# 日志操作手册

**发布 *3.6.8***

**Guido van Rossum**
**and the Python development team**

**五月 10, 2019**

**Python Software Foundation**
**Email: docs@python.org**

## Contents

1 在多个模块中使用日志 2

2 在多线程中使用日志 4

3 使用多个日志处理器和多种格式化 5

4 在多个地方记录日志 6

5 日志服务器配置示例 7

6 处理日志处理器的阻塞 8

7 Sending and receiving logging events across a network 9

8 Adding contextual information to your logging output 11
- 8.1 Using LoggerAdapters to impart contextual information . . . . . 11
- 8.2 Using Filters to impart contextual information . . . . . 12

9 Logging to a single file from multiple processes 14

10 Using file rotation 18

11 Use of alternative formatting styles 19

12 Customizing `LogRecord` 22

13 Subclassing QueueHandler - a ZeroMQ example 23

14 Subclassing QueueListener - a ZeroMQ example 24

15 An example dictionary-based configuration 24

16 Using a rotator and namer to customize log rotation processing 26

17 A more elaborate multiprocessing example 26

18 Inserting a BOM into messages sent to a SysLogHandler 30

19 Implementing structured logging 31

20 Customizing handlers with dictConfig() 32

21 Using particular formatting styles throughout your application 35
21.1 Using LogRecord factories . . . . . 35
21.2 Using custom message objects . . . . . 35

22 Configuring filters with dictConfig() 37

23 Customized exception formatting 38

24 Speaking logging messages 39

25 Buffering logging messages and outputting them conditionally 40

26 Formatting times using UTC (GMT) via configuration 42

27 Using a context manager for selective logging 43

索引 46

---

**作者** Vinay Sajip <vinay_sajip at red-dove dot com>

本页包含了许多日志记录相关的概念，这些概念在过去一直被认为很有用。

# 1 在多个模块中使用日志

多次调用 ‘‘logging.getLogger( ‘someLogger’ )‘‘时会返回对同一个 logger 对象的引用。这不仅是在同一个模块中，在其他模块调用也是如此，只要是在同一个 Python 解释器进程中。是应该引用同一个对象，此外，应用程序也可以在一个模块中定义和配置父 logger，而在单独的模块中创建（但不配置）子 logger，对子 logger 的所有调用都将传给父 logger。这里是主模块:

```
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)
```

(下页继续)

(续上页)

```
# 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()')
```

这里是辅助模块:

```
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"')
```

输出结果会像这样:

```
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()
```

(下页继续)

(续上页)

```
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 在多线程中使用日志

在多个线程中记录日志并不需要特殊处理，以下示例展示了如何在主线程（起始线程）和其他线程中记录:

```
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%(relativeCreated)6d %(threadName)s
↪%(message)s')
    info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

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

运行结果会像如下这样:

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
```

(下页继续)

(续上页)

```
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc
```

这表明不同线程的日志像期望的那样穿插输出，当然更多的线程也会像这样输出。

## 3 使用多个日志处理器和多种格式化

日志记录器是普通的 Python 对象。`addHandler()` 方法没有限制可以添加的日志处理器数量。有时候，应用程序需要将严重类的消息记录在一个文本文件，而将错误类或其他等级的消息输出在控制台中。要进行这样的设定，只需多配置几个日志处理器即可，在应用程序代码中的日志记录调用可以保持不变。以下是对之前的基于模块配置的示例的略微修改:

```
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')
```

需要注意的是，’应用程序’代码并不关心是否有多个日志处理器。示例中所做的改变只是添加和配置了一个新的名为 *fh* 的日志处理器。

在编写和测试应用程序时，创建能过滤不同等级消息的日志处理器是很有用的。不要去使用 `print` 去调试，而是使用 ‘‘logger.debug‘‘: 它不像打印语句需要在调试结束后注释或删除掉，你可以把它们保留在源码中并不输出。当需要再次调试时，只需要改变日志记录器或处理器的过滤等级即可。

## 4 在多个地方记录日志

假设有这样一种情况，你需要将日志按不同的格式和不同的情况存储在控制台和文件中。比如说想把日志等级为 DEBUG 或更高的消息记录于文件中，而把那些等级为 INFO 或更高的消息输出在控制台。而且记录在文件中的消息格式需要包含时间戳，打印在控制台的不需要。以下示例展示了如何做到:

```
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.')
```

当运行后，你会看到控制台如下所示

```
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.
```

而在文件中会看到像这样

```
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.
```

正如你所看到的，DEBUG 级别的消息只展示在文件中，而其他消息两个地方都会输出。

这个示例只演示了在控制台和文件中去记录日志，但你也可以自由组合任意数量的日志处理器。

## 5 日志服务器配置示例

以下是在一个模块中使用日志服务器配置的示例:

```
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()
```

然后如下的脚本，它接收文件名做为命令行参数，并将该文件以二进制编码的方式传给服务器，做为新的日志服务器配置:

```
#!/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)
```

(下页继续)

```
s.close()
print('complete')
```

## 6 处理日志处理器的阻塞

有时候需要让日志处理程序在不阻塞当前正在记录线程的情况下完成工作。这在 Web 应用程序中很常见，当然也会在其他场景中出现。

一个常见的缓慢行为是 SMTPHandler：由于开发者无法控制的多种原因（例如，性能不佳的邮件或网络基础架构），发送电子邮件可能需要很长时间。其实几乎所有基于网络的处理程序都可能造成阻塞：即便是 SocketHandler 也可能在底层进行 DNS 查询，这太慢了（这个查询会深入至套接字代码，位于 Python 层之下，这是不受开发者控制的）。

一种解决方案是分成两部分去处理。第一部分，针对那些对性能有要求的关键线程的日志记录附加一个 QueueHandler。日志记录器只需简单写入队列，该队列可以设置一个足够大的容量甚至不设置容量上限。通常写入队列是一个快速的操作，即使可能需要在代码中去捕获例如 queue.Full 等异常。如果你是一名处理关键线程的开发者，请务必记录这些信息（包括建议只为日志处理器附加 QueueHandlers）以便于其他开发者使用你的代码。

解决方案的另一部分是 QueueListener，它被设计用来作为 QueueHandler 的对应。QueueListener 非常简单：向其传入一个队列和一些处理句柄，它会启动一个内部线程来监听从 QueueHandlers（或任何其他可用的 LogRecords 源）发送过来的 LogRecords 队列。LogRecords 会从队列中被移除，并被传递给句柄进行处理。

使用一个单独的类 QueueListener 优点是可以使用同一个实例去服务于多个 “QueueHandlers“。这样会更节省资源，否则每个处理程序都占用一个线程没有任何益处。

以下是使用了这样两个类的示例（省略了导入语句）：

```
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()
```

在运行后会产生：

```
MainThread: Look out!
```

在 3.5 版更改：在 Python 3.5 之前，QueueListener 总是把从队列中接收的每个消息都传给它初始化的日志处理程序。(这是因为它会假设过滤级别总是在队列的另一侧去设置的。) 从 Python 3.5 开始，可以通过在监听器构造函数中添加一个参数 “respect_handler_level=True“改变这种情况。当这样设置时，监听器会比较每条消息的等级和日志处理器中设置的等级，只把需要传递的消息传给对应的日志处理器。

## 7 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)

# 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:

```
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:
```

(下页继续)

(续上页)

```
            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:
            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 = True

    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()
```

(下页继续)

(续上页)

```
            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.

# 8 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.

## 8.1 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`:

```
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 is a simple example:

```
class CustomAdapter(logging.LoggerAdapter):
    """
    This example adapter expects the passed in dict-like object to have a
    'connid' key, whose value in brackets is prepended to the log message.
    """
    def process(self, msg, kwargs):
        return '[%s] %s' % (self.extra['connid'], msg), kwargs
```

which you can use like this:

```
logger = logging.getLogger(__name__)
adapter = CustomAdapter(logger, {'connid': some_conn_id})
```

Then any events that you log to the adapter will have the value of `some_conn_id` prepended to the log messages.

#### Using objects other than dicts to pass contextual information

You don' t need to pass an actual dict to a `LoggerAdapter` - you could pass an instance of a class which implements `__getitem__` and `__iter__` so that it looks like a dict to logging. This would be useful if you want to generate values dynamically (whereas the values in a dict would be constant).

## 8.2 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:

```
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.
↪CRITICAL)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s IP: %(ip)-15s␣
↪User: %(user)-8s %(message)s')
    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 message
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1     User: sheila   An info␣
↪message with some parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A message at␣
↪CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1       User: jim      A message at␣
↪ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1       User: sheila   A message at␣
↪DEBUG level with 2 parameters
```

(下页继续)

(续上页)

```
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User: fred     A message at␣
↪ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1     User: jim      A message at␣
↪CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A message at␣
↪CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1     User: jim      A message at␣
↪DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1       User: sheila   A message at␣
↪ERROR level with 2 parameters
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User: fred     A message at␣
↪DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123 User: fred     A message at␣
↪INFO level with 2 parameters
```

## 9 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.) *This 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 https://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:

```
# 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
```

(下页继续)

(续上页)

```
# 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 avoid
# 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
↪%(message)s')
    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!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            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.
```

(下页继续)

(续上页)

```
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue)  # Just the one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter logic applied.
    root.setLevel(logging.DEBUG)

# 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()
    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:

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

(下页继续)

(续上页)

```
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
↪%(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                'mode': 'w',
                'formatter': 'detailed',
            },
            'foofile': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-foo.log',
                'mode': 'w',
                'formatter': 'detailed',
```

(下页继续)

(续上页)

```
            },
            '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.

## 10 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`:

```
import glob
import logging
import logging.handlers
```

(下页继续)

(续上页)

```
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 too small as an extreme example. You would want to set *maxBytes* to an appropriate value.

