSGI example application that showcases sending error mails for errors happened during a WSGI application:

```
from logbook import MailHandler

mail_handler = MailHandler('errors@example.com',
                           ['admin@example.com'],
                           format_string=u'''\
Subject: Application Error at {record.extra[url]}

Message type:       {record.level_name}
Location:           {record.filename}:{record.lineno}
Module:             {record.module}
Function:           {record.func_name}
Time:               {record.time:%Y-%m-%d %H:%M:%S}
Remote IP:          {record.extra[ip]}
Request:            {record.extra[url]} [{record.extra[method]}]

Message:

{record.message}
''', bubble=True)

def application(environ, start_response):
    request = Request(environ)

    def inject_info(record, handler):
        record.extra.update(
            ip=request.remote_addr,
            method=request.method,
            url=request.url
        )

    with mail_handler.threadbound(processor=inject_info):
        # standard WSGI processing happens here.  If an error
        # is logged, a mail will be sent to the admin on
        # example.com
        ...
```

1.3.3 Deeply Nested Setups

If you want deeply nested logger setups, you can use the `NestedSetup` class which simplifies that. This is best explained using an example:

```
import os
from logbook import NestedSetup, NullHandler, FileHandler, \
     MailHandler, Processor

def inject_information(record):
    record.extra['cwd'] = os.getcwd()

# a nested handler setup can be used to configure more complex setups
setup = NestedSetup([
    # make sure we never bubble up to the stderr handler
    # if we run out of setup handling
    NullHandler(),
    # then write messages that are at least warnings to to a logfile
    FileHandler('application.log', level='WARNING'),
    # errors should then be delivered by mail and also be kept
    # in the application log, so we let them bubble up.
    MailHandler('servererrors@example.com',
                    ['admin@example.com'],
                    level='ERROR', bubble=True),
    # while we're at it we can push a processor on its own stack to
    # record additional information.  Because processors and handlers
    # go to different stacks it does not matter if the processor is
    # added here at the bottom or at the very beginning.  Same would
    # be true for flags.
    Processor(inject_information)
])
```

Once such a complex setup is defined, the nested handler setup can be used as if it was a single handler:

```
with setup.threadbound():
    # everything here is handled as specified by the rules above.
    ...
```

1.3.4 Distributed Logging

For applications that are spread over multiple processes or even machines logging into a central system can be a pain. Logbook supports ZeroMQ to deal with that. You can set up a *ZeroMQHandler* that acts as ZeroMQ publisher and will send log records encoded as JSON over the wire:

```
from logbook.queues import ZeroMQHandler
handler = ZeroMQHandler('tcp://127.0.0.1:5000')
```

Then you just need a separate process that can receive the log records and hand it over to another log handler using the *ZeroMQSubscriber*. The usual setup is this:

```
from logbook.queues import ZeroMQSubscriber
subscriber = ZeroMQSubscriber('tcp://127.0.0.1:5000')
with my_handler:
    subscriber.dispatch_forever()
```

You can also run that loop in a background thread with *dispatch_in_background()*:

```
from logbook.queues import ZeroMQSubscriber
subscriber = ZeroMQSubscriber('tcp://127.0.0.1:5000')
subscriber.dispatch_in_background(my_handler)
```

If you just want to use this in a `multiprocessing` environment you can use the *MultiProcessingHandler*

and *MultiProcessingSubscriber* instead. They work the same way as the ZeroMQ equivalents but are connected through a `multiprocessing.Queue`:

```
from multiprocessing import Queue
from logbook.queues import MultiProcessingHandler, \
     MultiProcessingSubscriber
queue = Queue(-1)
handler = MultiProcessingHandler(queue)
subscriber = MultiProcessingSubscriber(queue)
```

1.3.5 Redirecting Single Loggers

If you want to have a single logger go to another logfile you have two options. First of all you can attach a handler to a specific record dispatcher. So just import the logger and attach something:

```
from yourapplication.yourmodule import logger
logger.handlers.append(MyHandler(...))
```

Handlers attached directly to a record dispatcher will always take precedence over the stack based handlers. The bubble flag works as expected, so if you have a non-bubbling handler on your logger and it always handles, it will never be passed to other handlers.

Secondly you can write a handler that looks at the logging channel and only accepts loggers of a specific kind. You can also do that with a filter function:

```
handler = MyHandler(filter=lambda r: r.channel == 'app.database')
```

Keep in mind that the channel is intended to be a human readable string and is not necessarily unique. If you really need to keep loggers apart on a central point you might want to introduce some more meta information into the extra dictionary.

You can also compare the dispatcher on the log record:

```
from yourapplication.yourmodule import logger
handler = MyHandler(filter=lambda r: r.dispatcher is logger)
```

This however has the disadvantage that the dispatcher entry on the log record is a weak reference and might go away unexpectedly and will not be there if log records are sent to a different process.

Last but not least you can check if you can modify the stack around the execution of the code that triggers that logger For instance if the logger you are interested in is used by a specific subsystem, you can modify the stacks before calling into the system.

1.4 Stacks in Logbook

Logbook keeps three stacks internally currently:

- one for the `Handlers`: each handler is handled from stack top to bottom. When a record was handled it depends on the `bubble` flag of the handler if it should still be processed by the next handler on the stack.
- one for the `Processors`: each processor in the stack is applied on a record before the log record is handled by the handler.
- one for the `Flags`: this stack manages simple flags such as how errors during logging should be processed or if stackframe introspection should be used etc.

1.4.1 General Stack Management

Generally all objects that are management by stacks have a common interface (*StackedObject*) and can be used in combination with the `NestedSetup` class.

Commonly stacked objects are used with a context manager (*with* statement):

```
with context_object.threadbound():
    # this is managed for this thread only
    ...

with context_object.applicationbound():
    # this is managed for all applications
    ...
```

Alternatively you can also use *try/finally*:

```
context_object.push_thread()
try:
    # this is managed for this thread only
    ...
finally:
    context_object.pop_thread()

context_object.push_application()
try:
    # this is managed for all applications
    ...
finally:
    context_object.pop_application()
```

It's very important that you will always pop from the stack again unless you really want the change to last until the application closes down, which probably is not the case.

If you want to push and pop multiple stacked objects at the same time, you can use the `NestedSetup`:

```
setup = NestedSetup([stacked_object1, stacked_object2])
with setup.threadbound():
    # both objects are now bound to the thread's stack
    ...
```

Sometimes a stacked object can be passed to one of the functions or methods in Logbook. If any stacked object can be passed, this is usually called the *setup*. This is for example the case when you specify a handler or processor for things like the *ZeroMQSubscriber*.

1.4.2 Handlers

Handlers use the features of the stack the most because not only do they stack, but they also specify how stack handling is supposed to work. Each handler can decide if it wants to process the record, and then it has a flag (the `bubble` flag) which specifies if the next handler in the chain is supposed to get this record passed to.

If a handler is bubbeling it will give the record to the next handler, even if it was properly handled. If it's not, it will stop promoting handlers further down the chain. Additionally there are so-called "blackhole" handlers (`NullHandler`) which stop processing at any case when they are reached. If you push a blackhole handler on top of an existing infrastructure you can build up a separate one without performance impact.

1.4.3 Processor

A processor can inject additional information into a log record when the record is handled. Processors are called once at least one log handler is interested in handling the record. Before that happens, no processing takes place.

Here an example processor that injects the current working directory into the extra attribute of the record:

```
import os

def inject_cwd(record):
    record.extra['cwd'] = os.getcwd()

with Processor(inject_cwd):
    # all logging calls inside this block in this thread will now
    # have the current working directory information attached.
    ...
```

1.4.4 Flags

The last pillar of logbook is the flags stack. This stack can be used to override settings of the logging system. Currently this can be used to change the behavior of logbook in case an exception during log handling happens (for instance if a log record is supposed to be delivered to the filesystem but it ran out of available space). Additionally there is a flag that disables frame introspection which can result in a speedup on JIT compiled Python interpreters.

Here an example of a silenced error reporting:

```
with Flags(errors='silent'):
    # errors are now silent for this block
    ...
```

1.5 Performance Tuning

The more logging calls you add to your application and libraries, the more overhead will you introduce. There are a couple things you can do to remedy this behavior.

1.5.1 Debug-Only Logging

There are debug log calls, and there are debug log calls. Some debug log calls would sometimes be interesting in a production environment, others really only if you are on your local machine fiddling around with the code. Logbook internally makes sure to process as little of your logging call as necessary, but it will still have to walk the current stack to figure out if there are any active handlers or not. Depending on the number of handlers on the stack, the kind of handler etc, there will be more or less processed.

Generally speaking a not-handled logging call is cheap enough that you don't have to care about it. However there is not only your logging call, there might also be some data you have to process for the record. This will always be processed, even if the log record ends up being discarded.

This is where the Python `__debug__` feature comes in handy. This variable is a special flag that is evaluated at the time where Python processes your script. It can elliminate code completely from your script so that it does not even exist in the compiled bytecode (requires Python to be run with the `-O` switch):

```
if __debug__:
    info = get_wallcalculate_debug_info()
    logger.debug("Call to response() failed.  Reason: {0}", info)
```

1.5.2 Keep the Fingers Crossed

Do you really need the debug info? In case you find yourself only looking at the logfiles when errors occurred it would be an option to put in the `FingersCrossedHandler`. Logging into memory is always cheaper than logging on a filesystem.

1.5.3 Keep the Stack Static

Whenever you do a push or pop from one of the stacks you will invalidate an internal cache that is used by logbook. This is an implementation detail, but this is how it works for the moment. That means that the first logging call after a push or pop will have a higher impact on the performance than following calls. That means you should not attempt to push or pop from a stack for each logging call. Make sure to do the pushing and popping only as needed. (start/end of application/request)

1.5.4 Disable Introspection

By default Logbook will try to pull in the interpreter frame of the caller that invoked a logging function. While this is a fast operation that usually does not slow down the execution of your script it also means that for certain Python implementations it invalidates assumptions a JIT compiler might have made of the function body. Currently this for example is the case for applications running on pypy. If you would be using a stock logbook setup on pypy, the JIT wouldn't be able to work properly.

In case you don't need the frame based information (name of module, calling function, filename, line number) you can disable the introspection feature:

```
from logbook import Flags

with Flags(introspection=False):
    # all logging calls here will not use introspection
    ...
```

1.6 Logbook in Libraries

Logging becomes more useful the higher the number of components in a system that are using it. Logbook itself is not a widely supported library so far, but a handful of libraries are using the `logging` already which can be redirected to Logbook if necessary.

Logbook itself is easier to support for libraries than logging because it does away with the central logger registry and can easily be mocked in case the library is not available.

