Logging Cookbook

Release 3.12.0rc1

Guido van Rossum and the Python development team

August 05, 2023

Python Software Foundation
Email: docs@python.org

Contents

1 Using logging in multiple modules **3**

2 Logging from multiple threads **4**

3 Multiple handlers and formatters **5**

4 Logging to multiple destinations **6**

5 Custom handling of levels **7**

6 Configuration server example **10**

7 Dealing with handlers that block **11**

8 Sending and receiving logging events across a network **12**
8.1 Running a logging socket listener in production 14

9 Adding contextual information to your logging output **15**
9.1 Using LoggerAdapters to impart contextual information 15
9.2 Using Filters to impart contextual information 16

10 Use of `contextvars` **17**

11 Imparting contextual information in handlers **21**

12 Logging to a single file from multiple processes **21**
12.1 Using concurrent.futures.ProcessPoolExecutor 25
12.2 Deploying Web applications using Gunicorn and uWSGI 26

13 Using file rotation **26**

14 Use of alternative formatting styles **27**

15 Customizing `LogRecord` **29**

16 Subclassing QueueHandler - a ZeroMQ example **30**

17 Subclassing QueueListener - a ZeroMQ example **31**

18 An example dictionary-based configuration **31**

19 Using a rotator and namer to customize log rotation processing **32**

20 A more elaborate multiprocessing example **33**

21 Inserting a BOM into messages sent to a SysLogHandler **37**

22 Implementing structured logging **37**

23 Customizing handlers with `dictConfig()` **38**

24 Using particular formatting styles throughout your application **40**
24.1 Using LogRecord factories . . . 41
24.2 Using custom message objects . . . 41

25 Configuring filters with `dictConfig()` **42**

26 Customized exception formatting **43**

27 Speaking logging messages **44**

28 Buffering logging messages and outputting them conditionally **45**

29 Sending logging messages to email, with buffering **47**

30 Formatting times using UTC (GMT) via configuration **48**

31 Using a context manager for selective logging **49**

32 A CLI application starter template **51**

33 A Qt GUI for logging **53**

34 Logging to syslog with RFC5424 support **57**

35 How to treat a logger like an output stream **58**

36 Patterns to avoid **60**
36.1 Opening the same log file multiple times . . . 60
36.2 Using loggers as attributes in a class or passing them as parameters . . . 61
36.3 Adding handlers other than `NullHandler` to a logger in a library . . . 61
36.4 Creating a lot of loggers . . . 61

37 Other resources **61**

Index **62**

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. For links to tutorial and reference information, please see *Other resources*.

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:

```
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:

```
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 Logging from multiple threads

Logging from multiple threads requires no special effort. The following example shows logging from the main (initial) thread and another thread:

```
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()
```

When run, the script should print something like the following:

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
```

(continues on next page)

(continued from previous page)

```
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
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
```

This shows the logging output interspersed as one might expect. This approach works for more threads than shown here, of course.

3 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:

```
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.warning('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.

4 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:

```
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='/tmp/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.

Note that the above choice of log filename `/tmp/myapp.log` implies use of a standard location for temporary files on POSIX systems. On Windows, you may need to choose a different directory name for the log - just ensure that the directory exists and that you have the permissions to create and update files in it.

5 Custom handling of levels

Sometimes, you might want to do something slightly different from the standard handling of levels in handlers, where all levels above a threshold get processed by a handler. To do this, you need to use filters. Let's look at a scenario where you want to arrange things as follows:

- Send messages of severity `INFO` and `WARNING` to `sys.stdout`
- Send messages of severity `ERROR` and above to `sys.stderr`
- Send messages of severity `DEBUG` and above to file `app.log`

Suppose you configure logging with the following JSON:

```
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
            "formatter": "simple",
            "stream": "ext://sys.stdout"
        },
        "stderr": {
            "class": "logging.StreamHandler",
            "level": "ERROR",
            "formatter": "simple",
            "stream": "ext://sys.stderr"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "simple",
            "filename": "app.log",
            "mode": "w"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": [
            "stderr",
            "stdout",
            "file"
        ]
    }
}
```

This configuration does *almost* what we want, except that `sys.stdout` would show messages of severity `ERROR` and above as well as `INFO` and `WARNING` messages. To prevent this, we can set up a filter which excludes those messages and add it to the relevant handler. This can be configured by adding a `filters` section parallel to `formatters` and `handlers`:

```
{
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

and changing the section on the `stdout` handler to add it:

```
{
    "stdout": {
        "class": "logging.StreamHandler",
        "level": "INFO",
        "formatter": "simple",
        "stream": "ext://sys.stdout",
        "filters": ["warnings_and_below"]
    }
}
```

A filter is just a function, so we can define the `filter_maker` (a factory function) as follows:

```
def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter
```

This converts the string argument passed in to a numeric level, and returns a function which only returns `True` if the level of the passed in record is at or below the specified level. Note that in this example I have defined the `filter_maker` in a test script `main.py` that I run from the command line, so its module will be `__main__` - hence the `__main__.filter_maker` in the filter configuration. You will need to change that if you define it in a different module.

