
Logging HOWTO

リリース 3.9.18

Guido van Rossum
and the Python development team

2月 06, 2024

目次

1	基本 logging チュートリアル	2
1.1	logging を使うとき	2
1.2	簡単な例	3
1.3	ファイルへの logging	4
1.4	複数のモジュールからのロギング	5
1.5	変数データのロギング	6
1.6	表示されるメッセージのフォーマットの変更	6
1.7	メッセージ内での日付と時刻の表示	7
1.8	次のステップ	7
2	上級ロギングチュートリアル	8
2.1	Logging Flow	9
2.2	ロガー	10
2.3	ハンドラ	11
2.4	フォーマッタ	12
2.5	ロギングの環境設定	12
2.6	環境設定が与えられないとどうなるか	16
2.7	ライブラリのためのロギングの設定	16
3	ロギングレベル	17
3.1	カスタムレベル	18
4	便利なハンドラ	18
5	ログ記録中に発生する例外	20

6	任意のオブジェクトをメッセージに使用する	20
7	最適化	20
	索引	22

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

1 基本 logging チュートリアル

logging は、あるソフトウェアが実行されているときに起こったイベントを追跡するための手段です。ソフトウェアの開発者は、特定のイベントが発生したことを示す logging の呼び出しをコードに加えます。イベントは、メッセージで記述され、これに変数データ (すなわち、イベントが起こる度に異なるかもしれないデータ) を加えることもできます。イベントには、開発者がそのイベントに定めた重要性も含まれます。重要性は、**レベル** (*level*) や **重大度** (*severity*) とも呼ばれます。

1.1 logging を使うとき

logging は、単純なロギングの用法に便利な関数群を提供しています。この中には、`debug()`、`info()`、`warning()`、`error()` および `critical()` があります。logging を使うべき時を決めるには、よくあるタスクに使う最適なツールを述べた、以下のテーブルを参照してください。

行いたいタスク	そのタスクに最適なツール
コマンドラインスクリプトやプログラムで普通に使う、コンソール出力の表示	<code>print()</code>
プログラムの通常の操作中に発生したイベントの報告 (例えば、状態の監視や障害の分析)	<code>logging.info()</code> (または、診断のための特に詳細な出力には <code>logging.debug()</code>)
特定のランタイムイベントに関わる警告の発行	その発行が避けられるもので、クライアントアプリケーションを修正してその警告を排除するべきなら <code>warnings.warn()</code> アプリケーションにできることはないが、それでもイベントを記録するべきなら <code>logging.warning()</code>
特定のランタイムイベントに関わるエラーの報告	例外の送出
例外の送出をしないエラーの抑制 (例えば、長期のサーバプロセス中のエラーハンドラ)	特定のエラーやアプリケーションドメインに応じて <code>logging.error()</code> 、 <code>logging.exception()</code> または <code>logging.critical()</code>

ロギング関数は、そのイベントのレベルや重大度から名前を付けられ、それが追跡に使われます。標準のレベルとその適用範囲は、以下に (重大度が増す順に) 記述されています:

レベル	いつ使うか
DEBUG	おもに問題を診断するときのみに関心があるような、詳細な情報。
INFO	想定された通りのことが起こったことの確認。
WARNING	想定外のことが起こった、または問題が近く起こりそうである (例えば、'disk space low') ことの表示。
ERROR	より重大な問題により、ソフトウェアがある機能を実行できないこと。
CRITICAL	プログラム自体が実行を続けられないことを表す、重大なエラー。

デフォルトのレベルは **WARNING** で、logging パッケージが他に設定されなければ、このレベル以上のイベントのみ追跡されます。

追跡されるイベントは、異なる方法で処理されます。追跡されたイベントを処理する最も単純な方法は、それをコンソールに表示することです。その他のよくある方法は、それをディスクファイルに書き出すことです。

1.2 簡単な例

ごく簡単な例は:

```
import logging
logging.warning('Watch out!') # will print a message to the console
logging.info('I told you so') # will not print anything
```

