ロギングクックブック

リリース *2.7ja1*

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

2011 年 12 月 25 日

Python Software Foundation
Email: docs@python.org

目次

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

このページでは、過去に見つかった、ロギングに関するいくつかの便利なレシピを紹介します。

1 複数のモジュールでロギングを使う

`logging.getLogger('someLogger')` の複数回の呼び出しは同じロガー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ロガーを定義して設定し、別のモジュールで子ロガーを定義する (ただし設定はしない) ことが可能で、すべての子ロガー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')
    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()
```

2 複数のハンドラとフォーマッタ

ロガーは通常の 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るとこうなります:

```
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')
```

「アプリケーション」のコードは複数のハンドラ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* という名のハンドラを追加して設定したところがすべてです。

新しいハンドラを高い、もしくは低い深刻度に対するフィルタ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ロガーおよび/またはハンドラの深刻度の設定をいじることだけです。

3 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、ファイルは以下のようになるはずです:

```
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.
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。

4 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')
```

5 ネットワーク越しのロギングイベントの送信と受信

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端のルートロガーに接続すれば、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,
        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')
    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.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.
```

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ライズ手法を使用できます。代替シリアライズ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。

6 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。

6.1 LoggerAdapter を使ったコンテキスト情報の伝達

ログ記録イベントの情報と一緒に出力される文脈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` インスタンスと文脈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、その際文脈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() メソッドが文脈情報をログ出力に加える舞台です。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、たとえば文脈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:

```
import logging

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

    def __getitem__(self, name):
        """
        To allow this instance to look like a dict.
        """
        from random import choice
        if name == 'ip':
            result = choice(['127.0.0.1', '192.168.0.1'])
        elif name == 'user':
            result = choice(['jim', 'fred', 'sheila'])
        else:
            result = self.__dict__.get(name, '?')
        return result

    def __iter__(self):
        """
        To allow iteration over keys, which will be merged into
        the LogRecord dict before formatting and output.
```

```
        """
        keys = ['ip', 'user']
        keys.extend(self.__dict__.keys())
        return keys.__iter__()

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

このスクリプトが実行されると、出力は以下のようになります:

```
2008-01-18 14:49:54,023 a.b.c DEBUG    IP: 123.231.231.123 User: sheila   A debug message
2008-01-18 14:49:54,023 a.b.c INFO     IP: 123.231.231.123 User: sheila   An info message with so
2008-01-18 14:49:54,023 d.e.f CRITICAL IP: 192.168.0.1     User: jim      A message at CRITICAL le
2008-01-18 14:49:54,033 d.e.f INFO     IP: 192.168.0.1     User: jim      A message at INFO level
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: sheila   A message at WARNING lev
2008-01-18 14:49:54,033 d.e.f ERROR    IP: 127.0.0.1       User: fred     A message at ERROR level
2008-01-18 14:49:54,033 d.e.f ERROR    IP: 127.0.0.1       User: sheila   A message at ERROR level
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: sheila   A message at WARNING lev
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: jim      A message at WARNING lev
2008-01-18 14:49:54,033 d.e.f INFO     IP: 192.168.0.1     User: fred     A message at INFO level
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 192.168.0.1     User: sheila   A message at WARNING lev
2008-01-18 14:49:54,033 d.e.f WARNING  IP: 127.0.0.1       User: jim      A message at WARNING lev
```

6.2 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。そのばあい、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.
    """

    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: %(us
   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、以下のようになります:

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

7 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) 以下の節では、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 `multiprocessing` モジュールの `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ホームで動作するわけではないことに注意してください。(http://bugs.python.org/issue3770 参照)

8 ファイルの循環を使う

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。そのファイルをある数だけ残し、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、ここでは例示のためにファイルの大きさをとんでもなく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。