## 11 Use of alternative formatting styles

When logging was added to the Python standard library, the only way of formatting messages with variable content was to use the %-formatting method. Since then, Python has gained two new formatting approaches: `string.Template` (added in Python 2.4) and `str.format()` (added in Python 2.6).

Logging (as of 3.2) provides improved support for these two additional formatting styles. The `Formatter` class been enhanced to take an additional, optional keyword parameter named `style`. This defaults to `'%'`, but other possible values are `'{'` and `'$'`, which correspond to the other two formatting styles. Backwards compatibility is maintained by default (as you would expect), but by explicitly specifying a style parameter, you get the ability to specify format strings which work with `str.format()` or `string.Template`. Here' s an example console session to show the possibilities:

```
>>> import logging
>>> root = logging.getLogger()
>>> root.setLevel(logging.DEBUG)
>>> handler = logging.StreamHandler()
>>> bf = logging.Formatter('{asctime} {name} {levelname:8s} {message}',
...                        style='{')
>>> handler.setFormatter(bf)
>>> root.addHandler(handler)
>>> logger = logging.getLogger('foo.bar')
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:11:55,341 foo.bar DEBUG    This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:12:11,526 foo.bar CRITICAL This is a CRITICAL message
>>> df = logging.Formatter('$asctime $name ${levelname} $message',
...                        style='$')
>>> handler.setFormatter(df)
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:13:06,924 foo.bar DEBUG This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:13:11,494 foo.bar CRITICAL This is a CRITICAL message
>>>
```

Note that the formatting of logging messages for final output to logs is completely independent of how an individual logging message is constructed. That can still use %-formatting, as shown here:

```
>>> logger.error('This is an%s %s %s', 'other,', 'ERROR,', 'message')
2010-10-28 15:19:29,833 foo.bar ERROR This is another, ERROR, message
>>>
```

Logging calls (`logger.debug()`, `logger.info()` etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the actual logging call (e.g. the `exc_info` keyword parameter to indicate that traceback information should be logged, or the `extra` keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using `str.format()` or `string.Template` syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There is, however, a way that you can use {}- and $- formatting to construct your individual log messages. Recall that for a message you can use an arbitrary object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes:

```
class BraceMessage:
    def __init__(self, fmt, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, **kwargs):
```

(下页继续)

(续上页)

```
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

Either of these can be used in place of a format string, to allow {}- or $-formatting to be used to build the actual "message" part which appears in the formatted log output in place of "%(message)s" or "{message}" or "$message". It's a little unwieldy to use the class names whenever you want to log something, but it's quite palatable if you use an alias such as ___ (double underscore —not to be confused with _, the single underscore used as a synonym/alias for `gettext.gettext()` or its brethren).

The above classes are not included in Python, though they're easy enough to copy and paste into your own code. They can be used as follows (assuming that they're declared in a module called `wherever`):

```
>>> from wherever import BraceMessage as __
>>> print(__('Message with {0} {name}', 2, name='placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})',
...       point=p))
Message with coordinates: (0.50, 0.50)
>>> from wherever import DollarMessage as __
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>
```

While the above examples use `print()` to show how the formatting works, you would of course use `logger.debug()` or similar to actually log using this approach.

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That's because the ___ notation is just syntax sugar for a constructor call to one of the XXXMessage classes.

If you prefer, you can use a `LoggerAdapter` to achieve a similar effect to the above, as in the following example:

```
import logging

class Message(object):
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

    def __str__(self):
        return self.fmt.format(*self.args)
```

(下页继续)

(续上页)

```
class StyleAdapter(logging.LoggerAdapter):
    def __init__(self, logger, extra=None):
        super(StyleAdapter, self).__init__(logger, extra or {})

    def log(self, level, msg, *args, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger._log(level, Message(msg, args), (), **kwargs)

logger = StyleAdapter(logging.getLogger(__name__))

def main():
    logger.debug('Hello, {}', 'world!')

