
Logging クックブック

リリース 3.14.0a0

Guido van Rossum and the Python development team

9月 25, 2024

目次

1	複数のモジュールで logging を使う	3
2	複数のスレッドからのロギング	5
3	複数の handler と formatter	6
4	複数の出力先にログを出力する	7
5	ログレベルのカスタム処理	8
6	設定サーバの例	12
7	ブロックする handler を扱う	13
8	ネットワーク越しの logging イベントの送受信	14
8.1	作成中のロギングソケットのリスナーを実行する	17
9	コンテキスト情報をログ記録出力に付加する	18
9.1	LoggerAdapter を使ったコンテキスト情報の伝達	18
9.2	Filter を使ったコンテキスト情報の伝達	20
10	contextvars の利用	21
11	ハンドラ内でのコンテキスト情報の付与	25
12	複数のプロセスからの単一ファイルへのログ記録	26
12.1	concurrent.futures.ProcessPoolExecutor の利用	31
12.2	Gunicorn と uWSGI を用いた Web アプリケーションのデプロイ	31
13	ファイルをローテートする	31
14	別の format スタイルを利用する	33
15	LogRecord のカスタマイズ	35

16	Subclassing QueueHandler and QueueListener- a ZeroMQ example	37
16.1	Subclass QueueHandler	37
16.2	Subclass QueueListener	37
17	Subclassing QueueHandler and QueueListener- a pyzmq example	38
17.1	Subclass QueueListener	38
17.2	Subclass QueueHandler	39
18	辞書ベースで構成する例	41
19	rotator と namer を使ってログローテートをカスタマイズする	42
20	より手の込んだ multiprocessing の例	43
21	SysLogHandler に送るメッセージに BOM を挿入する	48
22	構造化ログを実装する	48
23	handler を dictConfig() を使ってカスタマイズする	50
24	固有の書式化スタイルをアプリケーション全体で使う	52
24.1	LogRecord ファクトリを使う	53
24.2	カスタムなメッセージオブジェクトを使う	53
25	filter を dictConfig() を使ってカスタマイズする	54
26	例外の書式化をカスタマイズする	56
27	ロギングメッセージを喋る	57
28	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	58
29	バッファリングしながらロギングメッセージを email で送信する	60
30	設定によって時刻を UTC(GMT) で書式化する	62
31	ロギングの選択にコンテキストマネージャを使う	63
32	CLI アプリケーションスターターテンプレート	65
33	Qt GUI のログ出力	68
34	RFC5424 をサポートする syslog へのロギング	73
35	ロガーを出力ストリームのように扱う方法	74
36	避けるべきパターン	77
36.1	同じログファイルを何度も開く	77
36.2	ロガーをクラスの属性にするか、パラメータで渡す	78
36.3	ライブラリ内でロガーに NullHandler 以外のハンドラーを追加する	78
36.4	大量のロガーを作成する	78

著者

Vinay Sajip <vinay_sajip at red-dove dot com>

このページは、過去に有用であるとされていた、logging に関連するいくつかのレシピを含んでいます。チュートリアルやリファレンス情報へのリンクについては [その他のリソース](#) を参照してください。

1 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は限らず、モジュールをまたいで当てはまります。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')

    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、出力は以下のようになるはずです:

```
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。

3 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ようとなります:

```
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.warning('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。

4 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='/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.')
```

これを実行すると、コンソールには以下のように出力されます:

```
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、ファイルには以下のように出力されるはずです:

```
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)

```
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。

ここでファイル名として `/tmp/myapp.log` を選んだということは、一時ファイルの標準的な場所として POSIX システムを想定していることに注意してください。Windows の場合、ディレクトリが存在し、そのディレクトリに対してファイルを更新するための適切な権限を有することを保証するために、ログファイル向けのディレクトリ名として異なる選択を取る必要があるでしょう。

5 ログレベルのカスタム処理

しきい値以上のログレベル全てがハンドラによって処理される標準的な処理に対して、ときにはわずかに異なる振る舞いを必要とすることもあるでしょう。そのような場合はフィルタを使う必要があります。以下のような処理が必要なシナリオについて取り上げてみましょう:

- 深刻度 INFO と WARNING のメッセージを `sys.stdout` に送る
- 深刻度 ERROR 以上のメッセージを `sys.stderr` に送る
- 深刻度 DEBUG 以上のメッセージをファイル `app.log` に送る

logging モジュールを下記の JSON によって構成したとしましょう:

```
{
  "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"
    }
  }
}
```

(次のページに続く)


```

    },
    "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 only events of this severity and higher will be tracked 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"
        }
    }
}

```

そして `stdout` のハンドラに関するセクションにフィルターを追加します:

```

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

```

フィルターは単なる関数なので、`filter_maker` (ファクトリ関数) は以下のように定義することができます:

```

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

    def filter(record):

```

```

    return record.levelno <= level

return filter

```

この関数は引数として与えられた文字列をログレベルの数値に変換し、ログレコード内のログレベルが指定されたレベル以下の場合に `True` を返す関数を返します。この例では `filter_maker` をコマンドラインから実行するテストスクリプト `main.py` で関数が定義されているため、フィルター設定上のモジュール名は `__main__` になります。そのためフィルター設定の中では `__main__.filter_maker` となります。異なるモジュールで関数を定義した場合、この部分を変更する必要があります。

フィルターを追加した `main.py` の全体像は以下のようになり、これで実行できる状態になりました:

```

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": {
            "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')
```

これを以下のように実行します:

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

すると期待通りの結果を得ることができます:

```
$ 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
::::::::::::
```