1.6.1 Mocking Logbook

If you want to support Logbook in your library but not depend on it you can copy/paste the following piece of code. It will attempt to import logbook and create a `Logger` and if it fails provide a class that just swallows all calls:

```
try:
    from logbook import Logger
except ImportError:
    class Logger(object):
        def __init__(self, name, level=0):
            self.name = name
            self.level = level
```

```
        debug = info = warn = warning = notice = error = exception = \
            critical = log = lambda *a, **kw: None

log = Logger('My library')
```

1.6.2 Best Practices

- A library that wants to log to the Logbook system should generally be designed to provide an interface to the record dispatchers it is using. That does not have to be a reference to the record dispatcher itself, it is perfectly fine if there is a toggle to switch it on or off.
- The channel name should be readable and descriptive.
- For example, if you are a database library that wants to use the logging system to log all SQL statements issued in debug mode, you can enable and disable your record dispatcher based on that debug flag.
- Libraries should never set up log setups except temporarily on a per-thread basis if it never changes the stack for a longer duration than a function call in a library. For example, hooking in a null handler for a call to a noisy function is fine, changing the global stack in a function and not reverting it at the end of the function is bad.

1.6.3 Debug Loggers

Sometimes you want to have loggers in place that are only really good for debugging. For example you might have a library that does a lot of server/client communication and for debugging purposes it would be nice if you can enable/disable that log output as necessary.

In that case it makes sense to create a logger and disable that by default and give people a way to get hold of the logger to flip the flag. Additionally you can override the `disabled` flag to automatically set it based on another value:

```
class MyLogger(Logger):
    @property
    def disabled(self):
        return not database_connection.debug
database_connection.logger = MyLogger('mylibrary.dbconnection')
```

1.7 Unittesting Support

Logbook has builtin support for testing logging calls. There is a handler that can be hooked in and will catch all log records for inspection. Not only that, it also provides methods to test if certain things were logged.

1.7.1 Basic Setup

The interface to satisfaction is `logbook.TestHandler`. Create it, and bind it, and you're done. If you are using classic `unittest` test cases, you might want to set it up in the before and after callback methods:

```
import logbook
import unittest

class LoggingTestCase(unittest.TestCase):

    def setUp(self):
        self.log_handler = logbook.TestHandler()
```

```
        self.log_handler.push_thread()

    def tearDown(self):
        self.log_handler.pop_thread()
```

Alternatively you can also use it in a with statement in an individual test. This is also how this can work in nose and other testing systems:

```
def my_test():
    with logbook.TestHandler() as log_handler:
        ...
```

1.7.2 Test Handler Interface

The test handler has a few attributes and methods to gain access to the logged messages. The most important ones are `records` and `formatted_records`. The first is a list of the captured `LogRecords`, the second a list of the formatted records as unicode strings:

```
>>> from logbook import TestHandler, Logger
>>> logger = Logger('Testing')
>>> handler = TestHandler()
>>> handler.push_thread()
>>> logger.warn('Hello World')
>>> handler.records
[<logbook.base.LogRecord object at 0x100640cd0>]
>>> handler.formatted_records
[u'[WARNING] Testing: Hello World']
```

1.7.3 Probe Log Records

The handler also provide some convenience methods to do assertions:

```
>>> handler.has_warnings
True
>>> handler.has_errors
False
>>> handler.has_warning('Hello World')
True
```

Methods like `has_warning()` accept two arguments:

message If provided and not *None* it will check if there is at least one log record where the message matches.

channel If provided and not *None* it will check if there is at least one log record where the logger name of the record matches.

Example usage:

```
>>> handler.has_warning('A different message')
False
>>> handler.has_warning('Hello World', channel='Testing')
True
>>> handler.has_warning(channel='Testing')
True
```

1.8 Logging to Tickets

Logbook supports the concept of creating unique tickets for log records and keeping track of the number of times these log records were created. The default implementation logs into a relational database, but there is a baseclass that can be subclassed to log into existing ticketing systems such as trac or other data stores.

The ticketing handlers and store backends are all implemented in the module *logbook.ticketing*.

1.8.1 How does it work?

When a ticketing handler is used each call to a logbook logger is assigned a unique hash that is based on the name of the logger, the location of the call as well as the level of the message. The message itself is not taken into account as it might be changing depending on the arguments passed to it.

Once that unique hash is created the database is checked if there is already a ticket for that hash. If there is, a new occurrence is logged with all details available. Otherwise a new ticket is created.

This makes it possible to analyze how often certain log messages are triggered and over what period of time.

1.8.2 Why should I use it?

The ticketing handlers have the big advantage over a regular log handler that they will capture the full data of the log record in machine processable format. Whatever information was attached to the log record will be send straight to the data store in JSON.

This makes it easier to track down issues that might happen in production systems. Due to the higher overhead of ticketing logging over a standard logfile or something comparable it should only be used for higher log levels (`WARNING` or higher).

1.8.3 Common Setups

The builtin ticketing handler is called *TicketingHandler*. In the default configuration it will connect to a relational database with the help of SQLAlchemy and log into two tables there: tickets go into `${prefix}tickets` and occurrences go into `${prefix}occurrences`. The default table prefix is `'logbook_'` but can be overriden. If the tables do not exist already, the handler will create them.

Here an example setup that logs into a postgres database:

```
from logbook import ERROR
from logbook.ticketing import TicketingHandler
handler = TicketingHandler('postgres://localhost/database',
                           level=ERROR)
with handler:
    # everything in this block and thread will be handled by
    # the ticketing database handler
    ...
```

Alternative backends can be swapped in by providing the *backend* parameter. There is a second implementation of a backend that is using MongoDB: *MongoDBBackend*.

1.9 Logging Compatibility

Logbook provides backwards compatibility with the logging library. When activated, the logging library will transparently redirect all the logging calls to your Logbook logging setup.

1.9.1 Basic Setup

If you import the compat system and call the *redirect_logging()* function, all logging calls that happen after this call will transparently be redirected to Logbook:

```
from logbook.compat import redirect_logging
redirect_logging()
```

This also means you don't have to call `logging.basicConfig()`:

```
>>> from logbook.compat import redirect_logging
>>> redirect_logging()
>>> from logging import getLogger
>>> log = getLogger('My Logger')
>>> log.warn('This is a warning')
[2010-07-25 00:24] WARNING: My Logger: This is a warning
```

1.9.2 Advanced Setup

The way this is implemented is with a *RedirectLoggingHandler*. This class is a handler for the old logging system that sends records via an internal logbook logger to the active logbook handlers. This handler can then be added to specific logging loggers if you want:

```
>>> from logging import getLogger
>>> mylog = getLogger('My Log')
>>> from logbook.compat import RedirectLoggingHandler
>>> mylog.addHandler(RedirectLoggingHandler())
>>> otherlog = getLogger('Other Log')
>>> otherlog.warn('logging is deprecated')
No handlers could be found for logger "Other Log"
>>> mylog.warn('but logbook is awesome')
[2010-07-25 00:29] WARNING: My Log: but logbook is awesome
```

1.9.3 Reverse Redirects

You can also redirect logbook records to logging, so the other way round. For this you just have to activate the *LoggingHandler* for the thread or application:

```
from logbook import Logger
from logbook.compat import LoggingHandler

log = Logger('My app')
with LoggingHandler():
    log.warn('Going to logging')
```

1.10 API Documentation

This part of the documentation documents all the classes and functions provided by Logbook.

1.10.1 Core Interface

This implements the core interface.

class `logbook.base.`**`Logger`**(*name=None*, *level=0*)

Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer.

Names used by logbook should be descriptive and are intended for user display, not for filtering. Filtering should happen based on the context information instead.

A logger internally is a subclass of a *RecordDispatcher* that implements the actual logic. If you want to implement a custom logger class, have a look at the interface of that class as well.

`call_handlers`(*record*)

Pass a record to all relevant handlers in the following order:

- per-dispatcher handlers are handled first
- afterwards all the current context handlers in the order they were pushed

Before the first handler is invoked, the record is processed (*process_record()*).

`catch_exceptions`(**args*, ***kwargs*)

A context manager that catches exceptions and calls *exception()* for exceptions caught that way. Example:

```
with logger.catch_exceptions():
    execute_code_that_might_fail()
```

`critical`(**args*, ***kwargs*)

Logs a `LogRecord` with the level set to `CRITICAL`.

`debug`(**args*, ***kwargs*)

Logs a `LogRecord` with the level set to `DEBUG`.

`error`(**args*, ***kwargs*)

Logs a `LogRecord` with the level set to `ERROR`.

`exception`(**args*, ***kwargs*)

Works exactly like *error()* just that the message is optional and exception information is recorded.

`handle`(*record*)

Call the handlers for the specified record. This is invoked automatically when a record should be handled. The default implementation checks if the dispatcher is disabled and if the record level is greater than the level of the record dispatcher. In that case it will call the handlers (*call_handlers()*).

`info`(**args*, ***kwargs*)

Logs a `LogRecord` with the level set to `INFO`.

`level_name`

The level as unicode string

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

Logs a `LogRecord` with the level set to the *level* parameter. Because custom levels are not supported by logbook, this method is mainly used to avoid the use of reflection (e.g.: `getattr()`) for programmatic logging.

make_record_and_handle(*level*, *msg*, *args*, *kwargs*, *exc_info*, *extra*)
Creates a record from some given arguments and heads it over to the handling system.

notice(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `NOTICE`.

process_record(*record*)
Processes the record with all context specific processors. This can be overriden to also inject additional information as necessary that can be provided by this record dispatcher.

warn(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `WARNING`. This function has an alias named *warning()*.

warning(**args*, ***kwargs*)
Alias for *warn()*.

class logbook.base.**LoggerGroup**(*loggers=None*, *level=0*, *processor=None*)
A LoggerGroup represents a group of loggers. It cannot emit log messages on its own but it can be used to set the disabled flag and log level of all loggers in the group.

Furthermore the *process_record()* method of the group is called by any logger in the group which by default calls into the *processor* callback function.

add_logger(*logger*)
Adds a logger to this group.

disabled = None
the disabled flag for all loggers in the group, unless the loggers overrode the setting.

level = None
the level of the group. This is reflected to the loggers in the group unless they overrode the setting.

loggers = None
a list of all loggers on the logger group. Use the *add_logger()* and *remove_logger()* methods to add or remove loggers from this list.

process_record(*record*)
Like *Logger.process_record()* but for all loggers in the group. By default this calls into the *processor* function is it's not *None*.

processor = None
an optional callback function that is executed to process the log records of all loggers in the group.

remove_logger(*logger*)
Removes a logger from the group.

class logbook.base.**LogRecord**(*channel*, *level*, *msg*, *args=None*, *kwargs=None*, *exc_info=None*, *extra=None*, *frame=None*, *dispatcher=None*)
A LogRecord instance represents an event being logged.