With the filter added, we can run `main.py`, which in full is:

```
import json
import logging
import logging.config

CONFIG = '''
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
            "formatter": "simple",
            "stream": "ext://sys.stdout",
            "filters": ["warnings_and_below"]
        },
        "stderr": {
```

(continues on next page)

(continued from previous page)

```
            "class": "logging.StreamHandler",
            "level": "ERROR",
            "formatter": "simple",
            "stream": "ext://sys.stderr"
        },
        "file": {
            "class": "logging.FileHandler",
            "formatter": "simple",
            "filename": "app.log",
            "mode": "w"
        }
    },
    "root": {
        "level": "DEBUG",
        "handlers": [
            "stderr",
            "stdout",
            "file"
        ]
    }
}
'''

def filter_maker(level):
    level = getattr(logging, level)

    def filter(record):
        return record.levelno <= level

    return filter

logging.config.dictConfig(json.loads(CONFIG))
logging.debug('A DEBUG message')
logging.info('An INFO message')
logging.warning('A WARNING message')
logging.error('An ERROR message')
logging.critical('A CRITICAL message')
```

And after running it like this:

```
python main.py 2>stderr.log >stdout.log
```

We can see the results are as expected:

```
$ more *.log
::::::::::::::
app.log
::::::::::::::
DEBUG    - A DEBUG message
INFO     - An INFO message
WARNING  - A WARNING message
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
::::::::::::::
stderr.log
::::::::::::::
ERROR    - An ERROR message
CRITICAL - A CRITICAL message
::::::::::::::
stdout.log
::::::::::::::
```

(continues on next page)

(continued from previous page)

```
INFO     - An INFO message
WARNING  - A WARNING message
```

6 Configuration server example

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

```
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.warning('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:

```
#!/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')
```

7 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!
```

Note: Although the earlier discussion wasn't specifically talking about async code, but rather about slow logging handlers, it should be noted that when logging from async code, network and even file handlers could lead to problems (blocking the event loop) because some logging is done from `asyncio` internals. It might be best, if any async code is used in an application, to use the above approach for logging, so that any blocking code runs only in the `QueueListener` thread.

Changed in version 3.5: Prior to Python 3.5, the `QueueListener` always passed every message received from the queue to every handler it was initialized with. (This was because it was assumed that level filtering was all done on the other side, where the queue is filled.) From 3.5 onwards, this behaviour can be changed by passing a keyword argument `respect_handler_level=True` to the listener's constructor. When this is done, the

listener compares the level of each message with the handler's level, and only passes a message to a handler if it's appropriate to do so.

8 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:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
```

(continues on next page)

(continued from previous page)

```
            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()
            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.

8.1 Running a logging socket listener in production

To run a logging listener in production, you may need to use a process-management tool such as Supervisor. Here is a Gist which provides the bare-bones files to run the above functionality using Supervisor. It consists of the following files:

File	Purpose
`prepare.sh`	A Bash script to prepare the environment for testing
`supervisor.conf`	The Supervisor configuration file, which has entries for the listener and a multi-process web application
`ensure_app.sh`	A Bash script to ensure that Supervisor is running with the above configuration
`log_listener.py`	The socket listener program which receives log events and records them to a file
`main.py`	A simple web application which performs logging via a socket connected to the listener
`webapp.json`	A JSON configuration file for the web application
`client.py`	A Python script to exercise the web application

The web application uses Gunicorn, which is a popular web application server that starts multiple worker processes to handle requests. This example setup shows how the workers can write to the same log file without conflicting with one another — they all go through the socket listener.

To test these files, do the following in a POSIX environment:

1. Download the Gist as a ZIP archive using the *Download ZIP* button.
2. Unzip the above files from the archive into a scratch directory.
3. In the scratch directory, run `bash prepare.sh` to get things ready. This creates a `run` subdirectory to contain Supervisor-related and log files, and a `venv` subdirectory to contain a virtual environment into which `bottle`, `gunicorn` and `supervisor` are installed.
4. Run `bash ensure_app.sh` to ensure that Supervisor is running with the above configuration.
5. Run `venv/bin/python client.py` to exercise the web application, which will lead to records being written to the log.
6. Inspect the log files in the `run` subdirectory. You should see the most recent log lines in files matching the pattern `app.log*`. They won't be in any particular order, since they have been handled concurrently by different worker processes in a non-deterministic way.
7. You can shut down the listener and the web application by running `venv/bin/supervisorctl -c supervisor.conf shutdown`.

You may need to tweak the configuration files in the unlikely event that the configured ports clash with something else in your test environment.

9 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.

9.1 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).

9.2 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
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
```

10 Use of `contextvars`

Since Python 3.7, the `contextvars` module has provided context-local storage which works for both `threading` and `asyncio` processing needs. This type of storage may thus be generally preferable to thread-locals. The following example shows how, in a multi-threaded environment, logs can populated with contextual information such as, for example, request attributes handled by web applications.

For the purposes of illustration, say that you have different web applications, each independent of the other but running in the same Python process and using a library common to them. How can each of these applications have their own log, where all logging messages from the library (and other request processing code) are directed to the appropriate application's log file, while including in the log additional contextual information such as client IP, HTTP request method and client username?

Let's assume that the library can be simulated by the following code:

```
# webapplib.py
import logging
import time

logger = logging.getLogger(__name__)

def useful():
    # Just a representative event logged from the library
    logger.debug('Hello from webapplib!')
    # Just sleep for a bit so other threads get to run
    time.sleep(0.01)
```

We can simulate the multiple web applications by means of two simple classes, `Request` and `WebApp`. These simulate how real threaded web applications work - each request is handled by a thread:

```
# main.py
import argparse
from contextvars import ContextVar
import logging
```

(continues on next page)

(continued from previous page)

```
import os
from random import choice
import threading
import webapplib

logger = logging.getLogger(__name__)
root = logging.getLogger()
root.setLevel(logging.DEBUG)

class Request:
    """
    A simple dummy request class which just holds dummy HTTP request method,
    client IP address and client username
    """
    def __init__(self, method, ip, user):
        self.method = method
        self.ip = ip
        self.user = user

# A dummy set of requests which will be used in the simulation - we'll just pick
# from this list randomly. Note that all GET requests are from 192.168.2.XXX
# addresses, whereas POST requests are from 192.16.3.XXX addresses. Three users
# are represented in the sample requests.

