Logging Cookbook

Release 3.2.2

Guido van Rossum
Fred L. Drake, Jr., editor

Contents

Author Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains a number of recipes related to logging, which have been found useful in the past.

1 Using logging in multiple modules

Multiple calls to `logging.getLogger('someLogger')` return a reference to the same logger object. This is true not only within the same module, but also across modules as long as it is in the same Python interpreter process. It is true for references to the same object; additionally, application code can define and configure a parent

logger in one module and create (but not configure) a child logger in a separate module, and all logger calls to the child will pass up to the parent. Here is a main module:

```python
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

Here is the auxiliary module:

```python
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary')


class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')
    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')


def some_function():
    module_logger.info('received a call to "some_function"')
```

The output looks like this:

```
2005-03-23 23:47:11,663 - spam_application - INFO -
   creating an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary - INFO -
   creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
   created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
```

```
    calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary - INFO -
    doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary - INFO -
    done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
    finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
    calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
    received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
    done with auxiliary_module.some_function()
```

2 Multiple handlers and formatters

Loggers are plain Python objects. The `addHandler()` method has no minimum or maximum quota for the number of handlers you may add. Sometimes it will be beneficial for an application to log all messages of all severities to a text file while simultaneously logging errors or above to the console. To set this up, simply configure the appropriate handlers. The logging calls in the application code will remain unchanged. Here is a slight modification to the previous simple module-based configuration example:

```python
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Notice that the 'application' code does not care about multiple handlers. All that changed was the addition and configuration of a new handler named *fh*.

The ability to create new handlers with higher- or lower-severity filters can be very helpful when writing and testing an application. Instead of using many `print` statements for debugging, use `logger.debug`: Unlike the print statements, which you will have to delete or comment out later, the logger.debug statements can remain intact in the source code and remain dormant until you need them again. At that time, the only change that needs to happen is to modify the severity level of the logger and/or handler to debug.

3 Logging to multiple destinations

Let's say you want to log to console and file with different message formats and in differing circumstances. Say you want to log messages with levels of DEBUG and higher to file, and those messages at level INFO and higher to the console. Let's also assume that the file should contain timestamps, but the console messages should not. Here's how you can achieve this:

```python
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/temp/myapp.log',
                    filemode='w')
# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

When you run this, on the console you will see

```
root         : INFO      Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO      How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

and in the file you will see something like

```
10-22 22:19 root          INFO      Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1  DEBUG    Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1  INFO      How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2  WARNING  Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2  ERROR    The five boxing wizards jump quickly.
```

As you can see, the DEBUG message only shows up in the file. The other messages are sent to both destinations.

This example uses console and file handlers, but you can use any number and combination of handlers you choose.

4 Configuration server example

Here is an example of a module using the logging configuration server:

```python
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warn('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # cleanup
    logging.config.stopListening()
    t.join()
```

And here is a script that takes a filename and sends that file to the server, properly preceded with the binary-encoded length, as the new logging configuration:

```python
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```

5 Dealing with handlers that block

Sometimes you have to get your logging handlers to do their work without blocking the thread you're logging from. This is common in Web applications, though of course it also occurs in other scenarios.

A common culprit which demonstrates sluggish behaviour is the `SMTPHandler`: sending emails can take a long time, for a number of reasons outside the developer's control (for example, a poorly performing mail or network infrastructure). But almost any network-based handler can block: Even a `SocketHandler` operation may do a DNS query under the hood which is too slow (and this query can be deep in the socket library code, below the Python layer, and outside your control).

One solution is to use a two-part approach. For the first part, attach only a `QueueHandler` to those loggers which are accessed from performance-critical threads. They simply write to their queue, which can be sized to a large enough capacity or initialized with no upper bound to their size. The write to the queue will typically be accepted quickly, though you will probably need to catch the `queue.Full` exception as a precaution in your code. If you are a library developer who has performance-critical threads in their code, be sure to document this (together with a suggestion to attach only `QueueHandlers` to your loggers) for the benefit of other developers who will use your code.

The second part of the solution is `QueueListener`, which has been designed as the counterpart to `QueueHandler`. A `QueueListener` is very simple: it's passed a queue and some handlers, and it fires up an internal thread which listens to its queue for LogRecords sent from `QueueHandlers` (or any other source of `LogRecords`, for that matter). The `LogRecords` are removed from the queue and passed to the handlers for processing.

The advantage of having a separate `QueueListener` class is that you can use the same instance to service multiple `QueueHandlers`. This is more resource-friendly than, say, having threaded versions of the existing handler classes, which would eat up one thread per handler for no particular benefit.

An example of using these two classes follows (imports omitted):