これらの行をスクリプトにタイプして実行すると、次のようにコンソールに出力されます:

```
WARNING:root:Watch out!
```

デフォルトのレベルは **WARNING** なので、**INFO** メッセージは現れません。表示されたメッセージには、レベルの表示と、ロギングの呼び出しで提供された、イベントの説明すなわち 'Watch out!' が含まれます。'root' の部分は今は気にしないでください。あとで説明します。実際の出力は、必要に応じてかなり柔軟に書式化できます。書式化操作もあとで説明します。

1.3 ファイルへの logging

logging イベントをファイルに記録するのは非常によくあるパターンなので次はこれを見て行きましょう。以下のサンプルを試すときは Python インタプリタを新しく起動して、上のセッションの続きにならないようにしてください:

```
import logging
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

バージョン 3.9 で変更: *encoding* 引数が追加されました。以前のバージョンあるいは無指定の Python では、エンコーディングには `open()` が使うデフォルト値が使われます。上の例に出てきていませんが、今は省略できる *errors* 引数は、エンコーディングの扱われ方を決定します。利用可能な値およびデフォルト値については、`open()` のドキュメントを参照してください。

そして、ファイルの中身を確認すると、ログメッセージが確認できます:

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö
```

この例はまた、追跡のしきい値となるロギングレベルを設定する方法も示しています。この例では、しきい値を `DEBUG` に設定しているので、全てのメッセージが表示されています。

次のようなコマンドラインオプションでログレベルを設定したいと考え:

```
--log=INFO
```

`--log` に渡されたパラメータの値を変数 *loglevel* に保存しているとしたら、

```
getattr(logging, loglevel.upper())
```

というコードを使い、`basicConfig()` の *level* 引数に渡すべき値が得られます。ユーザの入力値をすべてエラーチェックしたくなり、次の例のように実装することもあるでしょう:

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

`basicConfig()` は、`debug()` や `info()` などの呼び出しよりも **前** に呼び出さなければなりません。これは、一度限りの単純な設定機能を意図しているので、実際に作用するのは最初の呼び出しのみで、続く呼び出しの効果は `no-op` です。

上記のスクリプトを複数回実行すると、2 回目以降の実行によるメッセージは `example.log` に加えられます。以前の実行によるメッセージを記憶せず、実行ごとに新たに始めたいなら、上記の例での呼び出しを次のように変え、`filemode` 引数を指定する方法がとれます:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

出力は先ほどと同じになりますが、ログファイルは追記されなくなり、以前の実行によるメッセージは失われます。

1.4 複数のモジュールからのロギング

プログラムが複数のモジュールでできているなら、そのロギングをどのように構成するかは例はこちらです:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

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

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

`myapp.py` を実行すれば、`myapp.log` でログが確認できます:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

この `mylib.py` でのパターンは、複数のモジュールに一般化できます。なお、この簡単な使用パターンでは、ログファイルを見ることで、イベントの説明は見られますが、アプリケーションの **どこから** メッセージが来たのかを知ることはできません。メッセージの位置を追跡したいなら、このチュートリアルレベルを超えたドキュメントが

必要になります -- [上級ロギングチュートリアル](#) を参照してください。

1.5 変数データのロギング

変数データのログを取るには、イベント記述メッセージにフォーマット文字列を使い、引数に変数データを加えてください。例えば:

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

により、次のように表示されます:

```
WARNING:root:Look before you leap!
```

ご覧の通り、イベント記述メッセージに変数データを統合するために、古い、% スタイルの文字列フォーマットを使っています。これは後方互換性のためです。logging パッケージは、`str.format()` や `string.Template` のような新しいフォーマットオプションよりも先に生まれました。新しいフォーマットオプションはサポートされていますが、その探求はこのチュートリアルでは対象としません。詳細は `formatting-styles` を参照してください。

1.6 表示されるメッセージのフォーマットの変更

メッセージを表示するのに使われるフォーマットを変更するには、使いたいフォーマットを指定する必要があります:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

により、次のように表示されます:

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

ご覧の通り、先の例に現れた 'root' が消失しています。フォーマット文字列に含めることができるものの一覧は、`logrecord-attributes` のドキュメントから参照できますが、単純な用途では、必要なものは `levelname` (重大度)、`message` (変数データを含むイベント記述)、それともしかしたら、イベントがいつ起こったかという表示だけです。これは次の節で解説します。