REQUESTS = [
    Request('GET', '192.168.2.20', 'jim'),
    Request('POST', '192.168.3.20', 'fred'),
    Request('GET', '192.168.2.21', 'sheila'),
    Request('POST', '192.168.3.21', 'jim'),
    Request('GET', '192.168.2.22', 'fred'),
    Request('POST', '192.168.3.22', 'sheila'),
]

# Note that the format string includes references to request context information
# such as HTTP method, client IP and username

formatter = logging.Formatter('%(threadName)-11s %(appName)s %(name)-9s %(user)-6s
↪%(ip)s %(method)-4s %(message)s')

# Create our context variables. These will be filled at the start of request
# processing, and used in the logging that happens during that processing

ctx_request = ContextVar('request')
ctx_appname = ContextVar('appname')

class InjectingFilter(logging.Filter):
    """
    A filter which injects context-specific information into logs and ensures
    that only information for a specific webapp is included in its log
    """
    def __init__(self, app):
        self.app = app

    def filter(self, record):
        request = ctx_request.get()
        record.method = request.method
        record.ip = request.ip
        record.user = request.user
        record.appName = appName = ctx_appname.get()
        return appName == self.app.name
```

(continues on next page)

(continued from previous page)

```
class WebApp:
    """
    A dummy web application class which has its own handler and filter for a
    webapp-specific log.
    """
    def __init__(self, name):
        self.name = name
        handler = logging.FileHandler(name + '.log', 'w')
        f = InjectingFilter(self)
        handler.setFormatter(formatter)
        handler.addFilter(f)
        root.addHandler(handler)
        self.num_requests = 0

    def process_request(self, request):
        """
        This is the dummy method for processing a request. It's called on a
        different thread for every request. We store the context information into
        the context vars before doing anything else.
        """
        ctx_request.set(request)
        ctx_appname.set(self.name)
        self.num_requests += 1
        logger.debug('Request processing started')
        webapplib.useful()
        logger.debug('Request processing finished')

def main():
    fn = os.path.splitext(os.path.basename(__file__))[0]
    adhf = argparse.ArgumentDefaultsHelpFormatter
    ap = argparse.ArgumentParser(formatter_class=adhf, prog=fn,
                                 description='Simulate a couple of web '
                                             'applications handling some '
                                             'requests, showing how request '
                                             'context can be used to '
                                             'populate logs')
    aa = ap.add_argument
    aa('--count', '-c', type=int, default=100, help='How many requests to simulate
↪')
    options = ap.parse_args()

    # Create the dummy webapps and put them in a list which we can use to select
    # from randomly
    app1 = WebApp('app1')
    app2 = WebApp('app2')
    apps = [app1, app2]
    threads = []
    # Add a common handler which will capture all events
    handler = logging.FileHandler('app.log', 'w')
    handler.setFormatter(formatter)
    root.addHandler(handler)

    # Generate calls to process requests
    for i in range(options.count):
        try:
            # Pick an app at random and a request for it to process
            app = choice(apps)
            request = choice(REQUESTS)
            # Process the request in its own thread
            t = threading.Thread(target=app.process_request, args=(request,))
            threads.append(t)
```

(continues on next page)

(continued from previous page)

```
            t.start()
        except KeyboardInterrupt:
            break

    # Wait for the threads to terminate
    for t in threads:
        t.join()

    for app in apps:
        print('%s processed %s requests' % (app.name, app.num_requests))

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

If you run the above, you should find that roughly half the requests go into `app1.log` and the rest into `app2.log`, and the all the requests are logged to `app.log`. Each webapp-specific log will contain only log entries for only that webapp, and the request information will be displayed consistently in the log (i.e. the information in each dummy request will always appear together in a log line). This is illustrated by the following shell output:

```
~/logging-contextual-webapp$ python main.py
app1 processed 51 requests
app2 processed 49 requests
~/logging-contextual-webapp$ wc -l *.log
  153 app1.log
  147 app2.log
  300 app.log
  600 total
~/logging-contextual-webapp$ head -3 app1.log
Thread-3 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
Thread-3 (process_request) app1 webapplib jim    192.168.3.21 POST Hello from␣
↪webapplib!
Thread-5 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
~/logging-contextual-webapp$ head -3 app2.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21 GET  Request␣
↪processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET  Hello from␣
↪webapplib!
Thread-2 (process_request) app2 __main__  jim    192.168.2.20 GET  Request␣
↪processing started
~/logging-contextual-webapp$ head app.log
Thread-1 (process_request) app2 __main__  sheila 192.168.2.21 GET  Request␣
↪processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET  Hello from␣
↪webapplib!
Thread-2 (process_request) app2 __main__  jim    192.168.2.20 GET  Request␣
↪processing started
Thread-3 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
Thread-2 (process_request) app2 webapplib jim    192.168.2.20 GET  Hello from␣
↪webapplib!
Thread-3 (process_request) app1 webapplib jim    192.168.3.21 POST Hello from␣
↪webapplib!
Thread-4 (process_request) app2 __main__  fred   192.168.2.22 GET  Request␣
↪processing started
Thread-5 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
Thread-4 (process_request) app2 webapplib fred   192.168.2.22 GET  Hello from␣
↪webapplib!
Thread-6 (process_request) app1 __main__  jim    192.168.3.21 POST Request␣
↪processing started
```

(continues on next page)

(continued from previous page)

```
~/logging-contextual-webapp$ grep app1 app1.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app2.log | wc -l
147
~/logging-contextual-webapp$ grep app1 app.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app.log | wc -l
147
```

11 Imparting contextual information in handlers

Each `Handler` has its own chain of filters. If you want to add contextual information to a `LogRecord` without leaking it to other handlers, you can use a filter that returns a new `LogRecord` instead of modifying it in-place, as shown in the following script:

```
import copy
import logging

def filter(record: logging.LogRecord):
    record = copy.copy(record)
    record.user = 'jim'
    return record

if __name__ == '__main__':
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    handler = logging.StreamHandler()
    formatter = logging.Formatter('%(message)s from %(user)-8s')
    handler.setFormatter(formatter)
    handler.addFilter(filter)
    logger.addHandler(handler)

    logger.info('A log message')
```

12 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.