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

6 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.warning('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')

```

7 ブロックする handler を扱う

ときどき、logging を行っているスレッドをブロックせずに、handler が動くようにしないといけないときがあります。これは Web アプリケーションではよくあることですし、もちろん他のシナリオでも起きる話です。

動作が鈍くなるときの元凶はたいてい、開発者のコントロール外にあるいくつかの理由で (例えば、残念なパフォーマンスのメールやネットワークのインフラ)、SMTPHandler: が電子メールを送るのに時間がかかることです。しかし、ほとんどのネットワークをまたぐ handler はブロックする可能性があります: SocketHandler による処理でさえ、裏で DNS への問い合わせというとても遅い処理を行うことがあります (そしてこの問い合わせ処理は、Python の層より下のあなたの手の届かない、ソケットライブラリの深いところにある可能性もあります)。

解決策の 1 つは、2 パートに分離したアプローチを用いることです。最初のパートは、パフォーマンスが重要なスレッドからアクセスされる、QueueHandler だけをアタッチした logger です。この logger は単に、十分大きい、あるいは無制限の容量を持ったキューに書き込むだけです。キューへの書き込みは通常すぐに完了しますが、念の為に queue.Full 例外をキャッチする必要があるかもしれません。もしパフォーマンスクリティカルなスレッドを持つライブラリの開発者であるなら、このことを (QueueHandler だけをアタッチした logger についての言及を添えて) ドキュメントに書いておきましょう。

2 つ目のパートは QueueHandler の対向として作られた QueueListener です。QueueListener はとてもシンプルで、キューと handler を渡され、内部で QueueHandler (もしくは他の LogRecord の出力元) から送られた LogRecord をキューから受け取るスレッドを起動します。LogRecord をキューから取り出して、handler に渡して処理させます。

分離した QueueListener クラスを持つメリットは、複数の QueueHandler に対して 1 つのインスタンスで logging できることです。既存の handler のスレッド利用版を使って handler ごとにスレッドを持つよりはずっとリソースにやさしくなります。

この 2 つのクラスを利用する例です (import は省略):

```

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()
```

実行すると次のように出力します:

```
MainThread: Look out!
```

注釈

上述の議論は特に同期コードについてのものではなく、むしろ遅いロギングハンドラについてでしたが、非同期コードやネットワーク、あるいはファイルハンドラからのロギングでさえも（イベントループをブロックしてしまう）問題につながる可能性があることには注意すべきでしょう。これはいくつかのロギングが `asyncio` 内部で行われることが理由です。何らかの非同期コードがアプリケーションの中で使われている場合、ロギングには上記のアプローチを使っていかなるブロッキングコードも `QueueListener` スレッド内だけで実行されるようにしておくことが最も良いでしょう。

バージョン 3.5 で変更: Python 3.5 以前は、`QueueListener` クラスは常にキューから受け取ったメッセージを、初期化元となっているそれぞれのハンドラーに受け渡していました。（というのも、レベルフィルターリングは別サイド、つまり、キューが満たされている場所で処理されるということが想定されているからです）Python 3.5 以降では、キーワードとなる引数 `respect_handler_level=True` をリスナーのコントラクターに受け渡すことで、この挙動を変更することができるようになっています。これが行われると、各メッセージのレベルをハンドラーのレベルと比較して、そうすることが適切な場合のみ、メッセージをハンドラーに渡します。

8 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端の 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,
        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 = 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()

```

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:

```
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.
```

注意: `pickle` にはいくつかのシナリオでセキュリティ上の問題があります。これらが影響する場合は、`makePickle()` メソッドをオーバーライドし代替シリアライズ手法を実装して、それを使うように上記のスク립トを修正することもできます。

8.1 作成中のロギングソケットのリスナーを実行する

作成中のロギングリスナーを実行するためには、`Supervisor` のようなプロセス管理ツールを使う必要があるかもしれません。こちらの [Gist](#) は `Supervisor` を使って上記の機能を実行するための必要最低限のファイルを提供しています。以下のファイルが必要になります:

ファイル	目的
<code>prepare.sh</code>	試験用の環境を準備する Bash スクリプト
<code>supervisor.conf</code>	リスナーとマルチプロセスの web アプリケーションのための設定を含む <code>Supervisor</code> の設定ファイル
<code>ensure_app.sh</code>	<code>Supervisor</code> が上記の設定で実行されていることを保証するための Bash スクリプト
<code>log_listener.py</code>	ログイベントを受信してファイルに記録するソケットリスナープログラム
<code>main.py</code>	リスナーに接続されたソケットを通じてロギングを実行する簡単な web アプリケーション
<code>webapp.json</code>	web アプリケーションのための JSON 設定ファイル
<code>client.py</code>	web アプリケーションを起動するための Python スクリプト

この web アプリケーションは、リクエストを処理する複数のワーカプロセスを起動する web アプリケーションサーバーである `Gunicorn` を使っています。ここに例として挙げた設定は、複数のワーカーがいかにして互いに衝突することなく同じログファイルに書き込みを行うことができるかを示しています --- ワーカーは全てソケットリスナーを通じてログを書き込むのです。

これらのファイルを試すためには、POSIX 環境において以下を行ってください:

1. `Download ZIP` ボタンを押して [Gist](#) を ZIP アーカイブとしてダウンロードしてください。
2. アーカイブファイルをスクラッチディレクトリに展開してください。
3. 準備のために、スクラッチディレクトリにおいて `bash prepare.sh` を実行してください。これにより `Supervisor` 関連のファイルおよびログファイルのための `run` サブディレクトリ、および `bottle`,

gunicorn そして supervisor がインストールされる仮想環境を含む venv サブディレクトリが生成されます。

4. `bash ensure_app.sh` を実行して Supervisor が上記の設定で実行されていることを確認してください。
5. `venv/bin/python client.py` を実行して web アプリケーションを起動してください。これによりログにレコードが書き込まれるはずです。
6. `run` サブディレクトリにあるログファイルを調べてください。最新のログは、パターン `app.log*` に一致する名前のファイルにあるはずです。ログは異なるワーカプロセスによって非決定論的な形で並行に処理されるため、特定の決まった順番にはなりません。
7. リスナーと web アプリケーションは `venv/bin/supervisorctl -c supervisor.conf shutdown` を実行することでシャットダウンできます。

ありそうもないことですが、テスト環境で設定したポートが別の設定と衝突してしまった場合、設定ファイルを修正する必要があるかもしれません。

9 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。

9.1 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
```

(次のページに続く)

```

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なることです。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。

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__ を実装していて 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。

9.2 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なります。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.
    """