```
que = queue.Queue(-1) # no limit on size
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
root.addHandler(queue_handler)
formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# The log output will display the thread which generated
# the event (the main thread) rather than the internal
# thread which monitors the internal queue. This is what
# you want to happen.
root.warning('Look out!')
listener.stop()
```

which, when run, will produce:

```
MainThread: Look out!
```

6 Sending and receiving logging events across a network

Let's say you want to send logging events across a network, and handle them at the receiving end. A simple way of doing this is attaching a `SocketHandler` instance to the root logger at the sending end:

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)
```

```python
# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

At the receiving end, you can set up a receiver using the socketserver module. Here is a basic working example:

```python
import pickle
import logging
import logging.handlers
import socketserver
import struct


class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte length,
        followed by the LogRecord in pickle format. Logs the record
        according to whatever policy is configured locally.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
```

```python
        name = record.name
    logger = logging.getLogger(name)
    # N.B. EVERY record gets logged. This is because Logger.handle
    # is normally called AFTER logger-level filtering. If you want
    # to do filtering, do it at the client end to save wasting
    # cycles and network bandwidth!
    logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    """
    Simple TCP socket-based logging receiver suitable for testing.
    """

    allow_reuse_address = 1

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        socketserver.ThreadingTCPServer.__init__(self, (host, port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                       [], [],
                                       self.timeout)
            if rd:
                self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

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

First run the server, and then the client. On the client side, nothing is printed on the console; on the server side, you should see something like:

```
About to start TCP server...
   59 root            INFO     Jackdaws love my big sphinx of quartz.
   59 myapp.area1     DEBUG    Quick zephyrs blow, vexing daft Jim.
   69 myapp.area1     INFO     How quickly daft jumping zebras vex.
   69 myapp.area2     WARNING  Jail zesty vixen who grabbed pay from quack.
   69 myapp.area2     ERROR    The five boxing wizards jump quickly.
```

Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the makePickle() method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.

7 Adding contextual information to your logging output

Sometimes you want logging output to contain contextual information in addition to the parameters passed to the logging call. For example, in a networked application, it may be desirable to log client-specific information in the log (e.g. remote client's username, or IP address). Although you could use the *extra* parameter to achieve this, it's not always convenient to pass the information in this way. While it might be tempting to create `Logger` instances on a per-connection basis, this is not a good idea because these instances are not garbage collected. While this is not a problem in practice, when the number of `Logger` instances is dependent on the level of granularity you want to use in logging an application, it could be hard to manage if the number of `Logger` instances becomes effectively unbounded.

7.1 Using LoggerAdapters to impart contextual information

An easy way in which you can pass contextual information to be output along with logging event information is to use the `LoggerAdapter` class. This class is designed to look like a `Logger`, so that you can call `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()` and `log()`. These methods have the same signatures as their counterparts in `Logger`, so you can use the two types of instances interchangeably.

When you create an instance of `LoggerAdapter`, you pass it a `Logger` instance and a dict-like object which contains your contextual information. When you call one of the logging methods on an instance of `LoggerAdapter`, it delegates the call to the underlying instance of `Logger` passed to its constructor, and arranges to pass the contextual information in the delegated call. Here's a snippet from the code of `LoggerAdapter`:

```python
def debug(self, msg, *args, **kwargs):
    """
    Delegate a debug call to the underlying logger, after adding
    contextual information from this adapter instance.
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

The `process()` method of `LoggerAdapter` is where the contextual information is added to the logging output. It's passed the message and keyword arguments of the logging call, and it passes back (potentially) modified versions of these to use in the call to the underlying logger. The default implementation of this method leaves the message alone, but inserts an 'extra' key in the keyword argument whose value is the dict-like object passed to the constructor. Of course, if you had passed an 'extra' keyword argument in the call to the adapter, it will be silently overwritten.

The advantage of using 'extra' is that the values in the dict-like object are merged into the `LogRecord` instance's `__dict__`, allowing you to use customized strings with your `Formatter` instances which know about the keys of the dict-like object. If you need a different method, e.g. if you want to prepend or append the contextual information to the message string, you just need to subclass `LoggerAdapter` and override `process()` to do what you need. Here's an example script which uses this class, which also illustrates what dict-like behaviour is needed from an arbitrary 'dict-like' object for use in the constructor:

```python
import logging

class ConnInfo:
    """
    An example class which shows how an arbitrary class can be used as
    the 'extra' context information repository passed to a LoggerAdapter.
    """

    def __getitem__(self, name):
        """
        To allow this instance to look like a dict.
        """
        from random import choice
```

```
        if name == 'ip':
            result = choice(['127.0.0.1', '192.168.0.1'])
        elif name == 'user':
            result = choice(['jim', 'fred', 'sheila'])
        else:
            result = self.__dict__.get(name, '?')
        return result

    def __iter__(self):
        """
        To allow iteration over keys, which will be merged into
        the LogRecord dict before formatting and output.
        """
        keys = ['ip', 'user']
        keys.extend(self.__dict__.keys())
        return keys.__iter__()

if __name__ == '__main__':
    from random import choice
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITI
    a1 = logging.LoggerAdapter(logging.getLogger('a.b.c'),
                                { 'ip' : '123.231.231.123', 'user' : 'sheila' })
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s IP: %(ip)-15s
    a1.debug('A debug message')
    a1.info('An info message with %s', 'some parameters')
    a2 = logging.LoggerAdapter(logging.getLogger('d.e.f'), ConnInfo())
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s', lvlname, 2, 'parameters')
```

When this script is run, the output should look something like this:

```
2008-01-18 14:49:54,023 a.b.c DEBUG    IP: 123.231.231.123 User: sheila   A debug messag
2008-01-18 14:49:54,023 a.b.c INFO     IP: 123.231.231.123 User: sheila   An info messag
2008-01-18 14:49:54,023 d.e.f CRITICAL IP: 192.168.0.1     User: jim      A message at C
2008-01-18 14:49:54,033 d.e.f INFO     IP: 192.168.0.1     User: jim      A message at I
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: sheila   A message at W
2008-01-18 14:49:54,033 d.e.f ERROR    IP: 127.0.0.1       User: fred     A message at E
2008-01-18 14:49:54,033 d.e.f ERROR    IP: 127.0.0.1       User: sheila   A message at E
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: sheila   A message at W
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: jim      A message at W
2008-01-18 14:49:54,033 d.e.f INFO     IP: 192.168.0.1     User: fred     A message at I
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: sheila   A message at W
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 127.0.0.1       User: jim      A message at W
```

7.2 Using Filters to impart contextual information

You can also add contextual information to log output using a user-defined Filter. Filter instances are allowed to modify the LogRecords passed to them, including adding additional attributes which can then be output using a suitable format string, or if needed a custom Formatter.

For example in a web application, the request being processed (or at least, the interesting parts of it) can be stored in a threadlocal (threading.local) variable, and then accessed from a Filter to add, say, information from the request - say, the remote IP address and remote user's username - to the LogRecord, using the attribute names 'ip' and 'user' as in the LoggerAdapter example above. In that case, the same format string can be used to get similar output to that shown above. Here's an example script:

```python
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into the log.