1.7 メッセージ内での日付と時刻の表示

イベントの日付と時刻を表示するには、フォーマット文字列に '%(asctime)s' を置いてください:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

これは以下の様なフォーマットで表示されます:

```
2010-12-12 11:41:42,612 is when this event was logged.
```

デフォルトの日付と時間のフォーマットは、ISO8601 の規格で表示されます（上記の表示結果）。日付と時間のフォーマットを変更した場合は、以下の例の様に、“basicConfig”関数の *datefmt* 引数に指定する必要があります:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

これは次のように表示されます:

```
12/12/2010 11:46:36 AM is when this event was logged.
```

datefmt 引数のフォーマットは、`time.strftime()` でサポートされているものと同じです。

1.8 次のステップ

基本チュートリアルはこれで終わりです。あなたがロギングを使っていくためには、これで十分でしょう。logging パッケージが提供するものはもっとありますが、それを使いこなすためには、もうちょっと時間をかけて、以下のセクションを読む必要があります。その用意ができれば、好きな飲み物を持って、次に進みましょう。

ロギングを簡潔に行いたいなら、上記の例を使って、ロギングをあなたのスクリプトに組み込んでください。問題があったり理解出来ないことがあったら、`comp.lang.python` Usenet group (<https://groups.google.com/forum/#!forum/comp.lang.python> から利用できます) に質問を投稿してくだされば、そう遠くないうちに助けが得られるでしょう。

まだいますか？ もう少し上級の、踏み込んだチュートリアルを綴った、幾つかの節を読み続けることができます。その後で、`logging-cookbook` もご覧ください。

2 上級ロギングチュートリアル

logging ライブラリはモジュール方式のアプローチを取り、いくつかのカテゴリの部品を提供します。ロガー、ハンドラ、フィルタ、フォーマッタです。

- ロガーは、アプリケーションコードが直接使うインターフェースを公開します。
- ハンドラは、(ロガーによって生成された) ログ記録を適切な送信先に送ります。
- フィルタは、どのログ記録を出力するかを決定する、きめ細かい機能を提供します。
- フォーマッタは、ログ記録が最終的に出力されるレイアウトを指定します。

ログイベント情報は `LogRecord` インスタンスの形で、`logger`, `handler`, `filter`, `formatter` の間でやりとりされます。

ロギングは、`Logger` クラスのインスタンス (以下 **ロガー**) にメソッドを呼び出すことで実行されます。各インスタンスには名前があり、名前空間階層構造に、ドット (ピリオド) をセパレータとして、概念的に並べられています。例えば、`'scan'` という名前のロガーは、ロガー `'scan.text'`, `'scan.html'` および `'scan.pdf'` の親です。ロガー名は、何でも望むものにでき、ロギングされたメッセージが発生した場所を指し示します。

ロガーに名前をつけるときの良い習慣は、ロギングを使う各モジュールに、以下のように名付けられた、モジュールレベルロガーを使うことです:

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

これにより、ロガー名はパッケージ/モジュール階層をなぞり、ロガー名だけで、どこでイベントのログが取られたか、直感的に明らかになります。

ロガーの階層構造の根源は、ルートロガーと呼ばれます。それが、関数 `debug()`, `info()`, `warning()`, `error()` および `critical()` によって使われるロガーとなります。これらの関数は単に、ルートロガーの同名のメソッドを呼び出します。これらの関数とメソッドは、同じ署名をもっています。ルートロガーの名前は、ログ出力では `'root'` と表示されます。

もちろん、メッセージを異なる送信先に記録することも出来ます。このパッケージでは、ファイルへ、HTTP GET/POST 先へ、SMTP 経由で電子メールへ、汎用のソケットへ、キューへ、または Windows NT イベントログのような OS 毎のログ記録機構への、ログメッセージの書きこみがサポートされています。送信先は、`handler` クラスによって取り扱われます。組み込みのハンドラクラスでは満たせないような、特殊な要件があるなら、独自のログ送信先を生成できます。