    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')
```

実行すると、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
↳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 contextvars の利用

Python 3.7 以降、contextvars モジュールは threading と asyncio の両方の処理上のニーズを満たすコンテキストローカルな変数を提供しています。このタイプの変数はスレッドローカルな変数に適しています。以下の例は、マルチスレッド環境において、いかにして web アプリケーションによって処理されるリクエスト属性のような処理の文脈上の情報とともにログを投入できるかを示します。

説明のため、同じ Python プロセス上で共通のライブラリを使っている独立した複数の web アプリケーションがあるとします。これらのアプリケーションが、クライアントの IP アドレスや HTTP リクエストメソッド、およびクライアントのユーザー名のようなコンテキスト依存の情報を追加情報として含んだ自身のログを、共通のライブラリ（と他のリクエスト処理のためのコード）から各アプリケーションのログファイルへ適切に振り向けるためにはどのようにしたら良いのでしょうか？

ここで、ライブラリは以下のコードで模倣することができますとします：

```

# 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!')

```

(次のページに続く)

```
# Just sleep for a bit so other threads get to run
time.sleep(0.01)
```

複数の web アプリケーションは 2 つの簡単なクラス, Request と WebApp, によって模倣することができます。これらは実際のスレッド化された web アプリケーションがどのように動作するかを模倣的にあらわしています - すなわち各リクエストはスレッドで処理される状況です:

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

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.ArgumentParserDefaultsHelpFormatter

```

```

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)
        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()

```

上記のコードを実行すると、およそ半分のリクエストが `app1.log` に、残り半分が `app2.log` にそれぞれ記録され、同時に全てのリクエストが `app.log` に記録されることがわかるでしょう。それぞれの web アプリケーション固有のログはそれぞれのアプリケーションからのログエントリだけを含み、リクエスト情報が矛盾なくログに表示されている（すなわち、各ダミーリクエストの情報は常にログの行とともに現れる）はずです。このことは以下のシェルコマンドの出力により例示されています：


```

~/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
~/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 ハンドラ内でのコンテキスト情報の付与

Handler はそれぞれ一連のフィルタを独自に持っています。コンテキスト情報を、他のハンドラに漏らすことなく LogRecord に付加したい場合、以下に示すスクリプトのように、直接レコードを編集する代わりに新しい LogRecord を返すフィルタを使うことができます:

```

import copy
import logging

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

```

(次のページに続く)

```

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 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。

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)。

別の方法として、`Queue` と `QueueHandler` を使って、マルチプロセスアプリケーションの 1 つのプロセスに全ての `logging` イベントを送る事ができます。次の例はこれを行う方法を示します。この例では独立した listener プロセスが他のプロセスから送られた event を受け取り、それを独自の `logging` 設定にしたがって保存します。この例は実装の方法の 1 つを示しているだけ (例えば、listener プロセスを分離する代わりに listener スレッドを使うこともできます) ですが、これは listener とアプリケーション内の他のプロセスで完全に異なる設定を使う例になっているので、各自の要求に応じたコードを書く叩き台になるでしょう:

```

# 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',
    '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()

```

上のスクリプトの亜種で、logging をメインプロセスの別スレッドで行う例:

```

import logging
import logging.config

```

```

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

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

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

if __name__ == '__main__':
    q = Queue()
    d = {
        'version': 1,
        'formatters': {
            'detailed': {
                'class': 'logging.Formatter',
                'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-10s
↳ %(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': {
            '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()

```

こちらは特定の logger に設定を適用する例になっています。foo logger は foo サブシステムの中の全てのイベントを mplog-foo.log に保存する特別な handler を持っています。これはメインプロセス側の log 処理で使われ、(logging イベントは worker プロセス側で生成されますが) 各メッセージを適切な出力先に出力します。

12.1 concurrent.futures.ProcessPoolExecutor の利用

もし、ワーカースタンプを起動するために `concurrent.futures.ProcessPoolExecutor` を使いたいのであれば、少し違う方法でキューを作る必要があります。次のような方法の代わりに

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

次のコードを利用する必要があります

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

そして、次のようなワーカー作成コードがあったとすると:

```
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()
```

次のように変更します (最初に `concurrent.futures` をインポートするのを忘れないようにしましょう):

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

12.2 Gunicorn と uWSGI を用いた Web アプリケーションのデプロイ

Gunicorn や uWSGI (またはそれに類するもの) を使って Web アプリケーションをデプロイする場合、クライアントのリクエストを処理するために複数のワーカースタンプが作られます。そのような環境では、web アプリケーションに直接ファイルベースのハンドラを作ることは避けてください。代わりに、`SocketHandler` を使って別のプロセスとして動作するリスナーに web アプリケーションからログを送信するようにしてください。これは Supervisor のようなプロセス管理ツールを使うことで設定できます - 詳しくは [作成中のログインソケットのリスナーを実行する](#) を参照してください。

13 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。

14 別の format スタイルを利用する

logging が Python 標準ライブラリに追加された時、メッセージを動的な内容でフォーマットする唯一の方法は % を使ったフォーマットでした。その後、Python には新しい文字列フォーマット機構として `string.Template` (Python 2.4 で追加) と `str.format()` (Python 2.6 で追加) が加わりました。

logging は (3.2 から) この 2 つの追加されたフォーマット方法をサポートしています。Formatter クラスが `style` という名前のオプションのキーワード引数を受け取ります。このデフォルト値は '%' ですが、他に '{' と '\$' が指定可能で、それぞれのフォーマット方法に対応しています。後方互換性はデフォルト値によって維持されていますが、明示的に `style` 引数を指定することで、`str.format()` か `string.Template` を使った `format` を指定する事ができます。次の例はこの機能を使っています:

```
>>> 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
>>>
```

最終的に出力されるログメッセージの `format` と、各メッセージを生成する部分は完全に独立していることに注意してください。次の例でわかるように、メッセージを生成する部分では % を使い続けています:

```
>>> 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 の呼び出し (`logger.debug()` や `logger.info()` など) は、ログメッセージのために位置引数しか受け取らず、キーワード引数はそれを処理するときのオプションを指定するためだけに使われます。(例えば、`exc_info` キーワード引数を使ってログを取るトレースバック情報を指定したり、`extra` キーワード引数を使ってログに付与する追加のコンテキスト情報を指定します。) logging パッケージは内部で % を使って `format` 文字列と引数をマージしているので、`str.format()` や `string.Template` を使って logging を呼び出す事はできません。既存の logging 呼び出しは %-format を使っているため、後方互換性のためにこの部分を変更することはできません。

しかし、{} や \$ を使ってログメッセージをフォーマットする方法はあります。ログメッセージには任意のオ

オブジェクトを format 文字列として渡すことができ、logging パッケージはそのオブジェクトに対して `str()` を使って実際の format 文字列を生成します。次の 2 つのクラスを見てください:

```
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)
```

どちらのクラスも format 文字列の代わりに利用して、`{}` や `$` を使って実際のログの `"%(message)s"`, `"{message}"`, `"$message"` で指定された `"message"` 部分を生成することができます。これは何かログを取りたいときに常に使うには使いにくいクラス名ですが、`__` (アンダースコア 2 つ --- `gettext.gettext()` やその仲間によくエイリアスとして使われるアンダースコア 1 つと混同しないように) などの使いやすいエイリアスを使うことができます。

上のクラスは Python には含まれませんが、自分のコードにコピペして使うのは簡単です。これらは次の例のようにして利用できます。(上のクラスが `wherever` というモジュールで定義されていると仮定します):

```
>>> 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
>>>
```

上の例ではフォーマットの動作を示すために `print()` を使っていますが、もちろん実際にこの方法でログを出力するには `logger.debug()` などを使います。

注意点として、この方法には大きなパフォーマンスのペナルティはありません。実際のフォーマット操作は logging の呼び出し時ではなくて、メッセージが実際に handler によって出力されるときに起こります。(出

力されないならフォーマットもされません) そのため、この方法で注意しないといけないのは、追加の括弧が書式文字列だけではなく引数も囲わないといけないことです。これは `__` が `XXXMessage` クラスのコンストラクタ呼び出しのシンタックスシュガーでしか無いからです。

次の例のように、`LoggerAdapter` を利用して上と似たような効果を実現する方法もあります:

```
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 log(self, level, msg, /, *args, stacklevel=1, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger.log(level, Message(msg, args), **kwargs,
                           stacklevel=stacklevel+1)

logger = StyleAdapter(logging.getLogger(__name__))

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

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

上のスクリプトは Python 3.8 以降では `Hello, world!` というメッセージをログするはずです。

15 LogRecord のカスタマイズ

全ての logging イベントは `LogRecord` のインスタンスとして表現されます。イベントのログが取られて logger のレベルでフィルタされなかった場合、イベントの情報を含む `LogRecord` が生成され、その logger (と、propagate が無効になるまでの上位 logger) の handler に渡されます。Python 3.2 までは、この生成が行われているのは 2 箇所だけでした:

- `Logger.makeRecord()`: 通常のイベント logging プロセスで呼ばれます。これは `LogRecord` を直接読んでインスタンスを生成します。
- `makeLogRecord()`: `LogRecord` に追加される属性を含む辞書を渡して呼び出されます。これはネットワーク越しに (pickle 形式で `SocketHandler` 経由で、あるいは JSON 形式で `HTTPHandler` 経由で) 辞書を受け取った場合などに利用されます。

そのために `LogRecord` で何か特別なことをしたい場合は、次のどちらかをしなければなりません。

- `Logger.makeRecord()` をオーバーライドした独自の `Logger` のサブクラスを作り、利用したい logger のどれかがインスタンス化される前に、それを `setLoggerClass()` を使って登録する。
- `filter()` メソッドで必要な特殊な処理を行う `Filter` を logger か handler に追加する。

最初の方法は、複数の異なるライブラリが別々のことをしようとした場合にうまく行きません。各ライブラリが独自の `Logger` のサブクラスを登録しようとして、最後に登録されたライブラリが生き残ります。

2つ目の方法はほとんどのケースでうまくいきますが、たとえば `LogRecord` を特殊化したサブクラスを使うことなどはできません。ライブラリの開発者は利用している logger に適切な filter を設定できますが、新しい logger を作るたびに忘れずに設定しないといけなくなります。(新しいパッケージやモジュールを追加し、モジュールレベルで次の式を実行することで、新しい logger が作られます)

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

これでは考えることが余計に1つ増えてしまうでしょう。開発者は、最も高いレベルのロガーに取り付けられた `NullHandler` に、フィルタを追加することもできますが、アプリケーション開発者が、より低いレベルに対するライブラリのロガーにハンドラを取り付けた場合、フィルタは呼び出されません --- 従って、そのハンドラからの出力はライブラリ開発者の意図を反映したものにはなりません。

Python 3.2 以降では、`LogRecord` の生成は、指定できるファクトリ経由になります。ファクトリは callable で、`setLogRecordFactory()` で登録でき、`getLogRecordFactory()` で現在のファクトリを取得できます。ファクトリは `LogRecord` コンストラクタと同じシグネチャで呼び出され、`LogRecord` がデフォルトのファクトリとして設定されています。

このアプローチはカスタムのファクトリが `LogRecord` の生成のすべての面を制御できるようにしています。たとえば、サブクラスを返したり、次のようなパターンを使って単に属性を追加することができます:

```
old_factory = logging.getLogRecordFactory()

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

logging.setLogRecordFactory(record_factory)
```

このアプローチでは複数の異なるライブラリがファクトリをチェーンさせて、お互いに同じ属性を上書きしたり、標準で提供されている属性を意図せず上書きしない限りはうまくいきます。しかし、チェーンのつながり全てが、全ての logging の操作に対しての実行時のオーバーヘッドになることを念頭に置き、このテクニックは `Filter` を利用するだけでは望む結果が得られない場合にのみ使うべきです。

16 Subclassing QueueHandler and QueueListener- a ZeroMQ example

16.1 Subclass QueueHandler

QueueHandler のサブクラスを作ってメッセージを他のキュー、例えば ZeroMQ の 'publish' ソケットに送信することができます。下の例では、ソケットを別に作ってそれを handler に ('queue' として) 渡します:

```
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)
```

もちろん同じことを別の設計でもできます。socket を作るのに必要な情報を handler に渡す例です:

```
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()
```

16.2 Subclass QueueListener

QueueListener のサブクラスを作って、メッセージを他のキュー、例えば ZeroMQ の 'subscribe' ソケットから取得する事もできます。サンプルです:

```
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)
```

(次のページに続く)

```

    super().__init__(socket, *handlers, **kwargs)

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

```

17 Subclassing QueueHandler and QueueListener- a pynng example

In a similar way to the above section, we can implement a listener and handler using `pynng`, which is a Python binding to `NNG`, billed as a spiritual successor to `ZeroMQ`. The following snippets illustrate -- you can test them in an environment which has `pynng` installed. Just for variety, we present the listener first.