    Rather than use actual contextual information, we just use random
    data in this demo.
    """

    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

    def filter(self, record):

        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
        return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITIC
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s IP: %(ip)-15s U
    a1 = logging.getLogger('a.b.c')
    a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
    a1.addFilter(f)
    a2.addFilter(f)
    a1.debug('A debug message')
    a1.info('An info message with %s', 'some parameters')
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s', lvlname, 2, 'parameters')
```

which, when run, produces something like:

```
2010-09-06 22:38:15,292 a.b.c DEBUG    IP: 123.231.231.123 User: fred    A debug messag
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1     User: sheila  An info messag
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila  A message at C
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1       User: jim     A message at E
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1       User: sheila  A message at D
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User: fred    A message at E
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1     User: jim     A message at C
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila  A message at C
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1     User: jim     A message at D
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1       User: sheila  A message at E
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User: fred    A message at D
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123 User: fred    A message at I
```

8 Logging to a single file from multiple processes

Although logging is thread-safe, and logging to a single file from multiple threads in a single process *is* supported, logging to a single file from *multiple processes* is *not* supported, because there is no standard way to serialize access to a single file across multiple processes in Python. If you need to log to a single file from multiple

processes, one way of doing this is to have all the processes log to a `SocketHandler`, and have a separate process which implements a socket server which reads from the socket and logs to file. (If you prefer, you can dedicate one thread in one of the existing processes to perform this function.) The following section documents this approach in more detail and includes a working socket receiver which can be used as a starting point for you to adapt in your own applications.

If you are using a recent version of Python which includes the `multiprocessing` module, you could write your own handler which uses the `Lock` class from this module to serialize access to the file from your processes. The existing `FileHandler` and subclasses do not make use of `multiprocessing` at present, though they may do so in the future. Note that at present, the `multiprocessing` module does not provide working lock functionality on all platforms (see http://bugs.python.org/issue3770).

Alternatively, you can use a `Queue` and a `QueueHandler` to send all logging events to one of the processes in your multi-process application. The following example script demonstrates how you can do this; in the example a separate listener process listens for events sent by other processes and logs them according to its own logging configuration. Although the example only demonstrates one way of doing it (for example, you may want to use a listener thread rather than a separate listener process – the implementation would be analogous) it does allow for completely different logging configurations for the listener and the other processes in your application, and can be used as the basis for code meeting your own specific requirements:

```python
# You'll need these imports in your own code
import logging
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

#
# Because you'll want to define the logging configurations for listener and workers, the
# listener and worker process functions take a configurer parameter which is a callable
# for configuring logging for that process. These functions are also passed the queue,
# which they use for communication.
#
# In practice, you can configure the listener however you want, but note that in this
# simple example, the listener does not apply level or filter logic to received records.
# In practice, you would probably want to do this logic in the worker processes, to avoi
# sending events which would be filtered out between processes.
#
# The size of the rotated files is made small so you can see the results easily.
def listener_configurer():
    root = logging.getLogger()
    h = logging.handlers.RotatingFileHandler('mptest.log', 'a', 300, 10)
    f = logging.Formatter('%(asctime)s %(processName)-10s %(name)s %(levelname)-8s %(mes
    h.setFormatter(f)
    root.addHandler(h)

# This is the listener process top-level loop: wait for logging events
# (LogRecords)on the queue and handle them, quit when you get a None for a
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None: # We send this as a sentinel to tell the listener to quit
                break
            logger = logging.getLogger(record.name)
            logger.handle(record) # No level or filter logic applied - just do it!
```

```python
        except (KeyboardInterrupt, SystemExit):
            raise
        except:
            import sys, traceback
            print >> sys.stderr, 'Whoops! Problem:'
            traceback.print_exc(file=sys.stderr)

# Arrays used for random selections in this demo

LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING,
          logging.ERROR, logging.CRITICAL]

LOGGERS = ['a.b.c', 'd.e.f']

MESSAGES = [
    'Random message #1',
    'Random message #2',
    'Random message #3',
]

# The worker configuration is done at the start of the worker process run.
# Note that on Windows you can't rely on fork semantics, so each process
# will run the logging configuration code when it starts.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue) # Just the one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    root.setLevel(logging.DEBUG) # send all messages, for demo; no other level or filter

# This is the worker process top-level loop, which just logs ten events with
# random intervening delays before terminating.
# The print messages are just so you know it's doing something!
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print('Worker finished: %s' % name)

# Here's where the demo gets orchestrated. Create the queue, create and start
# the listener, create ten workers and start them, wait for them to finish,
# then send a None to the queue to tell the listener to finish.
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process,
                                       args=(queue, listener_configurer))
    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process,
                                         args=(queue, worker_configurer))
        workers.append(worker)
        worker.start()
```