You could also write your own handler which uses the `Lock` class from the `multiprocessing` 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
# 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',
```

(continues on next page)

(continued from previous page)

```
    '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)
    # 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
import time
```

(continues on next page)

(continued from previous page)

```
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',
            },
            'errors': {
                'class': 'logging.FileHandler',
                'filename': 'mplog-errors.log',
                'mode': 'w',
                'level': 'ERROR',
                'formatter': 'detailed',
            },
        },
        'loggers': {
            'foo': {
```

(continues on next page)

(continued from previous page)

```
                '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.

12.1 Using concurrent.futures.ProcessPoolExecutor

If you want to use `concurrent.futures.ProcessPoolExecutor` to start your worker processes, you need to create the queue slightly differently. Instead of

```
queue = multiprocessing.Queue(-1)
```

you should use

```
queue = multiprocessing.Manager().Queue(-1)  # also works with the examples above
```

and you can then replace the worker creation from this:

```
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()
```

to this (remembering to first import `concurrent.futures`):

```
with concurrent.futures.ProcessPoolExecutor(max_workers=10) as executor:
    for i in range(10):
        executor.submit(worker_process, queue, worker_configurer)
```

12.2 Deploying Web applications using Gunicorn and uWSGI

When deploying Web applications using Gunicorn or uWSGI (or similar), multiple worker processes are created to handle client requests. In such environments, avoid creating file-based handlers directly in your web application. Instead, use a `SocketHandler` to log from the web application to a listener in a separate process. This can be set up using a process management tool such as Supervisor - see *Running a logging socket listener in production* for more details.

13 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

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.

14 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 be 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
```

(continues on next page)

(continued from previous page)

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

class DollarMessage:
    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". 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:
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

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

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

(continues on next page)

(continued from previous page)

```
    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.

15 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.

16 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()
```

17 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)
```

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

18 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': {
        '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'
```

(continues on next page)

(continued from previous page)

```
        },
        '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.

19 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 runnable script, which shows gzip compression of the log file:

```
import gzip
import logging
import logging.handlers
import os
import shutil

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

def rotator(source, dest):
    with open(source, 'rb') as f_in:
        with gzip.open(dest, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
    os.remove(source)


rh = logging.handlers.RotatingFileHandler('rotated.log', maxBytes=128,␣
↪backupCount=5)
rh.rotator = rotator
rh.namer = namer

root = logging.getLogger()
root.setLevel(logging.INFO)
root.addHandler(rh)
f = logging.Formatter('%(asctime)s %(message)s')
rh.setFormatter(f)
```

(continues on next page)

(continued from previous page)

```
for i in range(1000):
    root.info(f'Message no. {i + 1}')
```

After running this, you will see six new files, five of which are compressed:

```
$ ls rotated.log*
rotated.log       rotated.log.2.gz  rotated.log.4.gz
rotated.log.1.gz  rotated.log.3.gz  rotated.log.5.gz
$ zcat rotated.log.1.gz
2023-01-20 02:28:17,767 Message no. 996
2023-01-20 02:28:17,767 Message no. 997
2023-01-20 02:28:17,767 Message no. 998
```

20 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.
    """

    def handle(self, record):
        if record.name == "root":
            logger = logging.getLogger()
        else:
            logger = logging.getLogger(record.name)

        if logger.isEnabledFor(record.levelno):
            # 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):
```

(continues on next page)

(continued from previous page)

```
    """
    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':
        # 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 = {
```

(continues on next page)

(continued from previous page)

```
        'version': 1,
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO'
            }
        },
        'root': {
            'handlers': ['console'],
            'level': 'DEBUG'
        }
    }
    # 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': {
            'handlers': ['queue'],
            'level': 'DEBUG'
        }
    }
    # 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',
                'formatter': 'simple',
                'level': 'INFO'
            },
            'file': {
                'class': 'logging.FileHandler',
                'filename': 'mplog.log',
```

(continues on next page)

(continued from previous page)

```
                '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',
                'formatter': 'detailed',
                'level': 'ERROR'
            }
        },
        'loggers': {
            'foo': {
                'handlers': ['foofile']
            }
        },
        'root': {
            'handlers': ['console', 'file', 'errors'],
            'level': 'DEBUG'
        }
    }
    # 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()
```

21 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.

22 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:
    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:

```
import json
import logging


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

class StructuredMessage:
    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.

23 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,
```

(continues on next page)

(continued from previous page)

```
    '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 = {
    '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',
```

(continues on next page)

(continued from previous page)

```
        },
    },
    '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.

24 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 be 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.

24.1 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.

24.2 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:
    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):
        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.

25 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)
```

(continues on next page)

(continued from previous page)

```
    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.

26 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().formatException(exc_info)
        return repr(result)  # or format into one line however you want to

    def format(self, record):
        s = super().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:
        logging.exception('ZeroDivisionError: %s', e)
```

(continues on next page)

(continued from previous page)

```
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.

27 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')
    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.

28 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 (number of records buffered). 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)

    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)
```

(continues on next page)

(continued from previous page)

```
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 ...
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)
```

(continues on next page)

(continued from previous page)

```
def foo(fail=False):
    ...
```

29 Sending logging messages to email, with buffering

To illustrate how you can send log messages via email, so that a set number of messages are sent per email, you can subclass `BufferingHandler`. In the following example, which you can adapt to suit your specific needs, a simple test harness is provided which allows you to run the script with command line arguments specifying what you typically need to send things via SMTP. (Run the downloaded script with the `-h` argument to see the required and optional arguments.)