17.1 Subclass QueueListener

```

# listener.py
import json
import logging
import logging.handlers

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

interrupted = False

class NNGSocketListener(logging.handlers.QueueListener):

    def __init__(self, uri, /, *handlers, **kwargs):
        # Have a timeout for interruptability, and open a
        # subscriber socket
        socket = pynng.Sub0(listen=uri, recv_timeout=500)
        # The b'' subscription matches all topics
        topics = kwargs.pop('topics', None) or b''
        socket.subscribe(topics)
        # We treat the socket as a queue
        super().__init__(socket, *handlers, **kwargs)

    def dequeue(self, block):
        data = None
        # Keep looping while not interrupted and no data received over the
        # socket
        while not interrupted:
            try:
                data = self.queue.recv(block=block)
                break

```

```

        except pynng.Timeout:
            pass
        except pynng.Closed: # sometimes happens when you hit Ctrl-C
            break
    if data is None:
        return None
    # Get the logging event sent from a publisher
    event = json.loads(data.decode('utf-8'))
    return logging.makeLogRecord(event)

def enqueue_sentinel(self):
    # Not used in this implementation, as the socket isn't really a
    # queue
    pass

logging.getLogger('pynng').propagate = False
listener = NNGSocketListener(DEFAULT_ADDR, logging.StreamHandler(), topics=b'')
listener.start()
print('Press Ctrl-C to stop.')
try:
    while True:
        pass
except KeyboardInterrupt:
    interrupted = True
finally:
    listener.stop()

```

17.2 Subclass QueueHandler

```

# sender.py
import json
import logging
import logging.handlers
import time
import random

import pynng

DEFAULT_ADDR = "tcp://localhost:13232"

class NNGSocketHandler(logging.handlers.QueueHandler):

    def __init__(self, uri):
        socket = pynng.Pub0(dial=uri, send_timeout=500)
        super().__init__(socket)

    def enqueue(self, record):
        # Send the record as UTF-8 encoded JSON
        d = dict(record.__dict__)

```

```

        data = json.dumps(d)
        self.queue.send(data.encode('utf-8'))

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

logging.getLogger('pyng').propagate = False
handler = NNGSocketHandler(DEFAULT_ADDR)
# Make sure the process ID is in the output
logging.basicConfig(level=logging.DEBUG,
                    handlers=[logging.StreamHandler(), handler],
                    format='%(levelname)-8s %(name)10s %(process)6s %(message)s')
levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)
logger_names = ('myapp', 'myapp.lib1', 'myapp.lib2')
msgno = 1
while True:
    # Just randomly select some loggers and levels and log away
    level = random.choice(levels)
    logger = logging.getLogger(random.choice(logger_names))
    logger.log(level, 'Message no. %5d' % msgno)
    msgno += 1
    delay = random.random() * 2 + 0.5
    time.sleep(delay)

```

You can run the above two snippets in separate command shells. If we run the listener in one shell and run the sender in two separate shells, we should see something like the following. In the first sender shell:

```

$ python sender.py
DEBUG      myapp      613 Message no.      1
WARNING    myapp.lib2  613 Message no.      2
CRITICAL   myapp.lib2  613 Message no.      3
WARNING    myapp.lib2  613 Message no.      4
CRITICAL   myapp.lib1  613 Message no.      5
DEBUG      myapp      613 Message no.      6
CRITICAL   myapp.lib1  613 Message no.      7
INFO       myapp.lib1  613 Message no.      8

```

(and so on)

In the second sender shell:

```

$ python sender.py
INFO       myapp.lib2  657 Message no.      1
CRITICAL   myapp.lib2  657 Message no.      2
CRITICAL    myapp      657 Message no.      3
CRITICAL   myapp.lib1  657 Message no.      4
INFO       myapp.lib1  657 Message no.      5
WARNING    myapp.lib2  657 Message no.      6
CRITICAL    myapp      657 Message no.      7
DEBUG      myapp.lib1  657 Message no.      8

```

(and so on)

In the listener shell:

```
$ python listener.py
Press Ctrl-C to stop.
DEBUG      myapp      613 Message no.      1
WARNING    myapp.lib2  613 Message no.      2
INFO       myapp.lib2  657 Message no.      1
CRITICAL   myapp.lib2  613 Message no.      3
CRITICAL   myapp.lib2  657 Message no.      2
CRITICAL   myapp      657 Message no.      3
WARNING    myapp.lib2  613 Message no.      4
CRITICAL   myapp.lib1  613 Message no.      5
CRITICAL   myapp.lib1  657 Message no.      4
INFO       myapp.lib1  657 Message no.      5
DEBUG      myapp      613 Message no.      6
WARNING    myapp.lib2  657 Message no.      6
CRITICAL   myapp      657 Message no.      7
CRITICAL   myapp.lib1  613 Message no.      7
INFO       myapp.lib1  613 Message no.      8
DEBUG      myapp.lib1  657 Message no.      8
(and so on)
```

As you can see, the logging from the two sender processes is interleaved in the listener's output.

18 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': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d} {message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {message}',
            'style': '{',
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        },
    },
    'handlers': {
        'console': {
```

(次のページに続く)

(前のページからの続き)

```
        'level': 'INFO',
        'class': 'logging.StreamHandler',
        'formatter': 'simple',
    },
    'mail_admins': {
        'level': 'ERROR',
        'class': 'django.utils.log.AdminEmailHandler',
        'filters': ['special']
    }
},
'loggers': {
    'django': {
        'handlers': ['console'],
        'propagate': True,
    },
    'django.request': {
        'handlers': ['mail_admins'],
        'level': 'ERROR',
        'propagate': False,
    },
    'myproject.custom': {
        'handlers': ['console', 'mail_admins'],
        'level': 'INFO',
        '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。

19 rotator と namer を使ってログローテートをカスタマイズする

namer と rotator を定義する方法の例は以下の実行可能なスクリプトに示されています。ここではログファイルを gzip により圧縮する例を示しています:

```
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)
```