if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    main()
```

The above script should log the message `Hello, world!` when run with Python 3.2 or later.

## 12 Customizing `LogRecord`

Every logging event is represented by a `LogRecord` instance. When an event is logged and not filtered out by a logger's level, a `LogRecord` is created, populated with information about the event and then passed to the handlers for that logger (and its ancestors, up to and including the logger where further propagation up the hierarchy is disabled). Before Python 3.2, there were only two places where this creation was done:

- `Logger.makeRecord()`, which is called in the normal process of logging an event. This invoked `LogRecord` directly to create an instance.
- `makeLogRecord()`, which is called with a dictionary containing attributes to be added to the LogRecord. This is typically invoked when a suitable dictionary has been received over the network (e.g. in pickle form via a `SocketHandler`, or in JSON form via an `HTTPHandler`).

This has usually meant that if you need to do anything special with a `LogRecord`, you've had to do one of the following.

- Create your own `Logger` subclass, which overrides `Logger.makeRecord()`, and set it using `setLoggerClass()` before any loggers that you care about are instantiated.
- Add a `Filter` to a logger or handler, which does the necessary special manipulation you need when its `filter()` method is called.

The first approach would be a little unwieldy in the scenario where (say) several different libraries wanted to do different things. Each would attempt to set its own `Logger` subclass, and the one which did this last would win.

The second approach works reasonably well for many cases, but does not allow you to e.g. use a specialized subclass of `LogRecord`. Library developers can set a suitable filter on their loggers, but they would have to remember to do this every time they introduced a new logger (which they would do simply by adding new packages or modules and doing

```
logger = logging.getLogger(__name__)
```

at module level). It' s probably one too many things to think about. Developers could also add the filter to a `NullHandler` attached to their top-level logger, but this would not be invoked if an application developer attached a handler to a lower-level library logger —so output from that handler would not reflect the intentions of the library developer.

In Python 3.2 and later, `LogRecord` creation is done through a factory, which you can specify. The factory is just a callable you can set with `setLogRecordFactory()`, and interrogate with `getLogRecordFactory()`. The factory is invoked with the same signature as the `LogRecord` constructor, as `LogRecord` is the default setting for the factory.

This approach allows a custom factory to control all aspects of LogRecord creation. For example, you could return a subclass, or just add some additional attributes to the record once created, using a pattern similar to this:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

This pattern allows different libraries to chain factories together, and as long as they don' t overwrite each other' s attributes or unintentionally overwrite the attributes provided as standard, there should be no surprises. However, it should be borne in mind that each link in the chain adds run-time overhead to all logging operations, and the technique should only be used when the use of a `Filter` does not provide the desired result.

## 13 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' ):

```
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):
        self.queue.send_json(record.__dict__)


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:

```
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)
        super().__init__(socket)

    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

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

## 14 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:

```
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_string(zmq.SUBSCRIBE, '')  # subscribe to everything
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self):
        msg = self.queue.recv_json()
        return logging.makeLogRecord(msg)
```

**参见:**

**模块 `logging`** 日志记录模块的 API 参考。

**模块 `logging.config`** 日志记录模块的配置 API 。

**模块 `logging.handlers`** 日志记录模块附带的有用处理程序。

A basic logging tutorial

A more advanced logging tutorial

## 15 An example dictionary-based configuration

Below is an example of a logging configuration dictionary - it' s taken from the documentation on the Django project. This dictionary is passed to `dictConfig()` to put the configuration into effect:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
```

(下页继续)

(续上页)

```
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d
↪%(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level':'DEBUG',
            'class':'django.utils.log.NullHandler',
        },
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers':['null'],
            'propagate': True,
            'level':'INFO',
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

For more information about this configuration, you can see the relevant section of the Django documentation.

## 16 Using a rotator and namer to customize log rotation processing

An example of how you can define a namer and rotator is given in the following snippet, which shows zlib-based compression of the log file:

```
def namer(name):
    return name + ".gz"

def rotator(source, dest):
    with open(source, "rb") as sf:
        data = sf.read()
        compressed = zlib.compress(data, 9)
        with open(dest, "wb") as df:
            df.write(compressed)
    os.remove(source)

rh = logging.handlers.RotatingFileHandler(...)
rh.rotator = rotator
rh.namer = namer
```

These are not "true" .gz files, as they are bare compressed data, with no "container" such as you' d find in an actual gzip file. This snippet is just for illustration purposes.

## 17 A more elaborate multiprocessing example

The following working example shows how logging can be used with multiprocessing using configuration files. The configurations are fairly simple, but serve to illustrate how more complex ones could be implemented in a real multiprocessing scenario.

In the example, the main process spawns a listener process and some worker processes. Each of the main process, the listener and the workers have three separate configurations (the workers all share the same configuration). We can see logging in the main process, how the workers log to a QueueHandler and how the listener implements a QueueListener and a more complex logging configuration, and arranges to dispatch events received via the queue to the handlers specified in the configuration. Note that these configurations are purely illustrative, but you should be able to adapt this example to your own scenario.

Here' s the script - the docstrings and the comments hopefully explain how it works:

```
import logging
import logging.config
import logging.handlers
from multiprocessing import Process, Queue, Event, current_process
import os
import random
import time

class MyHandler:
    """
    A simple handler for logging events. It runs in the listener process and
    dispatches events to loggers based on the name in the received record,
    which then get dispatched, by the logging system, to the handlers
    configured for those loggers.
```

(下页继续)

(续上页)

```
    """
    def handle(self, record):
        logger = logging.getLogger(record.name)
        # The process name is transformed just to show that it's the listener
        # doing the logging to files and console
        record.processName = '%s (for %s)' % (current_process().name, record.processName)
        logger.handle(record)

def listener_process(q, stop_event, config):
    """
    This could be done in the main process, but is just done in a separate
    process for illustrative purposes.

    This initialises logging according to the specified configuration,
    starts the listener and waits for the main process to signal completion
    via the event. The listener is then stopped, and the process exits.
    """
    logging.config.dictConfig(config)
    listener = logging.handlers.QueueListener(q, MyHandler())
    listener.start()
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    stop_event.wait()
    listener.stop()

def worker_process(config):
    """
    A number of these are spawned for the purpose of illustration. In
    practice, they could be a heterogeneous bunch of processes rather than
    ones which are identical to each other.

    This initialises logging according to the specified configuration,
    and logs a hundred messages with random levels to randomly selected
    loggers.

    A small sleep is added to allow other processes a chance to run. This
    is not strictly needed, but it mixes the output from the different
    processes a bit more than if it's left out.
    """
    logging.config.dictConfig(config)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    if os.name == 'posix':
```

(下页继续)

(续上页)

```
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)
        time.sleep(0.01)

def main():
    q = Queue()
    # The main process gets a simple configuration which prints to the console.
    config_initial = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-10s
↪%(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
            },
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console']
        },
    }
    # The worker process configuration is just a QueueHandler attached to the
    # root logger, which allows all messages to be sent to the queue.
    # We disable existing loggers to disable the "setup" logger used in the
    # parent process. This is needed on POSIX because the logger will
    # be there in the child following a fork().
    config_worker = {
        'version': 1,
        'disable_existing_loggers': True,
        'handlers': {
            'queue': {
                'class': 'logging.handlers.QueueHandler',
                'queue': q,
            },
        },
        'root': {
```

(下页继续)

(续上页)

```
            'level': 'DEBUG',
            'handlers': ['queue']
        },
    }
    # The listener process configuration shows that the full flexibility of
    # logging configuration is available to dispatch events to handlers however
    # you want.
    # We disable existing loggers to disable the "setup" logger used in the
    # parent process. This is needed on POSIX because the logger will
    # be there in the child following a fork().
    config_listener = {
        'version': 1,
        'disable_existing_loggers': True,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-10s
↪%(message)s'
            },
            'simple': {
                'class': 'logging.Formatter',
                'format': '%(name)-15s %(levelname)-8s %(processName)-10s %(message)s'
            }
        },
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO',
                'formatter': 'simple',
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
                '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': {
```

(下页继续)

(续上页)

```
                'handlers': ['foofile']
            }
        },
        'root': {
            'level': 'DEBUG',
            'handlers': ['console', 'file', 'errors']
        },
    }
    # Log some initial events, just to show that logging in the parent works
    # normally.
    logging.config.dictConfig(config_initial)
    logger = logging.getLogger('setup')
    logger.info('About to create workers ...')
    workers = []
    for i in range(5):
        wp = Process(target=worker_process, name='worker %d' % (i + 1),
                     args=(config_worker,))
        workers.append(wp)
        wp.start()
        logger.info('Started worker: %s', wp.name)
    logger.info('About to create listener ...')
    stop_event = Event()
    lp = Process(target=listener_process, name='listener',
                 args=(q, stop_event, config_listener))
    lp.start()
    logger.info('Started listener')
    # We now hang around for the workers to finish their work.
    for wp in workers:
        wp.join()
    # Workers all done, listening can now stop.
    # Logging in the parent still works normally.
    logger.info('Telling listener to stop ...')
    stop_event.set()
    lp.join()
    logger.info('All done.')