LogRecord instances are created every time something is logged. They contain all the information pertinent to the event being logged. The main information passed in is in msg and args

args = None
the positional arguments for the format string.

calling_frame
The frame in which the record has been created. This only exists for as long the log record is not closed.

channel = None
the name of the logger that created it or any other textual channel description. This is a descriptive name and can be used for filtering.

close()
Closes the log record. This will set the frame and calling frame to *None* and frame-related information will no longer be available unless it was pulled in first (*pull_information()*). This makes a log record safe for pickling and will clean up memory that might be still referenced by the frames.

dispatcher
The dispatcher that created the log record. Might not exist because a log record does not have to be created from a logger or other dispatcher to be handled by logbook. If this is set, it will point to an object that implements the *RecordDispatcher* interface.

exc_info = None
optional exception information. If set, this is a tuple in the form `(exc_type, exc_value, tb)` as returned by `sys.exc_info()`.

exception_message
The message of the exception.

exception_name
The name of the exception.

exception_shortname
An abbreviated exception name (no import path)

extra = None
optional extra information as dictionary. This is the place where custom log processors can attach custom context sensitive data.

filename
The filename of the module in which the record has been created. Requires a frame or that *pull_information()* was called before.

formatted_exception
The formatted exception which caused this record to be created in case there was any.

frame = None
If available, optionally the interpreter frame that pulled the heavy init. This usually points to somewhere in the dispatcher. Might not be available for all calls and is removed when the log record is closed.

classmethod from_dict(*d*)
Creates a log record from an exported dictionary. This also supports JSON exported dictionaries.

func_name
The name of the function that triggered the log call if available. Requires a frame or that *pull_information()* was called before.

heavy_init()
Does the heavy initialization that could be expensive. This must not be called from a higher stack level than when the log record was created and the later the initialization happens, the more off the date information will be for example.

This is internally used by the record dispatching system and usually something not to worry about.

heavy_initialized = False
a flag that is *True* if the log record is heavy initialized which is not the case by default.

information_pulled = False
a flag that is *True* when all the information was pulled from the information that becomes unavailable on close.

keep_open = False
can be overriden by a handler to not close the record. This could lead to memory leaks so it should be used carefully.

kwargs = None
the keyword arguments for the format string.

late = False
a flag that is *True* when heavy initialization is no longer possible

level = None
the level of the log record as integer.

level_name
The level as unicode string

lineno
The line number of the file in which the record has been created. Requires a frame or that *pull_information()* was called before.

message
The formatted message.

module
The name of the module that triggered the log call if available. Requires a frame or that *pull_information()* was called before.

msg = None
The message of the log record as new-style format string.

process = None
the PID of the current process

process_name
The name of the process in which the record has been created.

pull_information()
A helper function that pulls all frame-related information into the object so that this information is available after the log record was closed.

thread
The ident of the thread. This is evaluated late and means that if the log record is passed to another thread, *pull_information()* was called in the old thread.

thread_name
The name of the thread. This is evaluated late and means that if the log record is passed to another thread, *pull_information()* was called in the old thread.

time = None
the time of the log record creation as `datetime.datetime` object. This information is unavailable until the record was heavy initialized.

to_dict(*json_safe=False*)
Exports the log record into a dictionary without the information that cannot be safely serialized like interpreter frames and tracebacks.

update_from_dict(*d*)
Like the *from_dict()* classmethod, but will update the instance in place. Helpful for constructors.

class logbook.base.**Flags**(***flags*)
Allows flags to be pushed on a flag stack. Currently two flags are available:

errors Can be set to override the current error behaviour. This value is used when logging calls fail. The default behaviour is spitting out the stacktrace to stderr but this can be overridden:

`'silent'`	fail silently
`'raise'`	raise a catchable exception
`'print'`	print the stacktrace to stderr (default)

introspection Can be used to disable frame introspection. This can give a speedup on production systems if you are using a JIT compiled Python interpreter such as pypy. The default is *True*.

Note that the default setup of some of the handler (mail for instance) includes frame dependent information which will not be available when introspection is disabled.

Example usage:

```
with Flags(errors='silent'):
    ...
```

applicationbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the application.

static get_flag(*flag*, *default=None*)
Looks up the current value of a specific flag.

pop_application()
Pops the context object from the stack.

pop_thread()
Pops the context object from the stack.

push_application()
Pushes the context object to the application stack.

push_thread()
Pushes the context object to the thread stack.

threadbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the thread.

class logbook.base.**Processor**(*callback=None*)
Can be pushed to a stack to inject additional information into a log record as necessary:

```
def inject_ip(record):
    record.extra['ip'] = '127.0.0.1'

with Processor(inject_ip):
    ...
```

applicationbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the application.

callback = None
the callback that was passed to the constructor

pop_application()
Pops the context object from the stack.

pop_thread()
Pops the context object from the stack.

process(*record*)
Called with the log record that should be overridden. The default implementation calls `callback` if it is not *None*.

push_application()
Pushes the context object to the application stack.

push_thread()
Pushes the context object to the thread stack.

threadbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the thread.

logbook.base.**get_level_name**(*level*)
Return the textual representation of logging level 'level'.

logbook.base.**lookup_level**(*level*)
Return the integer representation of a logging level.

logbook.base.**CRITICAL**
logbook.base.**ERROR**
logbook.base.**WARNING**
logbook.base.**INFO**
logbook.base.**DEBUG**
logbook.base.**NOTSET**
The log level constants

1.10.2 Handlers

This documents the base handler interface as well as the provided core handlers. There are additional handlers for special purposes in the *logbook.more*, *logbook.ticketing* and *logbook.queues* modules.

Base Interface

class logbook.handlers.**Handler**(*level=0*, *filter=None*, *bubble=False*)
Handler instances dispatch logging events to specific destinations.

The base handler class. Acts as a placeholder which defines the Handler interface. Handlers can optionally use Formatter instances to format records as desired. By default, no formatter is specified; in this case, the 'raw' message as determined by record.message is logged.

To bind a handler you can use the *push_application()* and *push_thread()* methods. This will push the handler on a stack of handlers. To undo this, use the *pop_application()* and *pop_thread()* methods:

```
handler = MyHandler()
handler.push_application()
# all here goes to that handler
handler.pop_application()
```

By default messages send to that handler will not go to a handler on an outer level on the stack, if handled. This can be changed by setting bubbling to *True*. This setup for example would not have any effect:

```
handler = NullHandler(bubble=False)
handler.push_application()
```

Whereas this setup disables all logging for the application:

```
handler = NullHandler()
handler.push_application()
```

There are also context managers to setup the handler for the duration of a *with*-block:

```
with handler.applicationbound():
    ...

with handler.threadbound():
    ...
```

Because *threadbound* is a common operation, it is aliased to a with on the handler itself:

```
with handler:
    ...
```

applicationbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the application.

blackhole = False
a flag for this handler that can be set to *True* for handlers that are consuming log records but are not actually displaying it. This flag is set for the `NullHandler` for instance.

bubble = None
the bubble flag of this handler

close()
Tidy up any resources used by the handler. This is automatically called by the destructor of the class as well, but explicit calls are encouraged. Make sure that multiple calls to close are possible.

emit(*record*)
Emit the specified logging record. This should take the record and deliver it to whereever the handler sends formatted log records.

emit_batch(*records*, *reason*)
Some handlers may internally queue up records and want to forward them at once to another handler. For example the `FingersCrossedHandler` internally buffers records until a level threshold is reached in which case the buffer is sent to this method and not `emit()` for each record.

The default behaviour is to call `emit()` for each record in the buffer, but handlers can use this to optimize log handling. For instance the mail handler will try to batch up items into one mail and not to emit mails for each record in the buffer.

Note that unlike `emit()` there is no wrapper method like `handle()` that does error handling. The reason is that this is intended to be used by other handlers which are already protected against internal breakage.

reason is a string that specifies the rason why `emit_batch()` was called, and not `emit()`. The following are valid values:

'buffer' Records were buffered for performance reasons or because the records were sent to another process and buffering was the only possible way. For most handlers this should be equivalent to calling `emit()` for each record.

'escalation' Escalation means that records were buffered in case the threshold was exceeded. In this case, the last record in the iterable is the record that triggered the call.

'group' All the records in the iterable belong to the same logical component and happened in the same process. For example there was a long running computation and the handler is invoked with a bunch of records that happened there. This is similar to the escalation reason, just that the first one is the significant one, not the last.

If a subclass overrides this and does not want to handle a specific reason it must call into the superclass because more reasons might appear in future releases.

Example implementation:

```
        def emit_batch(self, records, reason):
            if reason not in ('escalation', 'group'):
                Handler.emit_batch(self, records, reason)
            ...
```

filter = None
the filter to be used with this handler

format(*record*)
Formats a record with the given formatter. If no formatter is set, the record message is returned. Generally speaking the return value is most likely a unicode string, but nothing in the handler interface requires a formatter to return a unicode string.

The combination of a handler and formatter might have the formatter return an XML element tree for example.

formatter = None
the formatter to be used on records. This is a function that is passed a log record as first argument and the handler as second and returns something formatted (usually a unicode string)

handle(*record*)
Emits the record and falls back. It tries to *emit()* the record and if that fails, it will call into *handle_error()* with the record and traceback. This function itself will always emit when called, even if the logger level is higher than the record's level.

If this method returns *False* it signals to the calling function that no recording took place in which case it will automatically bubble. This should not be used to signal error situations. The default implementation always returns *True*.

handle_error(*record*, *exc_info*)
Handle errors which occur during an emit() call. The behaviour of this function depends on the current *errors* setting.

Check `Flags` for more information.

level = None
the level for the handler. Defaults to *NOTSET* which consumes all entries.

level_name
The level as unicode string

pop_application()
Pops the context object from the stack.

pop_thread()
Pops the context object from the stack.

push_application()
Pushes the context object to the application stack.

push_thread()
Pushes the context object to the thread stack.

should_handle(*record*)
Returns *True* if this handler wants to handle the record. The default implementation checks the level.

threadbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the thread.

class logbook.base.**NestedSetup**(*objects=None*)
A nested setup can be used to configure multiple handlers and processors at once.

class logbook.handlers.**StringFormatter**(*format_string*)
Many handlers format the log entries to text format. This is done by a callable that is passed a log record and returns an unicode string. The default formatter for this is implemented as a class so that it becomes possible to hook into every aspect of the formatting process.