```
import logging
import logging.handlers
import smtplib

class BufferingSMTPHandler(logging.handlers.BufferingHandler):
    def __init__(self, mailhost, port, username, password, fromaddr, toaddrs,
                 subject, capacity):
        logging.handlers.BufferingHandler.__init__(self, capacity)
        self.mailhost = mailhost
        self.mailport = port
        self.username = username
        self.password = password
        self.fromaddr = fromaddr
        if isinstance(toaddrs, str):
            toaddrs = [toaddrs]
        self.toaddrs = toaddrs
        self.subject = subject
        self.setFormatter(logging.Formatter("%(asctime)s %(levelname)-5s
↪%(message)s"))

    def flush(self):
        if len(self.buffer) > 0:
            try:
                smtp = smtplib.SMTP(self.mailhost, self.mailport)
                smtp.starttls()
                smtp.login(self.username, self.password)
                msg = "From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n" % (self.fromaddr,
↪ ','.join(self.toaddrs), self.subject)
                for record in self.buffer:
                    s = self.format(record)
                    msg = msg + s + "\r\n"
                smtp.sendmail(self.fromaddr, self.toaddrs, msg)
                smtp.quit()
            except Exception:
                if logging.raiseExceptions:
                    raise
            self.buffer = []

if __name__ == '__main__':
    import argparse

    ap = argparse.ArgumentParser()
    aa = ap.add_argument
    aa('host', metavar='HOST', help='SMTP server')
    aa('--port', '-p', type=int, default=587, help='SMTP port')
    aa('user', metavar='USER', help='SMTP username')
    aa('password', metavar='PASSWORD', help='SMTP password')
```

(continues on next page)

(continued from previous page)

```
    aa('to', metavar='TO', help='Addressee for emails')
    aa('sender', metavar='SENDER', help='Sender email address')
    aa('--subject', '-s',
       default='Test Logging email from Python logging module (buffering)',
       help='Subject of email')
    options = ap.parse_args()
    logger = logging.getLogger()
    logger.setLevel(logging.DEBUG)
    h = BufferingSMTPHandler(options.host, options.port, options.user,
                             options.password, options.sender,
                             options.to, options.subject, 10)
    logger.addHandler(h)
    for i in range(102):
        logger.info("Info index = %d", i)
    h.flush()
    h.close()
```

If you run this script and your SMTP server is correctly set up, you should find that it sends eleven emails to the addressee you specify. The first ten emails will each have ten log messages, and the eleventh will have two messages. That makes up 102 messages as specified in the script.

30 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):
    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',
```

(continues on next page)

(continued from previous page)

```
        },
        '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.

31 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

class LoggingContext:
    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.

32 A CLI application starter template

Here's an example which shows how you can:

- Use a logging level based on command-line arguments
- Dispatch to multiple subcommands in separate files, all logging at the same level in a consistent way
- Make use of simple, minimal configuration

Suppose we have a command-line application whose job is to stop, start or restart some services. This could be organised for the purposes of illustration as a file `app.py` that is the main script for the application, with individual commands implemented in `start.py`, `stop.py` and `restart.py`. Suppose further that we want to control the verbosity of the application via a command-line argument, defaulting to `logging.INFO`. Here's one way that `app.py` could be written:

```
import argparse
import importlib
import logging
import os
import sys

def main(args=None):
    scriptname = os.path.basename(__file__)
    parser = argparse.ArgumentParser(scriptname)
    levels = ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
    parser.add_argument('--log-level', default='INFO', choices=levels)
    subparsers = parser.add_subparsers(dest='command',
                                       help='Available commands:')
    start_cmd = subparsers.add_parser('start', help='Start a service')
    start_cmd.add_argument('name', metavar='NAME',
                           help='Name of service to start')
    stop_cmd = subparsers.add_parser('stop',
                                     help='Stop one or more services')
    stop_cmd.add_argument('names', metavar='NAME', nargs='+',
                          help='Name of service to stop')
    restart_cmd = subparsers.add_parser('restart',
                                        help='Restart one or more services')
    restart_cmd.add_argument('names', metavar='NAME', nargs='+',
                             help='Name of service to restart')
    options = parser.parse_args()
    # the code to dispatch commands could all be in this file. For the purposes
    # of illustration only, we implement each command in a separate module.
    try:
        mod = importlib.import_module(options.command)
        cmd = getattr(mod, 'command')
    except (ImportError, AttributeError):
        print('Unable to find the code for command \'%s\'' % options.command)
        return 1
    # Could get fancy here and load configuration from file or dictionary
    logging.basicConfig(level=options.log_level,
                        format='%(levelname)s %(name)s %(message)s')
    cmd(options)

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

And the `start`, `stop` and `restart` commands can be implemented in separate modules, like so for starting:

```
# start.py
import logging

logger = logging.getLogger(__name__)
```

(continues on next page)

(continued from previous page)

```
def command(options):
    logger.debug('About to start %s', options.name)
    # actually do the command processing here ...
    logger.info('Started the \'%s\' service.', options.name)
```

and thus for stopping:

```
# stop.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to stop %s', services)
    # actually do the command processing here ...
    logger.info('Stopped the %s service%s.', services, plural)
```

and similarly for restarting:

```
# restart.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to restart %s', services)
    # actually do the command processing here ...
    logger.info('Restarted the %s service%s.', services, plural)
```

If we run this application with the default log level, we get output like this:

```
$ python app.py start foo
INFO start Started the 'foo' service.

$ python app.py stop foo bar
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py restart foo bar baz
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

The first word is the logging level, and the second word is the module or package name of the place where the event was logged.

If we change the logging level, then we can change the information sent to the log. For example, if we want more information:

```
$ python app.py --log-level DEBUG start foo
DEBUG start About to start foo
INFO start Started the 'foo' service.