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

# 18 Inserting a BOM into messages sent to a SysLogHandler

**RFC 5424** requires that a Unicode message be sent to a syslog daemon as a set of bytes which have the following structure: an optional pure-ASCII component, followed by a UTF-8 Byte Order Mark (BOM), followed by Unicode encoded using UTF-8. (See the **relevant section of the specification**.)

In Python 3.1, code was added to `SysLogHandler` to insert a BOM into the message, but unfortunately, it was implemented incorrectly, with the BOM appearing at the beginning of the message and hence not allowing any pure-ASCII component to appear before it.

As this behaviour is broken, the incorrect BOM insertion code is being removed from Python 3.2.4 and later. However, it is not being replaced, and if you want to produce **RFC 5424**-compliant messages which include

a BOM, an optional pure-ASCII sequence before it and arbitrary Unicode after it, encoded using UTF-8, then you need to do the following:

1. Attach a `Formatter` instance to your `SysLogHandler` instance, with a format string such as:

   ```
   'ASCII section\ufeffUnicode section'
   ```

   The Unicode code point U+FEFF, when encoded using UTF-8, will be encoded as a UTF-8 BOM – the byte-string `b'\xef\xbb\xbf'`.

2. Replace the ASCII section with whatever placeholders you like, but make sure that the data that appears in there after substitution is always ASCII (that way, it will remain unchanged after UTF-8 encoding).

3. Replace the Unicode section with whatever placeholders you like; if the data which appears there after substitution contains characters outside the ASCII range, that' s fine – it will be encoded using UTF-8.

The formatted message *will* be encoded using UTF-8 encoding by `SysLogHandler`. If you follow the above rules, you should be able to produce **RFC 5424**-compliant messages. If you don' t, logging may not complain, but your messages will not be RFC 5424-compliant, and your syslog daemon may complain.

## 19 Implementing structured logging

Although most logging messages are intended for reading by humans, and thus not readily machine-parseable, there might be circumstances where you want to output messages in a structured format which *is* capable of being parsed by a program (without needing complex regular expressions to parse the log message). This is straightforward to achieve using the logging package. There are a number of ways in which this could be achieved, but the following is a simple approach which uses JSON to serialise the event in a machine-parseable manner:

```
import json
import logging

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message, json.dumps(self.kwargs))

_ = StructuredMessage   # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123, fnum=123.456))
```

If the above script is run, it prints:

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz", "foo": "bar"}
```

Note that the order of items might be different according to the version of Python used.

If you need more specialised processing, you can use a custom JSON encoder, as in the following complete example:

```
from __future__ import unicode_literals

import json
import logging

# This next bit is to ensure the script runs unchanged on 2.x and 3.x
try:
    unicode
except NameError:
    unicode = str

class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, unicode):
            return o.encode('unicode_escape').decode('ascii')
        return super(Encoder, self).default(o)

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage   # optional, to improve readability

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value={1, 2, 3}, snowman='\u2603'))

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

When the above script is run, it prints:

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```

Note that the order of items might be different according to the version of Python used.

## 20 Customizing handlers with `dictConfig()`

There are times when you want to customize logging handlers in particular ways, and if you use `dictConfig()` you may be able to do this without subclassing. As an example, consider that you may want to set the ownership of a log file. On POSIX, this is easily done using `shutil.chown()`, but the file handlers in the stdlib don' t offer built-in support. You can customize handler creation using a plain function such as:

```
def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
```

You can then specify, in a logging configuration passed to `dictConfig()`, that a logging handler be created by calling this function:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

```
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)

LOGGING = {
```

(下页继续)

(续上页)

```
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')
```

To run this, you will probably need to run as `root`:

```
$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log
```

Note that this example uses Python 3.3 because that' s where `shutil.chown()` makes an appearance. This approach should work with any Python version that supports `dictConfig()` - namely, Python 2.7, 3.2 or later. With pre-3.3 versions, you would need to implement the actual ownership change using e.g. `os.chown()`.

In practice, the handler-creating function may be in a utility module somewhere in your project. Instead of the line in the configuration:

```
'()': owned_file_handler,
```

you could use e.g.:

```
'()': 'ext://project.util.owned_file_handler',
```

where `project.util` can be replaced with the actual name of the package where the function resides. In the above working script, using `'ext://__main__.owned_file_handler'` should work. Here, the actual callable is resolved by `dictConfig()` from the `ext://` specification.

This example hopefully also points the way to how you could implement other types of file change - e.g. setting specific POSIX permission bits - in the same way, using `os.chmod()`.

Of course, the approach could also be extended to types of handler other than a `FileHandler` - for example, one of the rotating file handlers, or a different type of handler altogether.