Core Handlers

class logbook.handlers.**StreamHandler**(*stream*, *level=0*, *format_string=None*, *encoding=None*, *filter=None*, *bubble=False*)
a handler class which writes logging records, appropriately formatted, to a stream. note that this class does not close the stream, as sys.stdout or sys.stderr may be used.

If a stream handler is used in a *with* statement directly it will *close()* on exit to support this pattern:

```
with StreamHandler(my_stream):
    pass
```

Notes on the encoding

On Python 3, the encoding parameter is only used if a stream was passed that was opened in binary mode.

---

close()
The default stream handler implementation is not to close the wrapped stream but to flush it.

flush()
Flushes the inner stream.

format_and_encode(*record*)
Formats the record and encodes it to the stream encoding.

write(*item*)
Writes a bytestring to the stream.

class logbook.handlers.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *level=0*, *format_string=None*, *delay=False*, *filter=None*, *bubble=False*)
A handler that does the task of opening and closing files for you. By default the file is opened right away, but you can also *delay* the open to the point where the first message is written.

This is useful when the handler is used with a FingersCrossedHandler or something similar.

class logbook.handlers.**MonitoringFileHandler**(*filename*, *mode='a'*, *encoding='utf-8'*, *level=0*, *format_string=None*, *delay=False*, *filter=None*, *bubble=False*)
A file handler that will check if the file was moved while it was open. This might happen on POSIX systems if an application like logrotate moves the logfile over.

Because of different IO concepts on Windows, this handler will not work on a windows system.

class logbook.handlers.**StderrHandler**(*level=0*, *format_string=None*, *filter=None*, *bubble=False*)
A handler that writes to what is currently at stderr. At the first glace this appears to just be a *StreamHandler* with the stream set to sys.stderr but there is a difference: if the handler is created globally and sys.stderr changes later, this handler will point to the current *stderr*, whereas a stream handler would still point to the old one.

class logbook.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *encoding='utf-8'*, *level=0*, *format_string=None*, *delay=False*, *max_size=1048576*, *backup_count=5*, *filter=None*, *bubble=False*)
This handler rotates based on file size. Once the maximum size is reached it will reopen the file and start with an

empty file again. The old file is moved into a backup copy (named like the file, but with a `.backupnumber` appended to the file. So if you are logging to `mail` the first backup copy is called `mail.1`.)

The default number of backups is 5. Unlike a similar logger from the logging package, the backup count is mandatory because just reopening the file is dangerous as it deletes the log without asking on rollover.

class `logbook.handlers.`**`TimedRotatingFileHandler`**(*filename*, *mode='a'*, *encoding='utf-8'*, *level=0*, *format_string=None*, *date_format='%Y-%m-%d'*, *backup_count=0*, *filter=None*, *bubble=False*)

This handler rotates based on dates. It will name the file after the filename you specify and the *date_format* pattern.

So for example if you configure your handler like this:

```
handler = TimedRotatingFileHandler('/var/log/foo.log',
                                   date_formnat='%Y-%m-%d')
```

The filenames for the logfiles will look like this:

```
/var/log/foo-2010-01-10.log
/var/log/foo-2010-01-11.log
...
```

By default it will keep all these files around, if you want to limit them, you can specify a *backup_count*.

`files_to_delete`()
Returns a list with the files that have to be deleted when a rollover occours.

class `logbook.handlers.`**`TestHandler`**(*level=0*, *format_string=None*, *filter=None*, *bubble=False*)

Like a stream handler but keeps the values in memory. This logger provides some ways to test for the records in memory.

Example usage:

```
def my_test():
    with logbook.TestHandler() as handler:
        logger.warn('A warning')
        assert logger.has_warning('A warning')
        ...
```

`close`()
Close all records down when the handler is closed.

`formatted_records`
Captures the formatted log records as unicode strings.

`has_critical`(**args*, ***kwargs*)
True if a specific `CRITICAL` log record exists.

See *Probe Log Records* for more information.

`has_criticals`
True if any `CRITICAL` records were found.

`has_debug`(**args*, ***kwargs*)
True if a specific `DEBUG` log record exists.

See *Probe Log Records* for more information.

`has_debugs`
True if any `DEBUG` records were found.

has_error(**args*, ***kwargs*)
True if a specific `ERROR` log record exists.

See *Probe Log Records* for more information.

has_errors
True if any `ERROR` records were found.

has_info(**args*, ***kwargs*)
True if a specific `INFO` log record exists.

See *Probe Log Records* for more information.

has_infos
True if any `INFO` records were found.

has_notice(**args*, ***kwargs*)
True if a specific `NOTICE` log record exists.

See *Probe Log Records* for more information.

has_notices
True if any `NOTICE` records were found.

has_warning(**args*, ***kwargs*)
True if a specific `WARNING` log record exists.

See *Probe Log Records* for more information.

has_warnings
True if any `WARNING` records were found.

records = None
captures the `LogRecord`s as instances

class `logbook.handlers.`**MailHandler**(*from_addr*, *recipients*, *subject=None*, *server_addr=None*, *credentials=None*, *secure=None*, *record_limit=None*, *record_delta=None*, *level=0*, *format_string=None*, *related_format_string=None*, *filter=None*, *bubble=False*)
A handler that sends error mails. The format string used by this handler are the contents of the mail plus the headers. This is handy if you want to use a custom subject or `X-` header:

```
handler = MailHandler(format_string='''        Subject: {record.level_name} on My Application

{record.message}
{record.extra[a_custom_injected_record]}
''')
```

This handler will always emit text-only mails for maximum portability and best performance.

In the default setting it delivers all log records but it can be set up to not send more than n mails for the same record each hour to not overload an inbox and the network in case a message is triggered multiple times a minute. The following example limits it to 60 mails an hour:

```
from datetime import timedelta
handler = MailHandler(record_limit=1,
                      record_delta=timedelta(minutes=1))
```

The default timedelta is 60 seconds (one minute).

The mail handler is sending mails in a blocking manner. If you are not using some centralized system for logging these messages (with the help of ZeroMQ or others) and the logging system slows you down you can

wrap the handler in a *logbook.queues.ThreadedWrapperHandler* that will then send the mails in a background thread.

Changed in version 0.3: The handler supports the batching system now.

close_connection(*con*)
Closes the connection that was returned by *get_connection()*.

collapse_mails(*mail*, *related*, *reason*)
When escaling or grouped mails are

deliver(*msg*, *recipients*)
Delivers the given message to a list of recpients.

format_related_record(*record*)
Used for format the records that led up to another record or records that are related into strings. Used by the batch formatter.

generate_mail(*record*, *suppressed=0*)
Generates the final email (`email.message.Message`) with headers and date. *suppressed* is the number of mails that were not send if the *record_limit* feature is active.

get_connection()
Returns an SMTP connection. By default it reconnects for each sent mail.

get_recipients(*record*)
Returns the recipients for a record. By default the `recipients` attribute is returned for all records.

max_record_cache = 512
the maximum number of record hashes in the cache for the limiting feature. Afterwards, record_cache_prune percent of the oldest entries are removed

message_from_record(*record*, *suppressed*)
Creates a new message for a record as email message object (`email.message.Message`). *suppressed* is the number of mails not sent if the *record_limit* feature is active.

record_cache_prune = 0.333
the number of items to prune on a cache overflow in percent.

class logbook.handlers.**SyslogHandler**(*application_name=None*, *address=None*, *facility='user'*, *socktype=2*, *level=0*, *format_string=None*, *filter=None*, *bubble=False*)
A handler class which sends formatted logging records to a syslog server. By default it will send to it via unix socket.

class logbook.handlers.**NTEventLogHandler**(*application_name*, *log_type='Application'*, *level=0*, *format_string=None*, *filter=None*, *bubble=False*)
A handler that sends to the NT event log system.

unregister_logger()
Removes the application binding from the registry. If you call this, the log viewer will no longer be able to provide any information about the message.

class logbook.handlers.**NullHandler**(*level=0*, *filter=None*, *bubble=False*)
A handler that does nothing, meant to be inserted in a handler chain with `bubble=False` to stop further processing.

class logbook.handlers.**WrapperHandler**(*handler*)
A class that can wrap another handler and redirect all calls to the wrapped handler:

```
handler = WrapperHandler(other_handler)
```

Subclasses should override the `_direct_attrs` attribute as necessary.

logbook.handlers.**create_syshandler**(*application_name*, *level=0*)
Creates the handler the operating system provides. On Unix systems this creates a *SyslogHandler*, on Windows sytems it will create a *NTEventLogHandler*.

Special Handlers

class logbook.handlers.**FingersCrossedHandler**(*handler*, *action_level=5*, *buffer_size=0*, *pull_information=True*, *reset=False*, *filter=None*, *bubble=False*)
This handler wraps another handler and will log everything in memory until a certain level (*action_level*, defaults to *ERROR*) is exceeded. When that happens the fingers crossed handler will activate forever and log all buffered records as well as records yet to come into another handled which was passed to the constructor.

Alternatively it's also possible to pass a factory function to the constructor instead of a handler. That factory is then called with the triggering log entry and the finger crossed handler to create a handler which is then cached.

The idea of this handler is to enable debugging of live systems. For example it might happen that code works perfectly fine 99% of the time, but then some exception happens. But the error that caused the exception alone might not be the interesting bit, the interesting information were the warnings that lead to the error.

Here a setup that enables this for a web application:

```
from logbook import FileHandler
from logbook import FingersCrossedHandler

def issue_logging():
    def factory(record, handler):
        return FileHandler('/var/log/app/issue-%s.log' % record.time)
    return FingersCrossedHandler(factory)

def application(environ, start_response):
    with issue_logging():
        return the_actual_wsgi_application(environ, start_response)
```

Whenever an error occours, a new file in `/var/log/app` is created with all the logging calls that lead up to the error up to the point where the *with* block is exited.

Please keep in mind that the `FingersCrossedHandler` handler is a one-time handler. Once triggered, it will not reset. Because of that you will have to re-create it whenever you bind it. In this case the handler is created when it's bound to the thread.

Due to how the handler is implemented, the filter, bubble and level flags of the wrapped handler are ignored.

Changed in version 0.3.

The default behaviour is to buffer up records and then invoke another handler when a severity theshold was reached with the buffer emitting. This now enables this logger to be properly used with the `MailHandler`. You will now only get one mail for each bfufered record. However once the threshold was reached you would still get a mail for each record which is why the *reset* flag was added.

When set to *True*, the handler will instantly reset to the untriggered state and start buffering again:

```
handler = FingersCrossedHandler(MailHandler(...),
                                buffer_size=10,
                                reset=True)
```

New in version 0.3: The *reset* flag was added.

batch_emit_reason = 'escalation'
the reason to be used for the batch emit. The default is `'escalation'`.