$ python app.py --log-level DEBUG stop foo bar
DEBUG stop About to stop 'foo' and 'bar'
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py --log-level DEBUG restart foo bar baz
DEBUG restart About to restart 'foo', 'bar' and 'baz'
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

And if we want less:

```
$ python app.py --log-level WARNING start foo
$ python app.py --log-level WARNING stop foo bar
$ python app.py --log-level WARNING restart foo bar baz
```

In this case, the commands don't print anything to the console, since nothing at `WARNING` level or above is logged by them.

33 A Qt GUI for logging

A question that comes up from time to time is about how to log to a GUI application. The Qt framework is a popular cross-platform UI framework with Python bindings using PySide2 or PyQt5 libraries.

The following example shows how to log to a Qt GUI. This introduces a simple `QtHandler` class which takes a callable, which should be a slot in the main thread that does GUI updates. A worker thread is also created to show how you can log to the GUI from both the UI itself (via a button for manual logging) as well as a worker thread doing work in the background (here, just logging messages at random levels with random short delays in between).

The worker thread is implemented using Qt's `QThread` class rather than the `threading` module, as there are circumstances where one has to use `QThread`, which offers better integration with other `Qt` components.

The code should work with recent releases of either `PySide2` or `PyQt5`. You should be able to adapt the approach to earlier versions of Qt. Please refer to the comments in the code snippet for more detailed information.

```
import datetime
import logging
import random
import sys
import time

# Deal with minor differences between PySide2 and PyQt5
try:
    from PySide2 import QtCore, QtGui, QtWidgets
    Signal = QtCore.Signal
    Slot = QtCore.Slot
except ImportError:
    from PyQt5 import QtCore, QtGui, QtWidgets
    Signal = QtCore.pyqtSignal
    Slot = QtCore.pyqtSlot


logger = logging.getLogger(__name__)


#
```

(continues on next page)

(continued from previous page)

```
# Signals need to be contained in a QObject or subclass in order to be correctly
# initialized.
#
class Signaller(QtCore.QObject):
    signal = Signal(str, logging.LogRecord)

#
# Output to a Qt GUI is only supposed to happen on the main thread. So, this
# handler is designed to take a slot function which is set up to run in the main
# thread. In this example, the function takes a string argument which is a
# formatted log message, and the log record which generated it. The formatted
# string is just a convenience - you could format a string for output any way
# you like in the slot function itself.
#
# You specify the slot function to do whatever GUI updates you want. The handler
# doesn't know or care about specific UI elements.
#
class QtHandler(logging.Handler):
    def __init__(self, slotfunc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.signaller = Signaller()
        self.signaller.signal.connect(slotfunc)

    def emit(self, record):
        s = self.format(record)
        self.signaller.signal.emit(s, record)

#
# This example uses QThreads, which means that the threads at the Python level
# are named something like "Dummy-1". The function below gets the Qt name of the
# current thread.
#
def ctname():
    return QtCore.QThread.currentThread().objectName()


#
# Used to generate random levels for logging.
#
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)

#
# This worker class represents work that is done in a thread separate to the
# main thread. The way the thread is kicked off to do work is via a button press
# that connects to a slot in the worker.
#
# Because the default threadName value in the LogRecord isn't much use, we add
# a qThreadName which contains the QThread name as computed above, and pass that
# value in an "extra" dictionary which is used to update the LogRecord with the
# QThread name.
#
# This example worker just outputs messages sequentially, interspersed with
# random delays of the order of a few seconds.
#
class Worker(QtCore.QObject):
    @Slot()
    def start(self):
        extra = {'qThreadName': ctname() }
        logger.debug('Started work', extra=extra)
        i = 1
```

(continues on next page)

(continued from previous page)

```
        # Let the thread run until interrupted. This allows reasonably clean
        # thread termination.
        while not QtCore.QThread.currentThread().isInterruptionRequested():
            delay = 0.5 + random.random() * 2
            time.sleep(delay)
            level = random.choice(LEVELS)
            logger.log(level, 'Message after delay of %3.1f: %d', delay, i,␣
↪extra=extra)
            i += 1

#
# Implement a simple UI for this cookbook example. This contains:
#
# * A read-only text edit window which holds formatted log messages
# * A button to start work and log stuff in a separate thread
# * A button to log something from the main thread
# * A button to clear the log window
#
class Window(QtWidgets.QWidget):

    COLORS = {
        logging.DEBUG: 'black',
        logging.INFO: 'blue',
        logging.WARNING: 'orange',
        logging.ERROR: 'red',
        logging.CRITICAL: 'purple',
    }

    def __init__(self, app):
        super().__init__()
        self.app = app
        self.textedit = te = QtWidgets.QPlainTextEdit(self)
        # Set whatever the default monospace font is for the platform
        f = QtGui.QFont('nosuchfont')
        f.setStyleHint(f.Monospace)
        te.setFont(f)
        te.setReadOnly(True)
        PB = QtWidgets.QPushButton
        self.work_button = PB('Start background work', self)
        self.log_button = PB('Log a message at a random level', self)
        self.clear_button = PB('Clear log window', self)
        self.handler = h = QtHandler(self.update_status)
        # Remember to use qThreadName rather than threadName in the format string.
        fs = '%(asctime)s %(qThreadName)-12s %(levelname)-8s %(message)s'
        formatter = logging.Formatter(fs)
        h.setFormatter(formatter)
        logger.addHandler(h)
        # Set up to terminate the QThread when we exit
        app.aboutToQuit.connect(self.force_quit)

        # Lay out all the widgets
        layout = QtWidgets.QVBoxLayout(self)
        layout.addWidget(te)
        layout.addWidget(self.work_button)
        layout.addWidget(self.log_button)
        layout.addWidget(self.clear_button)
        self.setFixedSize(900, 400)

        # Connect the non-worker slots and signals
        self.log_button.clicked.connect(self.manual_update)
        self.clear_button.clicked.connect(self.clear_display)
```