(次のページに続く)

(前のページからの続き)

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

このスクリプトを実行すると、6つの新しいファイルが生成され、そのうち5つは圧縮されています:

```
$ 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 より手の込んだ multiprocessing の例

次の実際に動作する例は、logging を multiprocessing と設定ファイルを使って利用する方法を示しています。設定内容はかなりシンプルですが、より複雑な構成を実際の multiprocessing を利用するシナリオで実装する方法を示しています。

この例では、メインプロセスは listener プロセスといくつかのワーカープロセスを起動します。メイン、listener、ワーカープロセスはそれぞれ分離した設定を持っています (ワーカープロセス群は同じ設定を共有します)。この例から、メインプロセスの logging、ワーカーが QueueHandler でログを送っているところ、listener が利用する QueueListener の実装、複雑な設定、キューから受け取ったイベントを設定された handler に分配する部分を見ることができます。この設定は説明用のものですが、この例を自分のシナリオに適応させることができるでしょう。

これがそのスクリプトです。docstring とコメントで動作を説明しています:

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

(次のページに続く)

```

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):
    """
    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 = {
        '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',
            '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 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ptional なピュア 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 3.1 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ピュア 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 3.2.4 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、オプションのピュア 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:

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

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

2. ASCII セクションを好きなプ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プ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 は 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 は何もエラーを起しません、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。

22 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ライズする単純な方法です:

```
import json
import logging

class StructuredMessage:
```

(次のページに続く)


```

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:

```

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()

```

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。

23 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 shutil.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:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.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': {
        'handlers': ['file'],
    },
}
```

(次のページに続く)

```

        '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。

24 固有の書式化スタイルをアプリケーション全体で使う

Python 3.2 では、`Formatter` クラスが `style` という名前のオプションのキーワード引数を受け取ります。このデフォルト値は後方互換性を維持するために `%` となっていますが、`{` か `$` を指定することで、`str.format()` か `string.Template` でサポートされているのと同じ書式化のアプローチを採れます。これは最終的に出力されるログメッセージの書式化に影響を与えますが、個々のログメッセージが構築される方法とは完全に直交していることに注意してください。

ロギングの呼び出し (`debug()`, `info()` など) はログメッセージのためには位置引数しか取らず、ロギングの呼び出しを処理する方法を決定するためだけにキーワード引数を指定できます (たとえば `exc_info` キーワード引数はトレースバックの情報をログに含めるかどうかを指定し、`extra` キーワード引数はログに付加する追加のコンテキスト情報を指定します)。ロギングパッケージは内部で `%-format` を使って書式文字列に可変長引数を埋め込んでいるため、`str.format()` や `string.Template` の構文を使って直接ロギングの呼び出しを行うことはできません。既存のコードにおける全てのロギングの呼び出しは `%-format` 文字列を使っているため、後方互換性を維持する限りこの部分に変更されることはないでしょう。

特定のロガーに関連付ける書式スタイルへの提案がなされてきましたが、そのアプローチは同時に後方互換性の問題にぶち当たります。あらゆる既存のコードはロガーの名前を使っているでしょうし、`%` 形式書式化を使っているでしょう。

あらゆるサードパーティのライブラリ、あらゆるあなたのコードの間で相互運用可能なようにロギングを行うには、書式化についての決定は、個々のログ呼び出しのレベルで行う必要があります。これは受け入れ可能な代替書式化スタイルに様々な手段の可能性を広げます。

24.1 LogRecord ファクトリを使う

Python 3.2 において、上述した `Formatter` の変更とともに、`setLogRecordFactory()` 関数を使って `LogRecord` のサブクラスをユーザに指定することを可能にするロギングパッケージの機能拡張がありました。これにより、`getMessage()` をオーバーライドして `だ だ し い` ことをする、あなた自身の手による `LogRecord` のサブクラスをセットすることが出来ます。このメソッドの実装は基底クラスでは `msg % args` 書式化をし、あなたの代替の書式化の置換が出来る場所ですが、他のコードとの相互運用性を保障するために、全ての書式化スタイルをサポートするよう注意深く行うべきであり、また、`%`-書式化をデフォルトで認めるべきです。基底クラスの実装がそうしているように、`str(self.msg)` 呼び出しもしてください。

さらに詳しい情報は、リファレンスの `setLogRecordFactory()`, `LogRecord` を参照してください。

24.2 カスタムなメッセージオブジェクトを使う

あなた独自のログメッセージを構築するのに `{}`- および `$`- 書式化を使えるようにするための、もうひとつの、おそらくもっと簡単な方法があります。ロギングの際には、あなたはメッセージ書式文字列として、任意のオブジェクトを使えることを (arbitrary-object-messages より) 思い出してみましょう、そしてロギングパッケージはそのオブジェクトに対して実際の書式文字列を得るために `str()` を呼び出すことも。以下 2 つのクラスを検討してみましょう：

```
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)
```

どちらのクラスも `format` 文字列の代わりに利用して、`{}` や `$` を使って実際のログの “%(message)s” , “{message}” , “\$message” で指定された “message” 部分を生成することができます。これは何かログを取りたいときに常に使うには使いにくいクラス名ですが、使いやすいうようにエイリアスを作れば良いでしょう、`M` であるとか `_` のような (あるいは地域化のために既に `_` を使っているのであれば `__` が良いかもしれません)。

このアプローチによる例をお見せします。最初は `str.format()` を使ってフォーマットする例です:

```
>>> __ = 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)
```

2 つめは `string.Template` でフォーマットする例です:

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

注意点として、この方法には大きなパフォーマンスのペナルティはありません。実際のフォーマット操作は logging の呼び出し時ではなくて、メッセージが実際に handler によって出力されるときに起こります。(出力されないならフォーマットもされません) そのため、この方法で注意しないといけないのは、追加の括弧が書式文字列だけではなく引数も囲わないといけないことだけです。これは上記のとおり `__` が `XXXMessage` クラスのコンストラクタ呼び出しのシンタックスシュガーでしか無いからです。

25 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か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:
```

(次のページに続く)

```

        allow = self.param not in record.msg
    if allow:
        record.msg = 'changed: ' + record.msg
    return allow

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

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

```

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。

26 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グ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().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)