New in version 0.3.

buffer_size = None
the maximum number of entries in the buffer. If this is exhausted the oldest entries will be discarded to make place for new ones

buffered_records = None
the buffered records of the handler. Once the action is triggered (*triggered*) this list will be None. This attribute can be helpful for the handler factory function to select a proper filename (for example time of first log record)

triggered
This attribute is *True* when the action was triggered. From this point onwards the finger crossed handler transparently forwards all log records to the inner handler. If the handler resets itself this will always be *False*.

class logbook.handlers.**GroupHandler**(*handler*, *pull_information=True*)
A handler that buffers all messages until it is popped again and then forwards all messages to another handler. This is useful if you for example have an application that does computations and only a result mail is required. A group handler makes sure that only one mail is sent and not multiple. Some other handles might support this as well, though currently none of the builtins do.

Example:

```
with GroupHandler(MailHandler(...)):
    # everything here ends up in the mail
```

The *GroupHandler* is implemented as a *WrapperHandler* thus forwarding all attributes of the wrapper handler.

Notice that this handler really only emit the records when the handler is popped from the stack.

New in version 0.3.

Mixin Classes

class logbook.handlers.**StringFormatterHandlerMixin**(*format_string*)
A mixin for handlers that provides a default integration for the `StringFormatter` class. This is used for all handlers by default that log text to a destination.

default_format_string = u'[{record.time:%Y-%m-%d %H:%M}] {record.level_name}: {record.channel}: {record
a class attribute for the default format string to use if the constructor was invoked with *None*.

format_string
the currently attached format string as new-style format string.

formatter_class
the class to be used for string formatting

alias of *StringFormatter*

class logbook.handlers.**HashingHandlerMixin**
Mixin class for handlers that are hashing records.

hash_record(*record*)
Returns a hash for a record to keep it apart from other records. This is used for the *record_limit* feature. By default The level, channel, filename and location are hashed.

Calls into *hash_record_raw()*.

hash_record_raw(*record*)
Returns a hashlib object with the hash of the record.

class logbook.handlers.**LimitingHandlerMixin**(*record_limit*, *record_delta*)
Mixin class for handlers that want to limit emitting records.

In the default setting it delivers all log records but it can be set up to not send more than n mails for the same record each hour to not overload an inbox and the network in case a message is triggered multiple times a minute. The following example limits it to 60 mails an hour:

```
from datetime import timedelta
handler = MailHandler(record_limit=1,
                      record_delta=timedelta(minutes=1))
```

check_delivery(*record*)
Helper function to check if data should be delivered by this handler. It returns a tuple in the form `(suppression_count, allow)`. The first one is the number of items that were not delivered so far, the second is a boolean flag if a delivery should happen now.

1.10.3 Utilities

This documents general purpose utility functions available in Logbook.

logbook.**debug**(*self*, **args*, ***kwargs*)
Logs a LogRecord with the level set to DEBUG.

logbook.**info**(*self*, **args*, ***kwargs*)
Logs a LogRecord with the level set to INFO.

logbook.**warn**(*self*, **args*, ***kwargs*)
Logs a LogRecord with the level set to WARNING. This function has an alias named *warning()*.

logbook.**warning**(*self*, **args*, ***kwargs*)
Alias for *warn()*.

logbook.**notice**(*self*, **args*, ***kwargs*)
Logs a LogRecord with the level set to NOTICE.

logbook.**error**(*self*, **args*, ***kwargs*)
Logs a LogRecord with the level set to ERROR.

logbook.**exception**(*self*, **args*, ***kwargs*)
Works exactly like *error()* just that the message is optional and exception information is recorded.

logbook.**catch_exceptions**(*self*, **args*, ***kwargs*)
A context manager that catches exceptions and calls *exception()* for exceptions caught that way. Example:

```
with logger.catch_exceptions():
    execute_code_that_might_fail()
```

logbook.**critical**(*self*, **args*, ***kwargs*)
Logs a LogRecord with the level set to CRITICAL.

logbook.**log**(*self*, *level*, **args*, ***kwargs*)
Logs a LogRecord with the level set to the *level* parameter. Because custom levels are not supported by logbook, this method is mainly used to avoid the use of reflection (e.g.: getattr()) for programmatic logging.

1.10.4 Queue Support

The queue support module makes it possible to add log records to a queue system. This is useful for distributed setups where you want multiple processes to log to the same backend. Currently supported are ZeroMQ as well as the `multiprocessing` `Queue` class.

ZeroMQ

class `logbook.queues.`**`ZeroMQHandler`**(*uri=None*, *level=0*, *filter=None*, *bubble=False*, *context=None*)

A handler that acts as a ZeroMQ publisher, which publishes each record as json dump. Requires the pyzmq library.

The queue will be filled with JSON exported log records. To receive such log records from a queue you can use the *ZeroMQSubscriber*.

Example setup:

```
handler = ZeroMQHandler('tcp://127.0.0.1:5000')
```

`context` = **None**
the zero mq context

`export_record`(*record*)
Exports the record into a dictionary ready for JSON dumping.

`socket` = **None**
the zero mq socket.

class `logbook.queues.`**`ZeroMQSubscriber`**(*uri=None*, *context=None*)

A helper that acts as ZeroMQ subscriber and will dispatch received log records to the active handler setup. There are multiple ways to use this class.

It can be used to receive log records from a queue:

```
subscriber = ZeroMQSubscriber('tcp://127.0.0.1:5000')
record = subscriber.recv()
```

But it can also be used to receive and dispatch these in one go:

```
with target_handler:
    subscriber = ZeroMQSubscriber('tcp://127.0.0.1:5000')
    subscriber.dispatch_forever()
```

This will take all the log records from that queue and dispatch them over to *target_handler*. If you want you can also do that in the background:

```
subscriber = ZeroMQSubscriber('tcp://127.0.0.1:5000')
controller = subscriber.dispatch_in_background(target_handler)
```

The controller returned can be used to shut down the background thread:

```
controller.stop()
```

`close`()
Closes the zero mq socket.

`context` = **None**
the zero mq context

dispatch_forever()
Starts a loop that dispatches log records forever.

dispatch_in_background(*setup=None*)
Starts a new daemonized thread that dispatches in the background. An optional handler setup can be provided that pushed to the new thread (can be any *logbook.base.StackedObject*).

Returns a *ThreadController* object for shutting down the background thread. The background thread will already be running when this function returns.

dispatch_once(*timeout=None*)
Receives one record from the socket, loads it and dispatches it. Returns *True* if something was dispatched or *False* if it timed out.

recv(*timeout=None*)
Receives a single record from the socket. Timeout of 0 means nonblocking, *None* means blocking and otherwise it's a timeout in seconds after which the function just returns with *None*.

socket = None
the zero mq socket.

MultiProcessing

class logbook.queues.**MultiProcessingHandler**(*queue*, *level=0*, *filter=None*, *bubble=False*)
Implements a handler that dispatches over a queue to a different process. It is connected to a subscriber with a multiprocessing.Queue:

```
from multiprocessing import Queue
from logbook.queues import MultiProcessingHandler
queue = Queue(-1)
handler = MultiProcessingHandler(queue)
```

class logbook.queues.**MultiProcessingSubscriber**(*queue=None*)
Receives log records from the given multiprocessing queue and dispatches them to the active handler setup. Make sure to use the same queue for both handler and subscriber. Idaelly the queue is set up with maximum size (-1):

```
from multiprocessing import Queue
queue = Queue(-1)
```

It can be used to receive log records from a queue:

```
subscriber = MultiProcessingSubscriber(queue)
record = subscriber.recv()
```

But it can also be used to receive and dispatch these in one go:

```
with target_handler:
    subscriber = MultiProcessingSubscriber(queue)
    subscriber.dispatch_forever()
```

This will take all the log records from that queue and dispatch them over to *target_handler*. If you want you can also do that in the background:

```
subscriber = MultiProcessingSubscriber(queue)
controller = subscriber.dispatch_in_background(target_handler)
```

The controller returned can be used to shut down the background thread:

```
    controller.stop()
```

If no queue is provided the subscriber will create one. This one can the be used by handlers:

```
subscriber = MultiProcessingSubscriber()
handler = MultiProcessingHandler(subscriber.queue)
```

dispatch_forever()
Starts a loop that dispatches log records forever.

dispatch_in_background(*setup=None*)
Starts a new daemonized thread that dispatches in the background. An optional handler setup can be provided that pushed to the new thread (can be any *logbook.base.StackedObject*).

Returns a *ThreadController* object for shutting down the background thread. The background thread will already be running when this function returns.

dispatch_once(*timeout=None*)
Receives one record from the socket, loads it and dispatches it. Returns *True* if something was dispatched or *False* if it timed out.

Other

class logbook.queues.**ThreadedWrapperHandler**(*handler*)
This handled uses a single background thread to dispatch log records to a specific other handler using an internal queue. The idea is that if you are using a handler that requires some time to hand off the log records (such as the mail handler) and would block your request, you can let Logbook do that in a background thread.

The threaded wrapper handler will automatically adopt the methods and properties of the wrapped handler. All the values will be reflected:

```
>>> twh = ThreadedWrapperHandler(TestHandler())
>>> from logbook import WARNING
>>> twh.level_name = 'WARNING'
>>> twh.handler.level_name
'WARNING'
```

class logbook.queues.**SubscriberGroup**(*subscribers=None*, *queue_limit=10*)
This is a subscriber which represents a group of subscribers.

This is helpful if you are writing a server-like application which has "slaves". This way a user is easily able to view every log record which happened somewhere in the entire system without having to check every single slave:

```
subscribers = SubscriberGroup([
    MultiProcessingSubscriber(queue),
    ZeroMQSubscriber('tcp://localhost:5000')
])
with target_handler:
    subscribers.dispatch_forever()
```

add(*subscriber*)
Adds the given *subscriber* to the group.

stop()
Stops the group from internally recieving any more messages, once the internal queue is exhausted `recv()` will always return *None*.

Base Interface

class logbook.queues.**SubscriberBase**
Baseclass for all subscribers.

dispatch_forever()
Starts a loop that dispatches log records forever.

dispatch_in_background(*setup=None*)
Starts a new daemonized thread that dispatches in the background. An optional handler setup can be provided that pushed to the new thread (can be any *logbook.base.StackedObject*).