(continues on next page)

(continued from previous page)

```
        # Start a new worker thread and connect the slots for the worker
        self.start_thread()
        self.work_button.clicked.connect(self.worker.start)
        # Once started, the button should be disabled
        self.work_button.clicked.connect(lambda : self.work_button.
↪setEnabled(False))

    def start_thread(self):
        self.worker = Worker()
        self.worker_thread = QtCore.QThread()
        self.worker.setObjectName('Worker')
        self.worker_thread.setObjectName('WorkerThread')  # for qThreadName
        self.worker.moveToThread(self.worker_thread)
        # This will start an event loop in the worker thread
        self.worker_thread.start()

    def kill_thread(self):
        # Just tell the worker to stop, then tell it to quit and wait for that
        # to happen
        self.worker_thread.requestInterruption()
        if self.worker_thread.isRunning():
            self.worker_thread.quit()
            self.worker_thread.wait()
        else:
            print('worker has already exited.')

    def force_quit(self):
        # For use when the window is closed
        if self.worker_thread.isRunning():
            self.kill_thread()

    # The functions below update the UI and run in the main thread because
    # that's where the slots are set up

    @Slot(str, logging.LogRecord)
    def update_status(self, status, record):
        color = self.COLORS.get(record.levelno, 'black')
        s = '<pre><font color="%s">%s</font></pre>' % (color, status)
        self.textedit.appendHtml(s)

    @Slot()
    def manual_update(self):
        # This function uses the formatted message passed in, but also uses
        # information from the record to format the message in an appropriate
        # color according to its severity (level).
        level = random.choice(LEVELS)
        extra = {'qThreadName': ctname() }
        logger.log(level, 'Manually logged!', extra=extra)

    @Slot()
    def clear_display(self):
        self.textedit.clear()


def main():
    QtCore.QThread.currentThread().setObjectName('MainThread')
    logging.getLogger().setLevel(logging.DEBUG)
    app = QtWidgets.QApplication(sys.argv)
    example = Window(app)
    example.show()
```

(continues on next page)

(continued from previous page)

```
    sys.exit(app.exec_())

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

34 Logging to syslog with RFC5424 support

Although RFC 5424 dates from 2009, most syslog servers are configured by default to use the older RFC 3164, which hails from 2001. When `logging` was added to Python in 2003, it supported the earlier (and only existing) protocol at the time. Since RFC5424 came out, as there has not been widespread deployment of it in syslog servers, the `SysLogHandler` functionality has not been updated.

RFC 5424 contains some useful features such as support for structured data, and if you need to be able to log to a syslog server with support for it, you can do so with a subclassed handler which looks something like this:

```
import datetime
import logging.handlers
import re
import socket
import time

class SysLogHandler5424(logging.handlers.SysLogHandler):

    tz_offset = re.compile(r'([+-]\d{2})(\d{2})$')
    escaped = re.compile(r'([\]"\\])')

    def __init__(self, *args, **kwargs):
        self.msgid = kwargs.pop('msgid', None)
        self.appname = kwargs.pop('appname', None)
        super().__init__(*args, **kwargs)

    def format(self, record):
        version = 1
        asctime = datetime.datetime.fromtimestamp(record.created).isoformat()
        m = self.tz_offset.match(time.strftime('%z'))
        has_offset = False
        if m and time.timezone:
            hrs, mins = m.groups()
            if int(hrs) or int(mins):
                has_offset = True
        if not has_offset:
            asctime += 'Z'
        else:
            asctime += f'{hrs}:{mins}'
        try:
            hostname = socket.gethostname()
        except Exception:
            hostname = '-'
        appname = self.appname or '-'
        procid = record.process
        msgid = '-'
        msg = super().format(record)
        sdata = '-'
        if hasattr(record, 'structured_data'):
            sd = record.structured_data
            # This should be a dict where the keys are SD-ID and the value is a
            # dict mapping PARAM-NAME to PARAM-VALUE (refer to the RFC for what␣
↪these
```

(continues on next page)

(continued from previous page)

```
            # mean)
            # There's no error checking here - it's purely for illustration, and␣
↪you
            # can adapt this code for use in production environments
            parts = []

            def replacer(m):
                g = m.groups()
                return '\\' + g[0]

            for sdid, dv in sd.items():
                part = f'[{sdid}'
                for k, v in dv.items():
                    s = str(v)
                    s = self.escaped.sub(replacer, s)
                    part += f' {k}="{s}"'
                part += ']'
                parts.append(part)
            sdata = ''.join(parts)
        return f'{version} {asctime} {hostname} {appname} {procid} {msgid} {sdata}
↪{msg}'
```

You'll need to be familiar with RFC 5424 to fully understand the above code, and it may be that you have slightly different needs (e.g. for how you pass structural data to the log). Nevertheless, the above should be adaptable to your speciric needs. With the above handler, you'd pass structured data using something like this:

```
sd = {
    'foo@12345': {'bar': 'baz', 'baz': 'bozz', 'fizz': r'buzz'},
    'foo@54321': {'rab': 'baz', 'zab': 'bozz', 'zzif': r'buzz'}
}
extra = {'structured_data': sd}
i = 1
logger.debug('Message %d', i, extra=extra)
```

35 How to treat a logger like an output stream

Sometimes, you need to interface to a third-party API which expects a file-like object to write to, but you want to direct the API's output to a logger. You can do this using a class which wraps a logger with a file-like API. Here's a short script illustrating such a class:

```
import logging

class LoggerWriter:
    def __init__(self, logger, level):
        self.logger = logger
        self.level = level

    def write(self, message):
        if message != '\n':  # avoid printing bare newlines, if you like
            self.logger.log(self.level, message)

    def flush(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation
        pass

    def close(self):
        # doesn't actually do anything, but might be expected of a file-like
```

(continues on next page)

(continued from previous page)