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: division by zero|'Traceback (most recent call
↳last):\n  File "logtest7.py", line 30, in main\n    x = 1 / 0\nZeroDivisionError: division
↳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。

27 ログメッセージを喋る

ロ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くとも、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。

28 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

メッセージを一次領域に記録し、ある種の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 (バッファされたレコード数) 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)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
```

(次のページに続く)

```

        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

```

(前のページからの続き)

```
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
```

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):
    ...
```

29 バッファリングしながらロギングメッセージを email で送信する

ログメッセージをメールで、特に 1 つのメールにつき複数のログメッセージを、送信する方法を例示するため、`BufferingHandler` を継承します。以下の例は、必要に応じて改変することもできますが、SMTP 経由でログを送信するのに必要な情報をコマンドライン引数で指定してスクリプトを実行できるように簡単なテストハネスも提供しています (必須の引数およびオプション引数の詳細を見るためには、ダウンロードしたスクリプトを `-h` 引数をつけて実行してください)。

```
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
```

(次のページに続く)

```

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')
    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()

```

SMTP サーバーを正しく設定した上でスクリプトを実行すると、指定したアドレス宛てに 11 通のメールを受け取るでしょう。最初の 10 通のメールはそれぞれ 10 個のログメッセージを含み、11 通目のメールは 2 つのログメッセージを含むはずです。これらのログメッセージはスクリプト内で指定された 102 個のログメッセージから構成されています。

30 設定によって時刻を UTC(GMT) で書式化する

時刻を UTC でフォーマットしたい場合もあるでしょう。以下に示すように、そのようなフォーマット処理は `UTCFormatter` のようなクラスを使って行うことができます:

```
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できます:

```
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__':
    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トを与えています。

31 ロギングの選択にコンテキストマネージャを使う

一時的にロギングの設定を変えて、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、ロギング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、任意にロギングレベルを変更し、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ギ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:
    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グ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グガーに追加され、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も構いません。

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.')
    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。

32 CLI アプリケーションスターターテンプレート

このサンプルでは次のことを説明します:

- コマンドライン引数に応じてログレベルを使用する
- 複数のファイルに分割されたサブコマンドにディスパッチする。すべて一貫して同じレベルでログ出力を行う
- シンプルで最小限の設定で行えるようにする

サービスを停止したり、開始したり、再起動する役割を持ったコマンドラインアプリケーションがあるとします。説明のために、アプリケーションのメインスクリプトが `app.py`、個々のコマンドが `start.py`、`stop.py`、`restart.py` に実装されているものとします。デフォルトは `logging.INFO` ですが、コマンドライン引数を使ってアプリケーションのログの冗長性を制御したいとします。`app.py` は次のコードのようになるでしょう:

```
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
```

(次のページに続く)

(前のページからの続き)

```
logging.basicConfig(level=options.log_level,
                    format='%(levelname)s %(name)s %(message)s')

cmd(options)

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

start、stop、restart コマンドは個別のモジュールとして実装されます。次は起動コマンドのソースです:

```
# start.py
import logging

logger = logging.getLogger(__name__)

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)
```

停止コマンドのソースは次の通りです:

```
# 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)
```

同様に、再起動のコマンドは次の通りです:

```
# restart.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
```

(次のページに続く)

(前のページからの続き)

```
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)
```

このアプリケーションをデフォルトのログレベルで実行すると、次のような出力が得られます:

```
$ 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.
```

最初のワードはログレベルで、次のワードはイベントのログ出力が行われたモジュールまたはパッケージ名です。

ログレベルを変更し、ログに出力する情報を変更できるようにしましょう。もし、より詳細な情報が必要だとしましょう:

```
$ 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.
```

あるいは情報を減らしたい場合もあるでしょう:

```
$ 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
```

この場合、コマンドはコンソールに何も出力しなくなります。

33 Qt GUI のログ出力

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.

次のサンプルは Qt GUI でログ出力を行うサンプルです。ここではシンプルな `QtHandler` クラスを作成しています。これは呼び出し可能オブジェクトを受け取ります。これは GUI 更新を行うメインスレッドの中で利用されるスロットです。ワーカースレッドも作成し、UI 自身からボタンを使ってログを出力したり、バックグラウンドのタスクを行うワーカースレッドからログ出力を行います（ここではランダムな期間にランダムなレベルでメッセージを出しています）。

ワーカースレッドは `threading` モジュールではなく、Qt の `QThread` クラスを使っています。これは他の Qt コンポーネントとうまく統合できるように、`QThread` を使う必要があるからです。

The code should work with recent releases of any of `PySide6`, `PyQt6`, `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 different Qt packages
try:
    from PySide6 import QtCore, QtGui, QtWidgets
    Signal = QtCore.Signal
    Slot = QtCore.Slot
except ImportError:
    try:
        from PyQt6 import QtCore, QtGui, QtWidgets
        Signal = QtCore.pyqtSignal
        Slot = QtCore.pyqtSlot
    except ImportError:
        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__)

#
# Signals need to be contained in a QObject or subclass in order to be correctly
# initialized.
#
```

(次のページに続く)

```

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
        # 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)
            try:
                if random.random() < 0.1:
                    raise ValueError('Exception raised: %d' % i)
                else:
                    level = random.choice(LEVELS)
                    logger.log(level, 'Message after delay of %3.1f: %d', delay, i,
↪extra=extra)
            except ValueError as e:
                logger.exception('Failed: %s', e, 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')
        if hasattr(f, 'Monospace'):
            f.setStyleHint(f.Monospace)
        else:
            f.setStyleHint(f.StyleHint.Monospace) # for Qt6

```

```

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)

# 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()
    if hasattr(app, 'exec'):
        rc = app.exec()
    else:
        rc = app.exec_()
    sys.exit(rc)

if __name__ == '__main__':
    main()

```


34 RFC5424 をサポートする syslog へのロギング

RFC 5424 は 2009 年から始まっていますが、ほとんどの syslog サーバーはデフォルトで、2001 年から使われている古い **RFC 3164** を使って構成されています。2003 年に logging モジュールが Python に追加されたとき、モジュールは古い (そして当時唯一存在した) プロトコルをサポートしていました。RFC5424 は、それが登場して以来、syslog サーバーにおいて広く使われることがなかったために、SysLogHandler の機能は更新されてきませんでした。

