
Logging クックブック

リリース 2.7.15

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

2月16, 2019

Python Software Foundation
Email: docs@python.org

目次

第 1 章	複数のモジュールで logging を使う	4
第 2 章	複数のスレッドからのロギング	6
第 3 章	複数の handler と formatter	8
第 4 章	複数の出力先にログを出力する	9
第 5 章	設定サーバの例	11
第 6 章	ネットワーク越しの logging イベントの送受信	12
第 7 章	コンテキスト情報をログ記録出力に付加する	15
7.1	LoggerAdapter を使ったコンテキスト情報の伝達	15
7.1.1	コンテキスト情報を渡すために dict 以外のオブジェクトを使う	16
7.2	Filter を使ったコンテキスト情報の伝達	16
第 8 章	複数のプロセスからの単一ファイルへのログ記録	18
第 9 章	ファイルをローテートする	19
第 10 章	辞書ベースで構成する例	20
第 11 章	SysLogHandler に送るメッセージに BOM を挿入する	22
第 12 章	構造化ログを実装する	23
第 13 章	handler を dictConfig() を使ってカスタマイズする	25
第 14 章	filter を dictConfig() を使ってカスタマイズする	28
第 15 章	例外の書式化をカスタマイズする	30
第 16 章	ロギングメッセージを喋る	31
第 17 章	ロギングメッセージをバッファリングし、条件に従って出力する	32
第 18 章	設定によって時刻を UTC(GMT) で書式化する	35
第 19 章	ロギングの選択にコンテキストマネージャを使う	37

このページでは、過去に見つけた logging に関するいくつかの便利なレシピを紹介します。

第1章 複数のモジュールで logging を使う

`logging.getLogger('someLogger')` の複数回の呼び出しは同じ logger への参照を返します。これは同じ Python インタプリタプロセス上で動いている限り、一つのモジュールの中からに限らず、モジュールをまたいでも当てはまります。同じオブジェクトへの参照という点でも正しいです。さらに、一つのモジュールの中で親 logger を定義して設定し、別のモジュールで子 logger を定義する (ただし設定はしない) ことが可能で、すべての子 logger への呼び出しは親にまで渡されます。まずはメインのモジュールです:

```
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()')
```

そして補助モジュール (auxiliary module) がこちらです:

```
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')
```

(次のページに続く)

(前のページからの続き)

```
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"')
```

出力はこのようになります:

```
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()
```

第2章 複数のスレッドからのロギング

複数スレッドからのロギングでは特別に何かをする必要はありません。次の例は main (初期) スレッドとそれ以外のスレッドからのロギングの例です:

```
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()
```

実行すると、このスクリプトの出力は次のようなものになるはずです:

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

予想した通りかもしれませんが、ログ出力が散らばっているのが分かります。もちろん、この手法はより多くのスレッドでも上手くいきます。

第3章 複数の handler と formatter

logger は通常の Python オブジェクトです。addHandler() メソッドは追加されるハンドラの個数について最小値も最大値も定めていません。時にアプリケーションがすべての深刻度のすべてのメッセージをテキストファイルに記録しつつ、同時にエラーやそれ以上のものをコンソールに出力することが役に立ちます。これを実現する方法は、単に適切なハンドラを設定するだけです。アプリケーションコードの中のログ記録の呼び出しは変更されずに残ります。少し前に取り上げた単純なモジュール式の例を少し変えとこうなります:

```
import logging

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

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

‘application’ 部分のコードは複数の handler について何も気にしていないことに注目してください。変更した箇所は新しい *fh* という名の handler を追加して設定したところがすべてです。

新しい handler を、異なる深刻度に対する filter と共に生成できることは、アプリケーションを書いてテストを行うときとても助けになります。デバッグ用にたくさんの print 文を使う代わりに logger.debug を使いましょう。あとで消したりコメントアウトしたりしなければならない print 文と違って、logger.debug 命令はソースコードの中にそのまま残しておいて再び必要になるまで休眠させておけます。その時必要になるのはただ logger および/または handler の深刻度の設定をいじることだけです。

第4章 複数の出力先にログを出力する

コンソールとファイルに、別々のメッセージ書式で、別々の状況に応じたログ出力を行わせたいとしましょう。例えば DEBUG よりも高いレベルのメッセージはファイルに記録し、INFO 以上のレベルのメッセージはコンソールに出力したいという場合です。また、ファイルにはタイムスタンプを記録し、コンソールには出力しないとします。以下のようにすれば、こうした挙動を実現できます：