```
        # object - so optional depending on your situation. You might want
        # to set a flag so that later calls to write raise an exception
        pass

def main():
    logging.basicConfig(level=logging.DEBUG)
    logger = logging.getLogger('demo')
    info_fp = LoggerWriter(logger, logging.INFO)
    debug_fp = LoggerWriter(logger, logging.DEBUG)
    print('An INFO message', file=info_fp)
    print('A DEBUG message', file=debug_fp)

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

When this script is run, it prints

```
INFO:demo:An INFO message
DEBUG:demo:A DEBUG message
```

You could also use `LoggerWriter` to redirect `sys.stdout` and `sys.stderr` by doing something like this:

```
import sys

sys.stdout = LoggerWriter(logger, logging.INFO)
sys.stderr = LoggerWriter(logger, logging.WARNING)
```

You should do this *after* configuring logging for your needs. In the above example, the `basicConfig()` call does this (using the `sys.stderr` value *before* it is overwritten by a `LoggerWriter` instance). Then, you'd get this kind of result:

```
>>> print('Foo')
INFO:demo:Foo
>>> print('Bar', file=sys.stderr)
WARNING:demo:Bar
>>>
```

Of course, the examples above show output according to the format used by `basicConfig()`, but you can use a different formatter when you configure logging.

Note that with the above scheme, you are somewhat at the mercy of buffering and the sequence of write calls which you are intercepting. For example, with the definition of `LoggerWriter` above, if you have the snippet

```
sys.stderr = LoggerWriter(logger, logging.WARNING)
1 / 0
```

then running the script results in

```
WARNING:demo:Traceback (most recent call last):

WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 53, in
↪<module>

WARNING:demo:
WARNING:demo:main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 49, in main

WARNING:demo:
WARNING:demo:1 / 0
WARNING:demo:ZeroDivisionError
```

(continues on next page)

(continued from previous page)

```
WARNING:demo::
WARNING:demo:division by zero
```

As you can see, this output isn't ideal. That's because the underlying code which writes to `sys.stderr` makes multiple writes, each of which results in a separate logged line (for example, the last three lines above). To get around this problem, you need to buffer things and only output log lines when newlines are seen. Let's use a slghtly better implementation of `LoggerWriter`:

```
class BufferingLoggerWriter(LoggerWriter):
    def __init__(self, logger, level):
        super().__init__(logger, level)
        self.buffer = ''

    def write(self, message):
        if '\n' not in message:
            self.buffer += message
        else:
            parts = message.split('\n')
            if self.buffer:
                s = self.buffer + parts.pop(0)
                self.logger.log(self.level, s)
            self.buffer = parts.pop()
            for part in parts:
                self.logger.log(self.level, part)
```

This just buffers up stuff until a newline is seen, and then logs complete lines. With this approach, you get better output:

```
WARNING:demo:Traceback (most recent call last):
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 55, in
↪<module>
WARNING:demo:    main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 52, in main
WARNING:demo:    1/0
WARNING:demo:ZeroDivisionError: division by zero
```

36 Patterns to avoid

Although the preceding sections have described ways of doing things you might need to do or deal with, it is worth mentioning some usage patterns which are *unhelpful*, and which should therefore be avoided in most cases. The following sections are in no particular order.

36.1 Opening the same log file multiple times

On Windows, you will generally not be able to open the same file multiple times as this will lead to a "file is in use by another process" error. However, on POSIX platforms you'll not get any errors if you open the same file multiple times. This could be done accidentally, for example by:

- Adding a file handler more than once which references the same file (e.g. by a copy/paste/forget-to-change error).
- Opening two files that look different, as they have different names, but are the same because one is a symbolic link to the other.
- Forking a process, following which both parent and child have a reference to the same file. This might be through use of the `multiprocessing` module, for example.

Opening a file multiple times might *appear* to work most of the time, but can lead to a number of problems in practice:

- Logging output can be garbled because multiple threads or processes try to write to the same file. Although logging guards against concurrent use of the same handler instance by multiple threads, there is no such protection if concurrent writes are attempted by two different threads using two different handler instances which happen to point to the same file.
- An attempt to delete a file (e.g. during file rotation) silently fails, because there is another reference pointing to it. This can lead to confusion and wasted debugging time - log entries end up in unexpected places, or are lost altogether. Or a file that was supposed to be moved remains in place, and grows in size unexpectedly despite size-based rotation being supposedly in place.

Use the techniques outlined in *Logging to a single file from multiple processes* to circumvent such issues.

36.2 Using loggers as attributes in a class or passing them as parameters

While there might be unusual cases where you'll need to do this, in general there is no point because loggers are singletons. Code can always access a given logger instance by name using `logging.getLogger(name)`, so passing instances around and holding them as instance attributes is pointless. Note that in other languages such as Java and C#, loggers are often static class attributes. However, this pattern doesn't make sense in Python, where the module (and not the class) is the unit of software decomposition.

36.3 Adding handlers other than `NullHandler` to a logger in a library

Configuring logging by adding handlers, formatters and filters is the responsibility of the application developer, not the library developer. If you are maintaining a library, ensure that you don't add handlers to any of your loggers other than a `NullHandler` instance.

36.4 Creating a lot of loggers

Loggers are singletons that are never freed during a script execution, and so creating lots of loggers will use up memory which can't then be freed. Rather than create a logger per e.g. file processed or network connection made, use the *existing mechanisms* for passing contextual information into your logs and restrict the loggers created to those describing areas within your application (generally modules, but occasionally slightly more fine-grained than that).

37 Other resources

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.

Basic Tutorial

Advanced Tutorial

Index

R

RFC
    RFC 3164, 57
    RFC 5424, 37, 57
    RFC 5424#section-6, 37