デフォルトでは、どのロギングメッセージに対しても、送信先は設定されていません。チュートリアルの例のように、`basicConfig()` を使って、送信先 (コンソールやファイルなど) を指定できます。関数 `debug()`, `info()`, `warning()`, `error()` および `critical()` を呼び出すと、それらは送信先が設定されていないかを調べます。そして設定されていなければ、ルートロガーに委譲して実際にメッセージを出力する前に、コンソール (`sys.stderr`) を送信先に、デフォルトのフォーマットを表示されるメッセージに設定します。

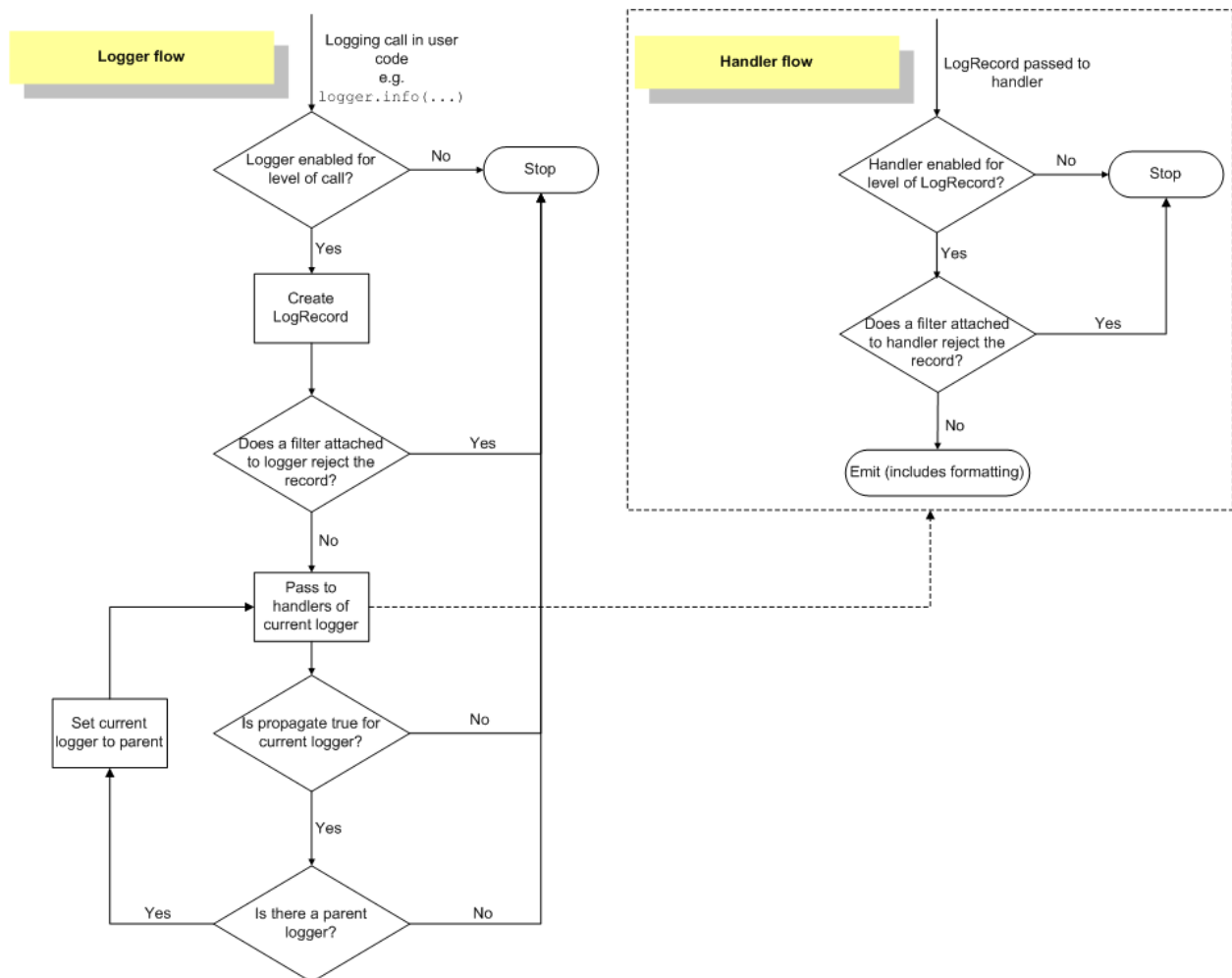
basicConfig() によるデフォルトのフォーマットメッセージは次のようになります:

```
severity:logger name:message
```

basicConfig() の *format* キーワード引数にフォーマット文字列を渡すことで、これを変更できます。フォーマット文字列を構成するためのすべてのオプションは、*formatter-objects* を参照してください。