```
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/temp/myapp.log',
                    filemode='w')

# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

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

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

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

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

このスクリプトを実行すると、コンソールには以下のように表示されるでしょう

```
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.
```

そして、ファイルは以下になるはずです

```
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.
```

(次のページに続く)

(前のページからの続き)

```
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.
```

これを見て分かる通り、DEBUG メッセージはファイルだけに出力され、その他のメッセージは両方に出力されます。

この例ではコンソールとファイルのハンドラだけを使っていますが、実際には任意の数のハンドラや組み合わせを使えます。

第5章 設定サーバの例

ログ記録設定サーバを使うモジュールの例です:

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

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

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

logger = logging.getLogger('simpleExample')

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

そしてファイル名を受け取ってそのファイルをサーバに送るスクリプトですが、それに先だってバイナリエンコード長を新しいログ記録の設定として先に送っておきます:

```
#!/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')
```

第6章 ネットワーク越しの logging イベントの送受信

ログイベントをネットワーク越しに送信し、受信端でそれを処理したいとしましょう。SocketHandler インスタンスを送信側の root logger にアタッチすれば、簡単に実現できます:

```
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.')
```

受信端では SocketServer モジュールを使って受信プログラムを作成しておきます。簡単な実用プログラムを以下に示します:

```
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,
```

(次のページに続く)

(前のページからの続き)

```

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

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

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because Logger.handle
        # is normally called AFTER logger-level filtering. If you want
        # to do filtering, do it at the client end to save wasting
        # cycles and network bandwidth!
        logger.handle(record)

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

    allow_reuse_address = 1

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

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

            if rd:
                self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')

```

(次のページに続く)

(前のページからの続き)

```
tcpserver = LogRecordSocketReceiver()
print('About to start TCP server...')
tcpserver.serve_until_stopped()

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

先にサーバを起動しておき、次にクライアントを起動します。クライアント側では、コンソールには何も出力されません; サーバ側では以下のようなメッセージを目にするはずです:

```
About to start TCP server...
59 root          INFO      Jackdaws love my big sphinx of quartz.
59 myapp.areal    DEBUG     Quick zephyrs blow, vexing daft Jim.
69 myapp.areal    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.
```

特定のシナリオでは pickle にはいくつかのセキュリティ上の問題があることに注意してください。これが問題になる場合、makePickle() メソッドをオーバーライドして代替手段を実装することで異なるシリアル化手法を使用できます。代替シリアル化手法を使うように上記のスクリプトを修正することもできます。

第7章 コンテキスト情報をログ記録出力に付加する

時にはログ記録出力にログ関数の呼び出し時に渡されたパラメータに加えてコンテキスト情報を含めたいこともあるでしょう。たとえば、ネットワークアプリケーションで、クライアント固有の情報 (例: リモートクライアントの名前、IP アドレス) もログ記録に残しておきたいと思ったとしましょう。 *extra* パラメータをこの目的に使うこともできますが、いつでもこの方法で情報を渡すのが便利なやり方とも限りません。また接続ごとに `Logger` インスタンスを生成する誘惑に駆られるかもしれませんが、生成した `Logger` インスタンスはガーベジコレクションで回収されないので、これは良いアイデアとは言えません。この例は現実的な問題ではないかもしれませんが、 `Logger` インスタンスの個数がアプリケーションの中でログ記録が行われるレベルの粒度に依存する場合、 `Logger` インスタンスの個数が事実上無制限にならないと、管理が難しくなります。

7.1 `LoggerAdapter` を使ったコンテキスト情報の伝達

logging イベントの情報と一緒に出力されるコンテキスト情報を渡す簡単な方法は、 `LoggerAdapter` を使うことです。このクラスは `Logger` のように見えるように設計されていて、 `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()`, `log()` の各メソッドを呼び出せるようになっています。これらのメソッドは対応する `Logger` のメソッドと同じ引数を取るので、二つの型を取り替えて使うことができます。

`LoggerAdapter` のインスタンスを生成する際には、 `Logger` インスタンスとコンテキスト情報を収めた辞書風 (dict-like) のオブジェクトを渡します。 `LoggerAdapter` のログ記録メソッドを呼び出すと、呼び出しをコンストラクタに渡された配下の `Logger` インスタンスに委譲し、その際コンテキスト情報をその委譲された呼び出しに埋め込みます。 `LoggerAdapter` のコードから少し抜き出してみます:

```
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)
```

`LoggerAdapter` の `process()` メソッドがコンテキスト情報をログ出力に加える場所です。そこではログ記録呼び出しのメッセージとキーワード引数が渡され、加工された (可能性のある) それらの情報を配下のロガーへの呼び出しに渡し直します。このメソッドのデフォルト実装ではメッセージは元のままですが、キーワード引数にはコンストラクタに渡された辞書風オブジェクトを値として "extra" キーが挿入されます。もちろん、呼び出し時に "extra" キーワードを使った場合には何事もなかったかのように上書きされます。

"extra" を用いる利点は辞書風オブジェクトの中の値が `LogRecord` インスタンスの `_dict_` にマージされることで、辞書風オブジェクトのキーを知っている `Formatter` を用意して文字列をカスタマイズするよ

うにできることです。それ以外のメソッドが必要なとき、たとえばコンテキスト情報をメッセージの前や後ろにつなげたい場合には、`LoggerAdapter` から `process()` を望むようにオーバーライドしたサブクラスを作ることが必要なだけです。次に挙げるのはこのクラスを使った例で、コンストラクタで使われる「辞書風」オブジェクトにどの振る舞いが必要なのかも示しています:

```
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
```

これを次のように使うことができます:

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

これで、この adapter 経由でログした全てのイベントに対して、`some_conn_id` の値がログメッセージの前に追加されます。

7.1.1 コンテキスト情報を渡すために `dict` 以外のオブジェクトを使う

`LoggerAdapter` に渡すのは本物の `dict` でなくても構いません。 `__getitem__` と `__iter__` を実装して `logging` が辞書のように扱えるクラスのインスタンスを利用することができます。これは (`dict` の値が固定されるのに対して) 値を動的に生成できるので便利です。

7.2 Filter を使ったコンテキスト情報の伝達

ユーザ定義の `Filter` を使ってログ出力にコンテキスト情報を加えることもできます。 `Filter` インスタンスは、渡された `LogRecords` を修正することができます。これにより、適切なフォーマット文字列や必要なら `Formatter` を使って、出力となる属性を新しく追加することも出来ます。

例えば、web アプリケーションで、処理されるリクエスト (または、少なくともその重要な部分) は、スレッドローカル (`threading.local`) な変数に保存して、 `Filter` からアクセスすることで、 `LogRecord` にリクエストの情報を追加できます。例えば、リモート IP アドレスやリモートユーザのユーザ名にアクセスしたいなら、上述の `LoggerAdapter` の例のように属性名 `'ip'` や `'user'` を使うといったようにです。その場合、同じフォーマット文字列を使って以下に示すように似たような出力を得られます。これはスクリプトの例です:

```
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.
    """
```

(次のページに続く)

(前のページからの続き)

```

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')

```

そして、実行時に、以下ようになります:

```

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

```

第8章 複数のプロセスからの単一ファイルへのログ記録

ログ記録はスレッドセーフであり、単一プロセスの複数のスレッドからの単一ファイルへのログ記録はサポートされていますが、複数プロセスからの単一ファイルへのログ記録はサポートされません。なぜなら、複数のプロセスをまたいで単一のファイルへのアクセスを直列化する標準の方法が Python には存在しないからです。複数のプロセスから単一ファイルへログ記録しなければならないなら、最も良い方法は、すべてのプロセスが `SocketHandler` に対してログ記録を行い、独立したプロセスとしてソケットサーバを動かすことです。ソケットサーバはソケットから読み取ってファイルにログを書き出します。（この機能を実行するために、既存のプロセスの1つのスレッドを割り当てることもできます）この節では、このアプローチをさらに詳細に文書化しています。動作するソケット受信プログラムが含まれているので、アプリケーションに組み込むための出発点として使用できるでしょう。

`multiprocessing` モジュールを含む最近のバージョンの Python を使用しているなら、複数のプロセスからファイルへのアクセスを直列化するために `Lock` クラスを使って独自のハンドラを書くことができます。既存の `FileHandler` とそのサブクラスは現在のところ `multiprocessing` を利用していませんが、将来は利用するようになるかもしれません。現在のところ `multiprocessing` モジュールが提供するロック機能はすべてのプラットフォームで動作するわけではないことに注意してください (<https://bugs.python.org/issue3770> 参照)。

第9章 ファイルをローテートする

ログファイルがある大きさに達したら、新しいファイルを開いてそこにログを取りたいことがあります。そのファイルのある数だけ残し、その数のファイルが生成されたらファイルを循環し、ファイルの数も大きさも制限したいこともあるでしょう。この利用パターンのために、logging パッケージは `RotatingFileHandler` を提供しています:

```
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)
```

この結果として、アプリケーションのログ履歴の一部である、6 つに別れたファイルが得られます:

```
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
```

最新のファイルはいつでも `logging_rotatingfile_example.out` で、サイズの上限に達するたびに拡張子 `.1` を付けた名前に改名されます。既にあるバックアップファイルはその拡張子がインクリメントされ (`.1` が `.2` になるなど)、`.6` ファイルは消去されます。

明らかに、ここでは極端な例示のためにファイルの大きさをかなり小さな値に設定しています。実際に使うときは `maxBytes` を適切な値に設定してください。

第10章 辞書ベースで構成する例

次の例は辞書を使った logging の構成です。この例は Django プロジェクトのドキュメント から持ってきました。この辞書を `dictConfig()` に渡して設定を有効にします:

```
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'
        },
        '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',
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

```
        'filters': ['special']
    }
}
```

この構成方法についてのより詳しい情報は、Django のドキュメントの [該当のセクション](#) で見ることができます。

第11章 SysLogHandler に送るメッセージに BOM を挿入する

RFC 5424 は syslog デーモンに Unicode メッセージを送る時、次の構造を要求しています: オプションな プュア ASCII 部分、続けて UTF-8 の Byte Order Mark (BOM)、続けて UTF-8 でエンコードされた Unicode. (仕様の該当セクション を参照)

Python 2.6, 2.7 で SysLogHandler に、message に BOM を挿入するコードが追加されました。しかし、そのときの実装が悪くて、message の先頭に BOM をつけてしまうのでプュア ASCII 部分をその前に書くことができませんでした。

この動作は壊れているので、Python 2.7.4 以降では BOM を挿入するコードが、削除されました。書き直されるのではなく削除されたので、RFC 5424 準拠の、BOM と、オプションのプュア ASCII 部分を BOM の前に、任意の Unicode を BOM の後ろに持つ UTF-8 でエンコードされた message を生成したい場合、次の手順に従う必要があります:

1. SysLogHandler のインスタンスに、次のような format 文字列を持った Formatter インスタンスをアタッチする:

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

Unicode のコードポイント `u'\ufeff'` は、UTF-8 でエンコードすると BOM - `'\xef\xbb\xbf'` になります。

2. ASCII セクションを好きなブレースホルダに変更する。ただしその部分の置換結果が常に ASCII になるように注意する (UTF-8 でエンコードされてもその部分が変化しないようにする)。
3. Unicode セクションを任意のブレースホルダに置き換える。この部分を置換したデータに ASCII 外の文字が含まれていても、それは単に UTF-8 でエンコードされるだけです。

フォーマットされた message が Unicode の場合、SysLogHandler によって UTF-8 でエンコードされます。上のルールに従えば、RFC 5424 準拠のメッセージを生成できます。上のルールに従わない場合、logging は何もエラーを起こしませんが、message は RFC 5424 に準拠しない形で送られるので、syslog デーモン側で何かエラーが起こる可能性があります。

第12章 構造化ログを実装する

多くのログメッセージは人間が読むために書かれるため、機械的に処理しにくくなっていますが、場合によっては(複雑な正規表現を使ってログメッセージをパースしなくても)プログラムがパースできる構造化されたフォーマットでメッセージを出力したい場合があります。logging パッケージを使うと、これを簡単に実現できます。実現する方法は幾つもありますが、次の例は JSON を使ってイベントを、機械でパースできる形にシリアル化するための単純な方法です:

```
import json
import logging

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

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

_ = StructuredMessage    # optional, to improve readability

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

上のスクリプトを実行すると次のように出力されます:

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

要素の順序は Python のバージョンによって異なることに注意してください。

より特殊な処理が必要な場合、次の例のように、カスタムの JSON エンコーダを作ることができます:

```
from __future__ import unicode_literals

import json
import logging

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

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

(次のページに続く)

(前のページからの続き)

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

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

_ = StructuredMessage    # optional, to improve readability

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

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

上のスクリプトを実行すると次のように出力されます:

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

要素の順序は Python のバージョンによって異なることに注意してください。

第13章 handler を dictConfig() を使ってカスタマイズする

logging handler に特定のカスタマイズを何度もしたい場合で、dictConfig() を使っているなら、サブクラスを作らなくてもカスタマイズが可能です。例えば、ログファイルの owner を設定したいとします。これは POSIX 環境では os.chown() を使って簡単に実現できますが、標準ライブラリの file handler はこの機能を組み込みでサポートしていません。handler の生成を通常関数を使ってカスタマイズすることができます:

```
def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        import os, pwd, grp
        # convert user and group names to uid and gid
        uid = pwd.getpwnam(owner[0]).pw_uid
        gid = grp.getgrnam(owner[1]).gr_gid
        owner = (uid, gid)
        if not os.path.exists(filename):
            open(filename, 'a').close()
        os.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
```

そして、dictConfig() に渡される構成設定の中で、この関数を使って logging handler を生成するように指定することができます:

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

(次のページに続く)

(前のページからの続き)

```

        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

```

この例は説明用のものですが、owner の user と group を pulse に設定しています。これを動くスクリプトに `chowntest.py` に組み込んでみます:

```

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',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

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

```

これを実行するには、root 権限で実行する必要があるかもしれません:

```

$ 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

```

`shutil.chown()` が追加されたのが Python 3.3 からなので、この例は Python 3.3 を使っています。この

アプローチ自体は `dictConfig()` をサポートした全ての Python バージョン - Python 2.7, 3.2 以降 - で利用できます。3.3 以前のバージョンでは、オーナーを変更するのに `os.chown()` を利用する必要があるでしょう。

実際には、handler を生成する関数はプロジェクト内のどこかにあるユーティリティモジュールに置くことになるでしょう。設定の中で直接関数を参照する代わりに:

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

次のように書くこともできます:

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

`project.util` は関数がある実際の場所に置き換えてください。上のスクリプトでは `'ext://__main___.owned_file_handler'` で動くはずですが、`dictConfig()` は `ext://` から実際の callable を見つけます。

この例は他のファイルに対する変更を実装する例にもなっています。例えば `os.chmod()` を使って、同じ方法で POSIX パーミッションを設定できるでしょう。

もちろん、このアプローチは `FileHandler` 以外の handler、ローテートする file handler のいずれかやその他の handler にも適用できます。

第14章 filter を dictConfig() を使ってカスタマイズする

dictConfig() によってフィルタを設定 出来ます が、どうやってそれを行うのが初見では明快とは言えないでしょう(そのためのこのレシピです)。Filter のみが唯一標準ライブラリに含まれているだけですし、それは何の要求にも応えてはくれません(ただの基底クラスですから)ので、典型的には filter() メソッドをオーバーライドした Filter のサブクラスをあなた自身で定義する必要があります。これをするには、設定辞書内のフィルタ指定部分に、() キーでそのフィルタを作るのに使われる callable を指定してください(クラスを指定するのが最もわかりやすいですが、Filter インスタンスを返却する callable を提供することでも出来ます)。以下に完全な例を示します:

```
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)
```

(次のページに続く)

(前のページからの続き)

```
logging.debug('hello')
logging.debug('hello - noshow')
```

どのようにして設定データとして、そのインスタンスを構築する callable をキーワードパラメータの形で渡すのか、をこの例は教えてくれます。上記スクリプトは実行すると、このような出力をします:

```
changed: hello
```

設定した通りに動いていますね。

ほかにもいくつか特筆すべき点があります:

- 設定内で直接その callable を参照出来ない場合 (例えばそれが異なるモジュール内にあり、設定辞書のある場所からそれを直接インポート出来ない、など) には、`logging-config-dict-externalobj` に記述されている `ext://...` 形式を使えます。例えば、上記例のように `MyFilter` と指定する代わりに、`'ext://__main___.MyFilter'` と記述することが出来ます。
- フィルタについてとともに、このテクニックは、カスタムハンドラ、カスタムフォーマッタに対しても同様に使えます。ロギングが設定において、どのようにユーザ定義のオブジェクトをサポートするのかについてのさらなる詳細については、`logging-config-dict-userdef` と、本クックブックの上の方のレシピ [handler](#) を `dictConfig()` を使ってカスタマイズする を参照してください。

第15章 例外の書式化をカスタマイズする

例外の書式化をカスタマイズしたいことがあるでしょう - わかりやすさのために、例外情報がある場合でもロギイベントごとに一行に収まることを死守したいと望むとしましょう。フォーマッタのクラスをカスタマイズして、このように出来ます:

```
import logging

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

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

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

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

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

実行してみましょう。このように正確に2行の出力を生成します:

```
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'|
```

これは扱いとしては単純過ぎますが、例外情報をどのようにしてあなた好みの書式化出来るかを示しています。さらに特殊なニーズが必要な場合には `traceback` モジュールが有用です。

第16章 ログイングメッセージを喋る

ログイングメッセージを目で見る形式ではなく音で聴く形式として出力したい、という状況があるかもしれません。これはあなたのシステムで text-to-speech (TTS) 機能が利用可能であれば、容易です。それが Python バインディングを持っていないとも、です。ほとんどの TTS システムはあなたが実行出来るコマンドラインプログラムを持っていて、このことで、`subprocess` を使うことでハンドラが呼び出せます。ここでは、TTS コマンドラインプログラムはユーザとの対話を期待せず、完了には時間がかかり、そしてログメッセージの頻度はユーザをメッセージで圧倒してしまうほどには高くはなく、そして並列で喋るよりはメッセージ一つにつき一回喋ることが受け容れられる、としておきます。ここでお見せする実装例では、次が処理される前に一つのメッセージを喋り終わるまで待ち、結果としてほかのハンドラを待たせることになります。espeak TTS パッケージが手許にあるとして、このアプローチによる短い例はこのようなものです:

```
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())
```

実行すれば、女性の声で ”Hello” に続き ”Goodbye” と喋るはずです。

このアプローチは、もちろんほかの TTS システムにも採用出来ますし、メッセージをコマンドライン経由で外部プログラムに渡せるようなものであれば、ほかのシステムであっても全く同じです。

第17章 ログメッセージをバッファリングし、条件に従って出力する

メッセージを一次領域に記録し、ある種特定の状況になった場合にだけ出力したい、ということがあるかもしれません。たとえばある関数内でのデバッグのためのログ出力をしたくても、エラーなしで終了する限りにおいては収集されたデバッグ情報による混雑は喰らいたくはなく、エラーがあった場合にだけエラー出力とともにデバッグ情報を見たいのだ、のようなことがあるでしょう。

このような振る舞いをするロギングをしたい関数に対して、デコレータを用いてこれを行う例をお見せします。それには `logging.handlers.MemoryHandler` を使います。これにより何か条件を満たすまでロギングイベントを溜め込むことが出来、条件を満たせば溜め込まれたイベントが `flushed` として他のハンドラ (`target` のハンドラ) に渡されます。デフォルトでは、`MemoryHandler` はそのバッファが一杯になるか、指定された閾値のレベル以上のイベントが起こるとフラッシュされます。何か特別なフラッシュの振る舞いをしたければ、このレシピはさらに特殊化した `MemoryHandler` とともに利用出来ます。

スクリプト例では、`foo` という、単に全てのログレベルについて、`sys.stderr` にもどのレベルを出力したのかについて書き出ししながら実際のログ出力も行う、という単純な関数を使っています。`foo` に真を与えると `ERROR` と `CRITICAL` の出力をし、そうでなければ `DEBUG`, `INFO`, `WARNING` だけを出力します。

スクリプトが行うことは単に、`foo` を必要とされている特定の条件でのロギングを行うようにするデコレータで修飾することだけです。このデコレータはパラメータとしてロガーを取り、修飾された関数が呼ばれている間だけメモリハンドラをアタッチします。追加のパラメータとして、ターゲットのハンドラ、フラッシュが発生すべきレベル、バッファの容量も受け取れます。これらのデフォルトは順に `sys.stderr` へ書き出す `StreamHandler`, `logging.ERROR`, `100` です。

スクリプトはこれです:

```
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)
            result = fn(*args, **kwargs)
            logger.removeHandler(handler)
            return result
        return wrapper
```

(次のページに続く)

(前のページからの続き)

```

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

    return decorator

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

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

decorated_foo = log_if_errors(logger)(foo)

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

```

実行すればこのような出力になるはずです:

```

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 ...

```

(次のページに続く)

(前のページからの続き)

```
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
```

見ての通り、実際のログ出力は重要度 ERROR かそれより大きい場合にのみ行っていますが、この場合はそれよりも重要度の低い ERROR よりも前に発生したイベントも出力されます。

当然のことですが、デコレーションはいつものやり方でどうぞ:

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

第18章 設定によって時刻を UTC(GMT) で書式化する

場合によっては、時刻として UTC を使いたいと思うかもしれません。これには以下に示すような *UTCFormatter* のようなクラスを使って出来ます:

```
import logging
import time

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

そしてコード中で *UTCFormatter* を *Formatter* の代わりに使えます。これを設定を通して行いたい場合、*dictConfig()* API を以下の完全な例で示すようなアプローチで使うことが出来ます:

```
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',
        },
        'console2': {
            'class': 'logging.StreamHandler',
            'formatter': 'local',
        },
    },
    'root': {
        'handlers': ['console1', 'console2'],
    }
}

if __name__ == '__main__':
```

(次のページに続く)

(前のページからの続き)

```
logging.config.dictConfig(LOGGING)
logging.warning('The local time is %s', time.asctime())
```

実行すれば、このような出力になるはずです:

```
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
```

時刻をローカル時刻と UTC の両方に書式化するのに、それぞれのハンドラにそれぞれフォーマッタを与えています。

第19章 ログिंगの選択にコンテキストマネージャを使う

一時的にログिंगの設定を変えて、作業をした後に設定を戻せると便利なときがあります。こういうときの、ログिंगコンテキストの保存と復元をする方法ではコンテキストマネージャを使うのが一番です。以下にあるのがそのためのコンテキストマネージャの簡単な例で、これを使うと、任意にログingleベルを変更し、コンテキストマネージャのスコープ内で他に影響を及ぼさずログングハンドラを追加できるようになります:

```
import logging
import sys

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

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

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

レベル値を指定した場合、コンテキストマネージャがカバーする with ブロックのスコープ内でロガーのレベルがその値に設定されます。ハンドラを指定した場合、ブロックに入るときにロガーに追加され、ブロックから抜けるときに取り除かれます。ブロックを抜けるときに、自分で追加したハンドラをクローズするようコンテキストマネージャに指示することもできます - そのハンドラがそれ以降必要無いのであればクローズしてしまって構いません。

どのように動作するのかを示すためには、次のコード群を上コードに付け加えるとよいです:

```
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.')
```

(次のページに続く)

(前のページからの続き)

```

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.')

```

最初はロガーのレベルを INFO に設定しているので、メッセージ #1 は現れ、メッセージ #2 は現れません。次に、その後の with ブロック内で一時的にレベルを DEBUG に変更したため、メッセージ #3 が現れます。そのブロックを抜けた後、ロガーのレベルは INFO に復元され、メッセージ #4 は現れません。次の with ブロック内では、再度レベルを DEBUG に設定し、sys.stdout に書き出すハンドラを追加します。そのおかげでメッセージ #5 が 2 回 (1 回は stderr を通して、もう 1 回は stdout を通して) コンソールに出力されます。with 文が完了すると、前の状態になるので (メッセージ #1 のように) メッセージ #6 が現れ、(まさにメッセージ #2 のように) メッセージ #7 は現れません。

出来上がったスクリプトを実行すると、結果は次のようになります:

```

$ 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.

```

stderr を /dev/null へパイプした状態でもう一度実行すると、次のようになり、これは stdout の方向に書かれたメッセージだけが現れています:

```

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

```

stdout を /dev/null へパイプした状態でさらにもう一度実行すると、こうなります:

```

$ 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.

```

この場合では、stdout の方に出力されたメッセージ #5 は予想通り現れません。

もちろんここで説明した手法は、例えば一時的にロギングフィルターを取り付けたりするのに一般化できます。上のコードは Python 2 だけでなく Python 3 でも動くことに注意してください。