Returns a *ThreadController* object for shutting down the background thread. The background thread will already be running when this function returns.

dispatch_once(*timeout=None*)
Receives one record from the socket, loads it and dispatches it. Returns *True* if something was dispatched or *False* if it timed out.

recv(*timeout=None*)
Receives a single record from the socket. Timeout of 0 means nonblocking, *None* means blocking and otherwise it's a timeout in seconds after which the function just returns with *None*.

Subclasses have to override this.

class logbook.queues.**ThreadController**(*subscriber*, *setup=None*)
A helper class used by queue subscribers to control the background thread. This is usually created and started in one go by *dispatch_in_background()* or a comparable function.

start()
Starts the task thread.

stop()
Stops the task thread.

class logbook.queues.**TWHThreadController**(*wrapper_handler*)
A very basic thread controller that pulls things in from a queue and sends it to a handler. Both queue and handler are taken from the passed *ThreadedWrapperHandler*.

start()
Starts the task thread.

stop()
Stops the task thread.

1.10.5 Ticketing Support

This documents the support classes for ticketing. With ticketing handlers log records are categorized by location and for every emitted log record a count is added. That way you know how often certain messages are triggered, at what times and when the last occurrence was.

class logbook.ticketing.**TicketingBaseHandler**(*hash_salt*, *level=0*, *filter=None*, *bubble=False*)
Baseclass for ticketing handlers. This can be used to interface ticketing systems that do not necessarily provide an interface that would be compatible with the *BackendBase* interface.

hash_record_raw(*record*)
Returns the unique hash of a record.

class logbook.ticketing.**TicketingHandler**(*uri*, *app_id='generic'*, *level=0*, *filter=None*, *bubble=False*, *hash_salt=None*, *backend=None*, ***db_options*)

A handler that writes log records into a remote database. This database can be connected to from different dispatchers which makes this a nice setup for web applications:

```
from logbook.ticketing import TicketingHandler
handler = TicketingHandler('sqlite:////tmp/myapp-logs.db')
```

Parameters

- **uri** – a backend specific string or object to decide where to log to.
- **app_id** – a string with an optional ID for an application. Can be used to keep multiple application setups apart when logging into the same database.
- **hash_salt** – an optional salt (binary string) for the hashes.
- **backend** – A backend class that implements the proper database handling. Backends available are: *SQLAlchemyBackend*, *MongoDBBackend*.

default_backend

The default backend that is being used when no backend is specified. Unless overriden by a subclass this will be the *SQLAlchemyBackend*.

alias of *SQLAlchemyBackend*

emit(*record*)

Emits a single record and writes it to the database.

process_record(*record*, *hash*)

Subclasses can override this to tamper with the data dict that is sent to the database as JSON.

record_ticket(*record*, *data*, *hash*)

Record either a new ticket or a new occurrence for a ticket based on the hash.

class logbook.ticketing.**BackendBase**(***options*)

Provides an abstract interface to various databases.

count_tickets()

Returns the number of tickets.

delete_ticket(*ticket_id*)

Deletes a ticket from the database.

get_occurrences(*ticket*, *order_by='-time'*, *limit=50*, *offset=0*)

Selects occurrences from the database for a ticket.

get_ticket(*ticket_id*)

Return a single ticket with all occurrences.

get_tickets(*order_by='-last_occurrence_time'*, *limit=50*, *offset=0*)

Selects tickets from the database.

record_ticket(*record*, *data*, *hash*, *app_id*)

Records a log record as ticket.

setup_backend()

Setup the database backend.

solve_ticket(*ticket_id*)

Marks a ticket as solved.

class logbook.ticketing.**SQLAlchemyBackend**(***options*)
Implements a backend that is writing into a database SQLAlchemy can interface.

This backend takes some additional options:

table_prefix an optional table prefix for all tables created by the logbook ticketing handler.

metadata an optional SQLAlchemy metadata object for the table creation.

autocreate_tables can be set to *False* to disable the automatic creation of the logbook tables.

class logbook.ticketing.**MongoDBBackend**(***options*)
Implements a backend that writes into a MongoDB database.

1.10.6 The More Module

The more module implements special handlers and other things that are beyond the scope of Logbook itself or depend on external libraries. Additionally there are some handlers in *logbook.ticketing*, *logbook.queues* and *logbook.notifiers*.

Tagged Logging

class logbook.more.**TaggingLogger**(*name=None*, *tags=None*)
A logger that attaches a tag to each record. This is an alternative record dispatcher that does not use levels but tags to keep log records apart. It is constructed with a descriptive name and at least one tag. The tags are up for you to define:

```
logger = TaggingLogger('My Logger', ['info', 'warning'])
```

For each tag defined that way, a method appears on the logger with that name:

```
logger.info('This is a info message')
```

To dispatch to different handlers based on tags you can use the *TaggingHandler*.

The tags themselves are stored as list named `'tags'` in the `extra` dictionary.

call_handlers(*record*)
Pass a record to all relevant handlers in the following order:

- per-dispatcher handlers are handled first
- afterwards all the current context handlers in the order they were pushed

Before the first handler is invoked, the record is processed (*process_record()*).

handle(*record*)
Call the handlers for the specified record. This is invoked automatically when a record should be handled. The default implementation checks if the dispatcher is disabled and if the record level is greater than the level of the record dispatcher. In that case it will call the handlers (*call_handlers()*).

make_record_and_handle(*level*, *msg*, *args*, *kwargs*, *exc_info*, *extra*)
Creates a record from some given arguments and heads it over to the handling system.

process_record(*record*)
Processes the record with all context specific processors. This can be overriden to also inject additional information as necessary that can be provided by this record dispatcher.

class logbook.more.**TaggingHandler**(*handlers*, *filter=None*, *bubble=False*)
A handler that logs for tags and dispatches based on those.

Example:

```
import logbook
from logbook.more import TaggingHandler

handler = TaggingHandler(dict(
    info=OneHandler(),
    warning=AnotherHandler()
))
```

Special Handlers

class logbook.more.**TwitterHandler**(*consumer_key*, *consumer_secret*, *username*, *password*, *level=0*, *format_string=None*, *filter=None*, *bubble=False*)
A handler that logs to twitter. Requires that you sign up an application on twitter and request xauth support. Furthermore the oauth2 library has to be installed.

If you don't want to register your own application and request xauth credentials, there are a couple of leaked consumer key and secret pairs from application explicitly whitelisted at Twitter (leaked secrets).

formatter_class
alias of TwitterFormatter

get_oauth_token()
Returns the oauth access token.

make_client()
Creates a new oauth client auth a new access token.

tweet(*status*)
Tweets a given status. Status must not exceed 140 chars.

class logbook.more.**ExternalApplicationHandler**(*arguments*, *stdin_format=None*, *encoding='utf-8'*, *level=0*, *filter=None*, *bubble=False*)
This handler invokes an external application to send parts of the log record to. The constructor takes a list of arguments that are passed to another application where each of the arguments is a format string, and optionally a format string for data that is passed to stdin.

For example it can be used to invoke the say command on OS X:

```
from logbook.more import ExternalApplicationHandler
say_handler = ExternalApplicationHandler(['say', '{record.message}'])
```

Note that the above example is blocking until say finished, so it's recommended to combine this handler with the logbook.ThreadedWrapperHandler to move the execution into a background thread.

New in version 0.3.

class logbook.more.**ExceptionHandler**(*exc_type*, *level=0*, *format_string=None*, *filter=None*, *bubble=False*)
An exception handler which raises exceptions of the given *exc_type*. This is especially useful if you set a specific error *level* e.g. to treat warnings as exceptions:

```
from logbook.more import ExceptionHandler

class ApplicationWarning(Exception):
    pass
```

```
exc_handler = ExceptionHandler(ApplicationWarning, level='WARNING')
```

New in version 0.3.

Colorized Handlers

New in version 0.3.

class logbook.more.**ColorizedStderrHandler**(*level=0*, *format_string=None*, *filter=None*, *bubble=False*)

A colorizing stream handler that writes to stderr. It will only colorize if a terminal was detected. Note that this handler does not colorize on Windows systems.

New in version 0.3.

class logbook.more.**ColorizingStreamHandlerMixin**

A mixin class that does colorizing.

New in version 0.3.

get_color(*record*)

Returns the color for this record.

should_colorize(*record*)

Returns *True* if colorizing should be applied to this record. The default implementation returns *True* if the stream is a tty and we are not executing on windows.

Other

class logbook.more.**JinjaFormatter**(*template*)

A formatter object that makes it easy to format using a Jinja 2 template instead of a format string.

1.10.7 The Notifiers Module

The notifiers module implements special handlers for various platforms that depend on external libraries. The more module implements special handlers and other things that are beyond the scope of Logbook itself or depend on external libraries.

logbook.notifiers.**create_notification_handler**(*application_name=None*, *level=0*, *icon=None*)

Creates a handler perfectly fit the current platform. On Linux systems this creates a *LibNotifyHandler*, on OS X systems it will create a *GrowlHandler*.

OSX Specific Handlers

class logbook.notifiers.**GrowlHandler**(*application_name=None*, *icon=None*, *host=None*, *password=None*, *record_limit=None*, *record_delta=None*, *level=0*, *filter=None*, *bubble=False*)

A handler that dispatches to Growl. Requires that either growl-py or py-Growl are installed.

get_priority(*record*)

Returns the priority flag for Growl. Errors and criticals are get highest priority (2), warnings get higher priority (1) and the rest gets 0. Growl allows values between -2 and 2.

is_sticky(*record*)
Returns *True* if the sticky flag should be set for this record. The default implementation marks errors and criticals sticky.

Linux Specific Handlers

class logbook.notifiers.**LibNotifyHandler**(*application_name=None*, *icon=None*, *no_init=False*, *record_limit=None*, *record_delta=None*, *level=0*, *filter=None*, *bubble=False*)
A handler that dispatches to libnotify. Requires pynotify installed. If *no_init* is set to *True* the initialization of libnotify is skipped.

get_expires(*record*)
Returns either EXPIRES_DEFAULT or EXPIRES_NEVER for this record. The default implementation marks errors and criticals as EXPIRES_NEVER.

get_urgency(*record*)
Returns the urgency flag for pynotify. Errors and criticals are get highest urgency (CRITICAL), warnings get higher priority (NORMAL) and the rest gets LOW.

set_notifier_icon(*notifier*, *icon*)
Used to attach an icon on a notifier object.

Other Services

class logbook.notifiers.**BoxcarHandler**(*email*, *password*, *record_limit=None*, *record_delta=None*, *level=0*, *filter=None*, *bubble=False*)
Sends notifications to boxcar.io. Can be forwarded to your iPhone or other compatible device.

get_screen_name(*record*)
Returns the value of the screen name field.

class logbook.notifiers.**NotifoHandler**(*application_name=None*, *username=None*, *secret=None*, *record_limit=None*, *record_delta=None*, *level=0*, *filter=None*, *bubble=False*, *hide_level=False*)
Sends notifications to notifo.com. Can be forwarded to your Desktop, iPhone, or other compatible device.