# 21 Using particular formatting styles throughout your application

In Python 3.2, the `Formatter` gained a `style` keyword parameter which, while defaulting to `%` for backward compatibility, allowed the specification of `{` or `$` to support the formatting approaches supported by `str.format()` and `string.Template`. Note that this governs the formatting of logging messages for final output to logs, and is completely orthogonal to how an individual logging message is constructed.

Logging calls (`debug()`, `info()` etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the `exc_info` keyword parameter to indicate that traceback information should be logged, or the `extra` keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using `str.format()` or `string.Template` syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There have been suggestions to associate format styles with specific loggers, but that approach also runs into backward compatibility problems because any existing code could be using a given logger name and using %-formatting.

For logging to work interoperably between any third-party libraries and your code, decisions about formatting need to be made at the level of the individual logging call. This opens up a couple of ways in which alternative formatting styles can be accommodated.

## 21.1 Using LogRecord factories

In Python 3.2, along with the `Formatter` changes mentioned above, the logging package gained the ability to allow users to set their own `LogRecord` subclasses, using the `setLogRecordFactory()` function. You can use this to set your own subclass of `LogRecord`, which does the Right Thing by overriding the `getMessage()` method. The base class implementation of this method is where the `msg % args` formatting happens, and where you can substitute your alternate formatting; however, you should be careful to support all formatting styles and allow %-formatting as the default, to ensure interoperability with other code. Care should also be taken to call `str(self.msg)`, just as the base implementation does.

Refer to the reference documentation on `setLogRecordFactory()` and `LogRecord` for more information.

## 21.2 Using custom message objects

There is another, perhaps simpler way that you can use {}- and $- formatting to construct your individual log messages. You may recall (from arbitrary-object-messages) that when logging you can use an arbitrary

object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes:

```
class BraceMessage(object):
    def __init__(self, fmt, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage(object):
    def __init__(self, fmt, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

Either of these can be used in place of a format string, to allow {}- or $-formatting to be used to build the actual "message" part which appears in the formatted log output in place of "%(message)s" or "{message}" or "$message" . If you find it a little unwieldy to use the class names whenever you want to log something, you can make it more palatable if you use an alias such as `M` or `_` for the message (or perhaps `__`, if you are using `_` for localization).

Examples of this approach are given below. Firstly, formatting with `str.format()`:

```
>>> __ = BraceMessage
>>> print(__('Message with {0} {1}', 2, 'placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
```

Secondly, formatting with `string.Template`:

```
>>> __ = DollarMessage
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>
```

One thing to note is that you pay no significant performance penalty with this approach: the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That' s because the ___ notation is just syntax sugar for a constructor call to one of the `XXXMessage` classes shown above.

## 22 Configuring filters with `dictConfig()`

You *can* configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it' s only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example:

```
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow

LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'filters': ['myfilter']
        }
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')
```

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print:

```
changed: hello
```

which shows that the filter is working as configured.

A couple of extra points to note:

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form `ext://...` as described in logging-config-dict-externalobj. For example, you could have used the text `'ext://__main__.MyFilter'` instead of `MyFilter` in the above example.
- As well as for filters, this technique can also be used to configure custom handlers and formatters. See logging-config-dict-userdef for more information on how logging supports using user-defined objects in its configuration, and see the other cookbook recipe *Customizing handlers with dictConfig()* above.

## 23 Customized exception formatting

There might be times when you want to do customized exception formatting - for argument's sake, let's say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example:

```
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super(OneLineExceptionFormatter, self).formatException(exc_info)
        return repr(result)  # or format into one line however you want to

    def format(self, record):
        s = super(OneLineExceptionFormatter, self).format(record)
        if record.exc_text:
            s = s.replace('\n', '') + '|'
        return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(message)s|',
                                  '%d/%m/%Y %H:%M:%S')
    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
```

(下页继续)

(续上页)

```
        logging.exception('ZeroDivisionError: %s', e)

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

When run, this produces a file with exactly two lines:

```
28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: integer division or modulo by zero|
↪'Traceback (most recent call last):\n  File "logtest7.py", line 30, in main\n    x = 1␣
↪/ 0\nZeroDivisionError: integer division or modulo by zero'|
```

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The `traceback` module may be helpful for more specialized needs.

## 24 Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text-to-speech (TTS) functionality available in your system, even if it doesn't have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using `subprocess`. It's assumed here that TTS command line programs won't expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it's acceptable to have the messages spoken one at a time rather than concurrently, The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the `espeak` TTS package is available:

```
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
```

(下页继续)

(续上页)

```
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
    sys.exit(main())
```

When run, this script should say "Hello" and then "Goodbye" in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.