RFC 5424 は構造化データのサポートなど、いくつかの有用な機能を持っています。その機能をサポートする syslog サーバーへのロギングを可能にする必要がある場合、以下のような派生ハンドラクラスを使うことで実現することができます:

```
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'):
```

(次のページに続く)

```

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
# 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}'

```

上記のコードを完全に理解するには RFC 5424 を熟知する必要があります。また、上記の例とはやや異なる要求を持つこともあるでしょう（たとえば構造化データをログに渡す方法について）。にもかかわらず、上記のコードは特有の要求に対する順応性があります。上記のハンドラにより、構造化データは以下のように渡すことができるでしょう：

```

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 ロガーを出力ストリームのように取り扱う方法

書き込み先として file-like オブジェクトを期待するサードパーティの API に接続する必要がある一方で、その API の出力を直接ロガーに送りたいということがときどきあります。これは file-like な API でロガーをラップするクラスを使うことで実現できます。以下はそのようなクラスを例解する短いスクリプトです：

```

import logging

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

```

(次のページに続く)

```

        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
        # 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()

```

このスクリプトを実行すると、次のように出力されます。

```

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

```

また、`sys.stdout` や `sys.stderr` をリダイレクトするには `LoggerWriter` を使って以下のようにします:

```

import sys

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

```

上記の操作は、必要に応じてロギングを設定した **後** に 行うべきです。上記の例では、`basicConfig()` の呼び出しが (`LoggerWriter` インスタンスで上書きされる **前の** `sys.stderr` を使って) 設定を行います。そして、以下のような結果を得るでしょう:

```

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

```

言うまでもなく上記の例は `basicConfig()` で使われている書式にもとづく出力を示していますが、ロギングの設定で異なるフォーマッタを使うことができます。

上記の例では、バッファリングや奪取した書き込み呼び出しのシーケンスの扱いについてはなすがままになっています。たとえば、上記の `LoggerWriter` の定義で、次のようなコードの断片があったとします。

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

このスクリプトを実行すると以下のような結果が得られます。

```
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
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 slightly 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)
```

この実装は改行があらわれるまでログをバッファリングし、行全体をログに出力するだけです。このアプローチにより、より適切な出力が得られます:

```
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 避けるべきパターン

これまでのセクションではログ出力を行うときに必要なこと、考慮すべきことなどを説明してきました。このセクションでは、**役に立たない** 利用パターンについて触れます。これは多くの場合避けるべきことです。これらの説明はどこから読んでも構いません。

36.1 同じログファイルを何度も開く

Windows では同じファイルを何度も開くことができず「このファイルは他のプロセスから利用されています」というエラーが表示されます。しかし、POSIX プラットフォームではエラーがおきることなく、同じファイルを何度も開けます。これは次のように間違っ使用される可能性があります。

- 同じファイルを指すファイルハンドラを 1 度以上追加する（例えばコピー&ペーストして書き換え忘れによるエラー）。
- 異なる名前を持つ、一見異なる 2 つのファイルを開くが、片方が他方へのシンボリックリンクとなっている。
- プロセスをフォークするが、その後親プロセスと子プロセスが同じファイルへの参照を維持する。これは例えば、`multiprocessing` モジュールなどを使うと発生する可能性があります。

ファイルを複数回開くことは、**一見** 動作しているように見えますが、さまざまな問題を引き起こす可能性があります：

- 複数のスレッドが同一ファイルに書き出そうとすると、ログ出力が文字化けする可能性があります。ログモジュールは同じハンドラのインスタンスに対して並列で利用しても正しく動くようになっていますが、同じファイルを参照する 2 つの異なるハンドラのインスタンスに対し、2 つのスレッドから同時に書き込みをした場合にはそのような保護は働きません。
- (たとえばファイルのローテーションの間に) ファイルを削除しようとする、そのファイルを指す別の参照が残っているために、何のエラーも発しないまま失敗します。これは混乱や不要なデバッグの時間のもととなる可能性があります - ログが思いもしない場所に記録されたり、完全に失われたりします。もしくは移動したと思われていたファイルが残っていたり、ファイルサイズにもとづくローテーションが行われているにもかかわらずファイルサイズが予想外に増加したりすることもあります。

この問題を回避するには **複数のプロセスからの単一ファイルへのログ記録** で紹介したテクニックを使用してください。

36.2 ロガーをクラスの属性にするか、パラメータで渡す

ロガーはシングルトンであるため、一般的には意味がなく、これを行う必要が出てくることは滅多にありません。コードからは名前を使って `logging.getLogger(name)` 経由でロガーインスタンスにアクセスできるため、インスタンスを持って回って、インスタンス属性として保持することは意味がありません。Java や C# といった他の言語ではよく静的クラス属性にしています。しかし、Python においてはこのパターンはクラスではなくモジュールがソフトウェア分解の単位となっているため、無意味です。

36.3 ライブラリ内でロガーに `NullHandler` 以外のハンドラーを追加する

ハンドラーやフォーマッター、フィルターを追加してログ出力をカスタマイズするのはライブラリ開発者ではなく、アプリケーション開発者の責務です。もしあなたがライブラリのメンテナンスをしているのであれば、`NullHandler` インスタンス以外のロガーを追加してはいけない、ということを意味します。

36.4 大量のロガーを作成する

ロガーはシングルトンであり、スクリプトの実行中に解放されることがないため、大量のロガーを作成すると、メモリが解放されることなく消費されます。ファイルの処理単位やネットワーク接続単位でロガーを作るのではなく、**既存のメカニズム** を使ってコンテキスト依存の情報をログに渡し、ロガーはアプリケーション内の説明の単位（通常はモジュールだが、場合によってはそれよりも小さい可能性もある）で作るように制限してください。

37 その他のリソース

参考

`logging` モジュール

`logging` モジュールの API リファレンス。

`logging.config` モジュール

`logging` モジュールの環境設定 API です。

`logging.handlers` モジュール

`logging` モジュールに含まれる、便利なハンドラです。

基本チュートリアル

上級チュートリアル

索引

R

RFC

RFC 3164, [73](#)

RFC 5424, [48](#), [73](#)

RFC 5424#section-6, [48](#)