```python
        for w in workers:
            w.join()
        queue.put_nowait(None)
        listener.join()

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

A variant of the above script keeps the logging in the main process, in a separate thread:

```python
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue
import random
import threading
import time

def logger_thread(q):
    while True:
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)


def worker_process(q):
    qh = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(qh)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-10s %(
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
```

```python
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
                'handlers' : ['foofile']
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1), args=(q,))
        workers.append(wp)
        wp.start()
    logging.config.dictConfig(d)
    lp = threading.Thread(target=logger_thread, args=(q,))
    lp.start()
    # At this point, the main process could do some useful work of its own
    # Once it's done that, it can wait for the workers to terminate...
    for wp in workers:
        wp.join()
    # And now tell the logging thread to finish up, too
    q.put(None)
    lp.join()
```

This variant shows how you can e.g. apply configuration for particular loggers - e.g. the `foo` logger has a special handler which stores all events in the `foo` subsystem in a file `mplog-foo.log`. This will be used by the logging machinery in the main process (even though the logging events are generated in the worker processes) to direct the messages to the appropriate destinations.

9 Using file rotation

Sometimes you want to let a log file grow to a certain size, then open a new file and log to that. You may want to keep a certain number of these files, and when that many files have been created, rotate the files so that the number of files and the size of the files both remain bounded. For this usage pattern, the logging package provides a `RotatingFileHandler`:

```python
import glob
import logging
```

```python
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
              LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```

The result should be 6 separate files, each with part of the log history for the application:

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

The most current file is always `logging_rotatingfile_example.out`, and each time it reaches the size limit it is renamed with the suffix `.1`. Each of the existing backup files is renamed to increment the suffix (`.1` becomes `.2`, etc.) and the `.6` file is erased.

Obviously this example sets the log length much much too small as an extreme example. You would want to set *maxBytes* to an appropriate value.

10 Subclassing QueueHandler - a ZeroMQ example

You can use a `QueueHandler` subclass to send messages to other kinds of queues, for example a ZeroMQ 'publish' socket. In the example below,the socket is created separately and passed to the handler (as its 'queue'):

```python
import zmq # using pyzmq, the Python binding for ZeroMQ
import json # for serializing records portably

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB) # or zmq.PUSH, or other suitable value
sock.bind('tcp://*:5556') # or wherever

class ZeroMQSocketHandler(QueueHandler):
    def enqueue(self, record):
        data = json.dumps(record.__dict__)
        self.queue.send(data)

handler = ZeroMQSocketHandler(sock)
```

Of course there are other ways of organizing this, for example passing in the data needed by the handler to create the socket:

```python
class ZeroMQSocketHandler(QueueHandler):
    def __init__(self, uri, socktype=zmq.PUB, ctx=None):
        self.ctx = ctx or zmq.Context()
        socket = zmq.Socket(self.ctx, socktype)
        socket.bind(uri)
        QueueHandler.__init__(self, socket)

    def enqueue(self, record):
        data = json.dumps(record.__dict__)
        self.queue.send(data)

    def close(self):
        self.queue.close()
```

11 Subclassing QueueListener - a ZeroMQ example

You can also subclass `QueueListener` to get messages from other kinds of queues, for example a ZeroMQ 'subscribe' socket. Here's an example:

```python
class ZeroMQSocketListener(QueueListener):
    def __init__(self, uri, *handlers, **kwargs):
        self.ctx = kwargs.get('ctx') or zmq.Context()
        socket = zmq.Socket(self.ctx, zmq.SUB)
        socket.setsockopt(zmq.SUBSCRIBE, '') # subscribe to everything
        socket.connect(uri)

    def dequeue(self):
        msg = self.queue.recv()
        return logging.makeLogRecord(json.loads(msg))
```

See Also:

Module `logging` API reference for the logging module.

Module `logging.config` Configuration API for the logging module.

Module `logging.handlers` Useful handlers included with the logging module.

A basic logging tutorial

A more advanced logging tutorial