## 25 Buffering logging messages and outputting them conditionally

There might be situations where you want to log messages in a temporary area and only output them if a certain condition occurs. For example, you may want to start logging debug events in a function, and if the function completes without errors, you don't want to clutter the log with the collected debug information, but if there is an error, you want all the debug information to be output as well as the error.

Here is an example which shows how you could do this using a decorator for your functions where you want logging to behave this way. It makes use of the `logging.handlers.MemoryHandler`, which allows buffering of logged events until some condition occurs, at which point the buffered events are `flushed` - passed to another handler (the `target` handler) for processing. By default, the `MemoryHandler` flushed when its buffer gets filled up or an event whose level is greater than or equal to a specified threshold is seen. You can use this recipe with a more specialised subclass of `MemoryHandler` if you want custom flushing behavior.

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to `sys.stderr` to say what level it's about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at ERROR and CRITICAL levels - otherwise, it only logs at DEBUG, INFO and WARNING levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that's required. The decorator takes a logger as a parameter and attaches a memory handler for the duration of the call to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer. These default to a `StreamHandler` which writes to `sys.stderr`, `logging.ERROR` and `100` respectively.

Here's the script:

```
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level, target=target_handler)
```

(下页继续)

(续上页)

```
    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
                logger.removeHandler(handler)
        return wrapper

    return decorator

def write_line(s):
    sys.stderr.write('%s\n' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)
```

When this script is run, the following output should be observed:

```
Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
```

(下页继续)

(续上页)

```
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
about to log at CRITICAL ...
Actually logged at CRITICAL
```

As you can see, actual logging output only occurs when an event is logged whose severity is ERROR or greater, but in that case, any previous events at lower severities are also logged.

You can of course use the conventional means of decoration:

```
@log_if_errors(logger)
def foo(fail=False):
    ...
```

## 26 Formatting times using UTC (GMT) via configuration

Sometimes you want to format times using UTC, which can be done using a class such as *UTCFormatter*, shown below:

```
import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

and you can then use the `UTCFormatter` in your code instead of `Formatter`. If you want to do that via configuration, you can use the `dictConfig()` API with an approach illustrated by the following complete example:

```
import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
```

(下页继续)

(续上页)

```
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        }
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',
        },
        'console2': {
            'class': 'logging.StreamHandler',
            'formatter': 'local',
        },
    },
    'root': {
        'handlers': ['console1', 'console2'],
   }
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())
```

When this script is run, it should print something like:

```
2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29 2015
2015-10-17 13:53:29,501 The local time is Sat Oct 17 13:53:29 2015
```

showing how the time is formatted both as local time and UTC, one for each handler.

## 27 Using a context manager for selective logging

There are times when it would be useful to temporarily change the logging configuration and revert it back after doing something. For this, a context manager is the most obvious way of saving and restoring the logging context. Here is a simple example of such a context manager, which allows you to optionally change the logging level and add a logging handler purely in the scope of the context manager:

```
import logging
import sys

```

(下页继续)

(续上页)

```
class LoggingContext(object):
    def __init__(self, logger, level=None, handler=None, close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions
```

If you specify a level value, the logger' s level is set to that value in the scope of the with block covered by the context manager. If you specify a handler, it is added to the logger on entry to the block and removed on exit from the block. You can also ask the manager to close the handler for you on block exit - you could do this if you don' t need the handler any more.

To illustrate how it works, we can add the following block of code to the above:

```
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
    logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h, close=True):
        logger.debug('5. This should appear twice - once on stderr and once on stdout.')
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

We initially set the logger' s level to `INFO`, so message #1 appears and message #2 doesn' t. We then change the level to `DEBUG` temporarily in the following `with` block, and so message #3 appears. After the block exits, the logger' s level is restored to `INFO` and so message #4 doesn' t appear. In the next `with` block, we set the level to `DEBUG` again but also add a handler writing to `sys.stdout`. Thus, message #5 appears twice on the console (once via `stderr` and once via `stdout`). After the `with` statement' s completion, the status is as it was before so message #6 appears (like message #1) whereas message #7 doesn' t (just like message #2).

If we run the resulting script, the result is as follows:

```
$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

If we run it again, but pipe `stderr` to `/dev/null`, we see the following, which is the only message written to `stdout`:

```
$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on stdout.
```

Once again, but piping `stdout` to `/dev/null`, we get:

```
$ python logctx.py >/dev/null
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

In this case, the message #5 printed to `stdout` doesn' t appear, as expected.

Of course, the approach described here can be generalised, for example to attach logging filters temporarily. Note that the above code works in Python 2 as well as Python 3.

# 索引

## R

RFC
    RFC 5424, 30, 31
    RFC 5424#section-6, 30