Base Interface

class logbook.notifiers.**NotificationBaseHandler**(*application_name=None*, *record_limit=None*, *record_delta=None*, *level=0*, *filter=None*, *bubble=False*)
Baseclass for notification handlers.

make_text(*record*)
Called to get the text of the record.

make_title(*record*)
Called to get the title from the record.

1.10.8 Compatibility

This documents compatibility support with existing systems such as `logging` and `warnings`.

Logging Compatibility

logbook.compat.**redirect_logging**()
Permanently redirects logging to the stdlib. This also removes all otherwise registered handlers on root logger of the logging system but leaves the other loggers untouched.

logbook.compat.**redirected_logging**()
Temporarily redirects logging for all threads and reverts it later to the old handlers. Mainly used by the internal unittests:

```
from logbook.compat import redirected_logging
with redirected_logging():
    ...
```

class logbook.compat.**RedirectLoggingHandler**
A handler for the stdlib's logging system that redirects transparently to logbook. This is used by the *redirect_logging()* and *redirected_logging()* functions.

If you want to customize the redirecting you can subclass it.

convert_level(*level*)
Converts a logging level into a logbook level.

convert_record(*old_record*)
Converts an old logging record into a logbook log record.

convert_time(*timestamp*)
Converts the UNIX timestamp of the old record into a datetime object as used by logbook.

find_caller(*old_record*)
Tries to find the caller that issued the call.

find_extra(*old_record*)
Tries to find custom data from the old logging record. The return value is a dictionary that is merged with the log record extra dictionaries.

class logbook.compat.**LoggingHandler**(*logger=None*, *level=0*, *filter=None*, *bubble=False*)
Does the opposite of the *RedirectLoggingHandler*, it sends messages from logbook to logging. Because of that, it's a very bad idea to configure both.

This handler is for logbook and will pass stuff over to a logger from the standard library.

Example usage:

```
from logbook.compat import LoggingHandler, warn
with LoggingHandler():
    warn('This goes to logging')
```

convert_level(*level*)
Converts a logbook level into a logging level.

convert_record(*old_record*)
Converts a record from logbook to logging.

convert_time(*dt*)
Converts a datetime object into a timestamp.

get_logger(*record*)
Returns the logger to use for this record. This implementation always return `logger`.

Warnings Compatibility

logbook.compat.**redirect_warnings**()
Like *redirected_warnings()* but will redirect all warnings to the shutdown of the interpreter:

```
from logbook.compat import redirect_warnings
redirect_warnings()
```

logbook.compat.**redirected_warnings**()
A context manager that copies and restores the warnings filter upon exiting the context, and logs warnings using the logbook system.

The `channel` attribute of the log record will be the import name of the warning.

Example usage:

```
from logbook.compat import redirected_warnings
from warnings import warn

with redirected_warnings():
    warn(DeprecationWarning('logging should be deprecated'))
```

1.10.9 Internal API

This documents the internal API that might be useful for more advanced setups or custom handlers.

logbook.base.**dispatch_record**(*record*)
Passes a record on to the handlers on the stack. This is useful when log records are created programmatically and already have all the information attached and should be dispatched independent of a logger.

class logbook.base.**StackedObject**
Baseclass for all objects that provide stack manipulation operations.

applicationbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the application.

pop_application()
Pops the stacked object from the application stack.

pop_thread()
Pops the stacked object from the thread stack.

push_application()
Pushes the stacked object to the application stack.

push_thread()
Pushes the stacked object to the thread stack.

threadbound(*_cls=<class 'logbook._fallback._StackBound'>*)
Can be used in combination with the *with* statement to execute code while the object is bound to the thread.

class logbook.base.**RecordDispatcher**(*name=None*, *level=0*)
A record dispatcher is the internal base class that implements the logic used by the `Logger`.

call_handlers(*record*)
Pass a record to all relevant handlers in the following order:

- per-dispatcher handlers are handled first
- afterwards all the current context handlers in the order they were pushed

Before the first handler is invoked, the record is processed (*process_record()*).

group = None
optionally the name of the group this logger belongs to

handle(*record*)
Call the handlers for the specified record. This is invoked automatically when a record should be handled. The default implementation checks if the dispatcher is disabled and if the record level is greater than the level of the record dispatcher. In that case it will call the handlers (*call_handlers()*).

handlers = None
list of handlers specific for this record dispatcher

level
the level of the record dispatcher as integer

make_record_and_handle(*level*, *msg*, *args*, *kwargs*, *exc_info*, *extra*)
Creates a record from some given arguments and heads it over to the handling system.

name = None
the name of the record dispatcher

process_record(*record*)
Processes the record with all context specific processors. This can be overriden to also inject additional information as necessary that can be provided by this record dispatcher.

suppress_dispatcher = False
If this is set to *True* the dispatcher information will be suppressed for log records emitted from this logger.

class logbook.base.**LoggerMixin**
This mixin class defines and implements the "usual" logger interface (i.e. the descriptive logging functions).

Classes using this mixin have to implement a `handle()` method which takes a `LogRecord` and passes it along.

catch_exceptions(**args*, ***kwargs*)
A context manager that catches exceptions and calls *exception()* for exceptions caught that way. Example:

```
with logger.catch_exceptions():
    execute_code_that_might_fail()
```

critical(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `CRITICAL`.

debug(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `DEBUG`.

error(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `ERROR`.

exception(**args*, ***kwargs*)
Works exactly like *error()* just that the message is optional and exception information is recorded.

info(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `INFO`.

level_name
The name of the minimium logging level required for records to be created.

log(*level*, **args*, ***kwargs*)
Logs a `LogRecord` with the level set to the *level* parameter. Because custom levels are not supported by

logbook, this method is mainly used to avoid the use of reflection (e.g.: `getattr()`) for programmatic logging.

notice(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `NOTICE`.

warn(**args*, ***kwargs*)
Logs a `LogRecord` with the level set to `WARNING`. This function has an alias named *warning()*.

warning(**args*, ***kwargs*)
Alias for *warn()*.

class logbook.handlers.**RotatingFileHandlerBase**(**args*, ***kwargs*)
Baseclass for rotating file handlers.

Changed in version 0.3: This class was deprecated because the interface is not flexible enough to implement proper file rotations. The former builtin subclasses no longer use this baseclass.

perform_rollover()
Called if *should_rollover()* returns *True* and has to perform the actual rollover.

should_rollover(*record*, *formatted_record*)
Called with the log record and the return value of the `format_and_encode()` method. The method has then to return *True* if a rollover should happen or *False* otherwise.

Changed in version 0.3: Previously this method was called with the number of bytes returned by `format_and_encode()`

class logbook.handlers.**StringFormatterHandlerMixin**(*format_string*)
A mixin for handlers that provides a default integration for the `StringFormatter` class. This is used for all handlers by default that log text to a destination.

default_format_string = u'[{record.time:%Y-%m-%d %H:%M}] {record.level_name}: {record.channel}: {record
a class attribute for the default format string to use if the constructor was invoked with *None*.

format_string
the currently attached format string as new-style format string.

formatter_class
the class to be used for string formatting

alias of *StringFormatter*

1.11 The Design Explained

This part of the documentation explains the design of Logbook in detail. This is not strictly necessary to make use of Logbook but might be helpful when writing custom handlers for Logbook or when using it in a more complex environment.

1.11.1 Dispatchers and Channels

Logbook does not use traditional loggers, instead a logger is internally named as *RecordDispatcher*. While a logger also has methods to create new log records, the base class for all record dispatchers itself only has ways to dispatch `LogRecord`s to the handlers. A log record itself might have an attribute that points to the dispatcher that was responsible for dispatching, but it does not have to be.

If a log record was created from the builtin `Logger` it will have the channel set to the name of the logger. But that itself is no requirement. The only requirement for the channel is that it's a string with some human readable origin

information. It could be `'Database'` if the database issued the log record, it could be `'Process-4223'` if the process with the pid 4223 issued it etc.

For example if you are logging from the *logbook.log()* function they will have a cannel set, but no dispatcher:

```
>>> from logbook import TestHandler, warn
>>> handler = TestHandler()
>>> handler.push_application()
>>> warn('This is a warning')
>>> handler.records[0].channel
'Generic'
>>> handler.records[0].dispatcher is None
True
```

If you are logging from a custom logger, the channel attribute points to the logger for as long this logger class is not garbage collected:

```
>>> from logbook import Logger, TestHandler
>>> logger = Logger('Console')
>>> handler = TestHandler()
>>> handler.push_application()
>>> logger.warn('A warning')
>>> handler.records[0].dispatcher is logger
True
```

You don't need a record dispatcher to dispatch a log record though. The default dispatching can be triggered from a function *dispatch_record()*:

```
>>> from logbook import dispatch_record, LogRecord, INFO
>>> record = LogRecord('My channel', INFO, 'Hello World!')
>>> dispatch_record(record)
[2010-09-04 15:56] INFO: My channel: Hello World!
```

It is pretty common for log records to be created without a dispatcher. Here some common use cases for log records without a dispatcher:

- log records that were redirected from a different logging system such as the standard library's `logging` module or the `warnings` module.
- log records that came from different processes and do not have a dispatcher equivalent in the current process.
- log records that came from over the network.

1.11.2 The Log Record Container

The `LogRecord` class is a simple container that holds all the information necessary for a log record. Usually they are created from a `Logger` or one of the default log functions (*logbook.warn()* etc.) and immediately dispatched to the handlers. The logger will apply some additional knowledge to figure out where the record was created from and if a traceback information should be attached.

Normally if log records are dispatched they will be closed immediately after all handlers had their chance to write it down. On closing, the interpreter frame and traceback object will be removed from the log record to break up circular dependencies.

Sometimes however it might be necessary to keep log records around for a longer time. Logbook provides three different ways to accomplish that:

1. Handlers can set the `keep_open` attribute of a log record to *True* so that the record dispatcher will not close the object. This is for example used by the `TestHandler` so that unittests can still access interpreter frames and traceback objects if necessary.

2. Because some information on the log records depends on the interpreter frame (such as the location of the log call) it is possible to pull that related information directly into the log record so that it can safely be closed without losing that information (see `pull_information()`).
3. Last but not least, log records can be converted to dictionaries and recreated from these. It is also possible to make these dictionaries safe for JSON export which is used by the *`TicketingHandler`* to store information in a database or the `MultiProcessingHandler` to send information between processes.

1.12 Design Principles

Logbook is a logging library that breaks many expectations people have in logging libraries to support paradigms we think are more suitable for modern applications than the traditional Java inspired logging system that can also be found in the Python standard library and many more programming languages.

This section of the documentation should help you understand the design of Logbook and why it was implemented like this.

1.12.1 No Logger Registry

Logbook is unique in that it has the concept of logging channels but that it does not keep a global registry of them. In the standard library's logging module a logger is attached to a tree of loggers that are stored in the logging module itself as global state.

In logbook a logger is just an opaque object that might or might not have a name and attached information such as log level or customizations, but the lifetime and availability of that object is controlled by the person creating that logger.

The registry is necessary for the logging library to give the user the ability to configure these loggers.

Logbook has a completely different concept of dispatching from loggers to the actual handlers which removes the requirement and usefulness of such a registry. The advantage of the logbook system is that it's a cheap operation to create a logger and that a logger can easily be garbage collected to remove all traces of it.

Instead Logbook moves the burden of delivering a log record from the log channel's attached log to an independent entity that looks at the context of the execution to figure out where to deliver it.

1.12.2 Context Sensitive Handler Stack

Python has two builtin ways to express implicit context: processes and threads. What this means is that if you have a function that is passed no arguments at all, you can figure out what thread called the function and what process you are sitting in. Logbook supports this context information and lets you bind a handler (or more!) for such a context.

This is how this works: there are two stacks available at all times in Logbook. The first stack is the process wide stack. It is manipulated with `Handler.push_application` and `Handler.pop_application` (and of course the context manager `Handler.applicationbound`). Then there is a second stack which is per thread. The manipulation of that stack happens with `Handler.push_thread`, `Handler.pop_thread` and the `Handler.threadbound` contextmanager.

Let's take a WSGI web application as first example. When a request comes in your WSGI server will most likely do one of the following two things: either spawn a new Python process (or reuse a process in a pool), or create a thread (or again, reuse something that already exists). Either way, we can now say that the context of process id and thread id is our playground. For this context we can define a log handler that is active in this context only for a certain time. In pseudocode this would look like this:

```
def my_application(environ, start_response):
    my_handler = FileHandler(...)
    my_handler.push_thread()
    try:
        # whatever happens here in terms of logging is handled
        # by the `my_handler` handler.
        ...
    finally:
        my_handler.pop_thread()
```

Because this is a lot to type, you can also use the *with* statement to do the very same:

```
def my_application(environ, start_response):
    with FileHandler(...).threadbound() as my_handler:
        # whatever happens here in terms of logging is handled
        # by the `my_handler` handler.
        ...
```

Additionally there is another place where you can put handlers: directly onto a logging channel (for example on a `Logger`).

This stack system might seem like overkill for a traditional system, but it allows complete decoupling from the log handling system and other systems that might log messages.

Let's take a GUI application rather than a web application. You have an application that starts up, shuts down and at any point in between might fail or log messages. The typical default behaviour here would be to log into a logfile. Fair enough, that's how these applications work.

But what's the point in logging if not even a single warning happened? The traditional solution with the logging library from Python is to set the level high (like *ERROR* or *WARNING*) and log into a file. When things break, you have a look at the file and hope it contains enough information.

When you are in full control of the context of execution with a stack based system like Logbook has, there is a lot more you can do.

For example you could immediately after your application boots up instanciate a `FingersCrossedHandler`. This handler buffers *all* log records in memory and does not emit them at all. What's the point? That handler activates when a certain threshold is reached. For example, when the first warning occurs you can write the buffered messages as well as the warning that just happened into a logfile and continue logging from that point. Because there is no point in logging when you will never look at that file anyways.

But that alone is not the killer feature of a stack. In a GUI application there is the point where we are still initializing the windowing system. So a file is the best place to log messages. But once we have the GUI initialized, it would be very helpful to show error messages to a user in a console window or a dialog. So what we can do is to initialize at that point a new handler that logs into a dialog.

When then a long running tasks in the GUI starts we can move that into a separate thread and intercept all the log calls for that thread into a separate window until the task succeeded.

Here such a setup in pseudocode:

```
from logbook import FileHandler, WARNING
from logbook import FingersCrossedHandler

def main():
    # first we set up a handler that logs everything (including debug
    # messages, but only starts doing that when a warning happens
    default_handler = FingersCrossedHandler(FileHandler(filename,
                                                        delay=True),
                                            WARNING)
```

```
    # this handler is now activated as the default handler for the
    # whole process.  We do not bubble up to the default handler
    # that logs to stderr.
    with default_handler.applicationbound(bubble=False):
        # now we initialize the GUI of the application
        initialize_gui()
        # at that point we can hook our own logger in that intercepts
        # errors and displays them in a log window
        with gui.log_handler.applicationbound():
            # run the gui mainloop
            gui.mainloop()
```

This stack can also be used to inject additional information automatically into log records. This is also used to replace the need for custom log levels.

1.12.3 No Custom Log Levels

This change over logging was controversial, even under the two original core developers. There clearly are use cases for custom log levels, but there is an inherent problem with then: they require a registry. If you want custom log levels, you will have to register them somewhere or parts of the system will not know about them. Now we just spent a lot of time ripping out the registry with a stack based approach to solve delivery problems, why introduce a global state again just for log levels?

Instead we looked at the cases where custom log levels are useful and figured that in most situations custom log levels are used to put additional information into a log entry. For example it's not uncommon to have separate log levels to filter user input out of a logfile.

We instead provide powerful tools to inject arbitrary additional data into log records with the concept of log processors.

So for example if you want to log user input and tag it appropriately you can override the `Logger.process_record()` method:

```
class InputLogger(Logger):
    def process_record(self, record):
        record.extra['kind'] = 'input'
```

A handler can then use this information to filter out input:

```
def no_input(record, handler):
    return record.extra.get('kind') != 'input'

with MyHandler().threadbound(filter=no_input):
    ...
```

1.12.4 Injecting Context-Sensitive Information

For many situations it's not only necessary to inject information on a per-channel basis but also for all logging calls from a given context. This is best explained for web applications again. If you have some libraries doing logging in code that is triggered from a request you might want to record the URL of that request for each log record so that you get an idea where a specific error happened.

This can easily be accomplished by registering a custom processor when binding a handler to a thread:

```
def my_application(environ, start_reponse):
    def inject_request_info(record, handler):
        record.extra['path'] = environ['PATH_INFO']
    with Processor(inject_request_info).threadbound():
```

```
        with my_handler.threadbound():
            # rest of the request code here
            ...
```

1.12.5 Logging Compatibility

The last pillar of logbook's design is the compatibility with the standard libraries logging system. There are many libraries that exist currently that log information with the standard libraries logging module. Having two separate logging systems in the same process is countrproductive and will cause separate logfiles to appear in the best case or complete chaos in the worst.

Because of that, logbook provides ways to transparently redirect all logging records into the logbook stack based record delivery system. That way you can even continue to use the standard libraries logging system to emit log messages and can take the full advantage of logbook's powerful stack system.

If you are curious, have a look at *Logging Compatibility*.

1.13 Logbook Changelog

Here you can see the full list of changes between each Logbook release.

1.13.1 Version 0.4

Release date to be announced. Codename to be selected.

- Added *logbook.notifiers.NotifoHandler*
- *channel* is now documented to be used for filtering purposes if wanted. Previously this was an opaque string that was not intended for filtering of any kind.

1.13.2 Version 0.3

Released on October 23rd. Codename "Informant"

- Added *logbook.more.ColorizingStreamHandlerMixin* and *logbook.more.ColorizedStderrHandler*
- Deprecated `logbook.RotatingFileHandlerBase` because the interface was not flexible enough.
- Provided basic Python 3 compatibility. This did cause a few smaller API changes that caused minimal changes on Python 2 as well. The deprecation of the `logbook.RotatingFileHandlerBase` was a result of this.
- Added support for Python 2.4
- Added batch emitting support for handlers which now makes it possible to use the `logbook.more.FingersCrossedHandler` with the `logbook.MailHandler`.
- Moved the `FingersCrossedHandler` handler into the base package. The old location stays importable for a few releases.
- Added `logbook.GroupHandler` that buffers records until the handler is popped.
- Added *logbook.more.ExternalApplicationHandler* that executes an external application for each log record emitted.

1.13.3 Version 0.2.1

Bugfix release, Released on September 22nd.

- Fixes Python 2.5 compatibility.

1.13.4 Version 0.2

Released on September 21st. Codename "Walls of Text"

- Implemented default with statement for handlers which is an alias for *threadbound*.
- *applicationbound* and *threadbound* return the handler now.
- Implemented channel recording on the log records.
- The `logbook.more.FingersCrossedHandler` now is set to *ERROR* by default and has the ability to create new loggers from a factory function.
- Implemented maximum buffer size for the `logbook.more.FingersCrossedHandler` as well as a lock for thread safety.
- Added ability to filter for context.
- Moved bubbling flags and filters to the handler object.
- Moved context processors on their own stack.
- Removed the *iter_context_handlers* function.
- Renamed *NestedHandlerSetup* to `NestedSetup` because it can now also configure processors.
- Added the `logbook.Processor` class.
- There is no difference between logger attached handlers and context specific handlers any more.
- Added a function to redirect warnings to logbook (*logbook.compat.redirected_warnings()*).
- Fixed and improved `logbook.LoggerGroup`.
- The `logbook.TestHandler` now keeps the record open for further inspection.
- The traceback is now removed from a log record when the record is closed. The formatted traceback is a cached property instead of a function.
- Added ticketing handlers that send logs directly into a database.
- Added MongoDB backend for ticketing handlers
- Added a *logbook.base.dispatch_record()* function to dispatch records to handlers independently of a logger (uses the default record dispatching logic).
- Renamed *logger_name* to *channel*.
- Added a multi processing log handler (`logbook.more.MultiProcessingHandler`).
- Added a twitter handler.
- Added a ZeroMQ handler.
- Added a Growl handler.
- Added a Libnotify handler.
- Added a monitoring file handler.
- Added a handler wrapper that moves the actual handling into a background thread.

- The mail handler can now be configured to deliver each log record not more than n times in m seconds.
- Added support for Python 2.5
- Added a *logbook.queues.SubscriberGroup* to deal with multiple subscribers.
- Added a *logbook.compat.LoggingHandler* for redirecting logbook log calls to the standard library's `logging` module.

1.13.5 Version 0.1

First public release.

CHAPTER 2

Project Information

- Download from PyPI
- Master repository on GitHub
- Mailing list
- IRC: `#pocoo` on freenode

l

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

U

W

Z