2.1 Logging Flow

次の図はログイベントが logger と handler をどう流れるかを示しています。



2.2 ロガー

Logger オブジェクトの仕事は大きく三つに分かれます。一つ目は、アプリケーションが実行中にメッセージを記録できるように、いくつかのメソッドをアプリケーションから呼べるようにしています。二つ目に、ロガーオブジェクトはどのメッセージに対して作用するかを、深刻度 (デフォルトのフィルタ機構) またはフィルタオブジェクトに基づいて決定します。三つ目に、ロガーオブジェクトは関心を持っているすべてのログハンドラに関連するログメッセージを回送します。

とりわけ広く使われるロガーオブジェクトのメソッドは、二つのカテゴリーに分類できます: 設定とメッセージ送信です。

これらが設定メソッドの中でよく使われます:

- `Logger.setLevel()` はロガーが扱うログメッセージの最も低い深刻度を指定します。組み込みの深刻度の中では `DEBUG` が一番低く、`CRITICAL` が一番高くなります。たとえば、深刻度が `INFO` と設定されたロガーは `INFO`, `WARNING`, `ERROR`, `CRITICAL` のメッセージしか扱わず、`DEBUG` メッセージは無視します。
- `Logger.addHandler()` と `Logger.removeHandler()` は、ハンドラオブジェクトをロガーオブジェクトから追加または削除します。ハンドラについては、[ハンドラ](#) で詳しく述べます。
- `Logger.addFilter()` と `Logger.removeFilter()` はロガーオブジェクトにフィルタオブジェクトを追加または削除します。フィルタについては、[filter](#) で詳しく述べます。

これらのメソッドを、生成したすべてのロガーに毎回呼び出さなければならないわけではありません。最後の 2 段落を参照してください。

ロガーオブジェクトが設定されれば、以下のメソッドがログメッセージを生成します:

- `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()`, `Logger.critical()` はすべて、メッセージとメソッド名に対応したレベルでログ記録を作り出します。メッセージは実際にはフォーマット文字列であり、通常の文字列代入に使う `%s`, `%d`, `%f` などを含むことができます。残りの引数はメッセージの代入される位置に対応するオブジェクトのリストです。`**kwargs` については、ログ記録メソッドが気にするキーワードは `exc_info` だけで、例外の情報をログに記録するかを決定するのに使います。
- `Logger.exception()` は `Logger.error()` と似たログメッセージを作成します。違いは `Logger.exception()` がスタックトレースと一緒にダンプすることです。例外ハンドラでだけ使うようにしてください。
- `Logger.log()` はログレベルを明示的な引数として受け取ります。これは上に挙げた便宜的なログレベル毎のメソッドを使うより少しコード量が多くなりますが、独自のログレベルを使うことができます。

`getLogger()` は、指定されればその特定の名前の、そうでなければ `root` のロガーインスタンスへの参照を返します。ロガーの名前はピリオド区切りの階層構造を表します。同じ名前でも `getLogger()` を複数回呼び出した場合、同一のロガーオブジェクトへの参照が返されます。階層リストを下ったロガーはリスト上位のロガーの子で

す。たとえば、名前が `foo` であるロガーがあったとして、`foo.bar`, `foo.bar.baz`, `foo.bam` といった名前のロガーはすべて `foo` の子孫になります。

ロガーには、**有効レベル** (*effective level*) の概念があります。ロガーにレベルが明示的に設定されていなければ、代わりに親のレベルがその有効レベルとして使われます。親のレベルが設定されなければ、**その** 親のレベルが確かめられ、以下同様に、明示的に設定されたレベルが見つかるまで祖先が探されます。ルートロガーは、必ず明示的なレベルが設定されています (デフォルトでは `WARNING` です)。イベントを処理するかを決定するとき、ロガーの有効レベルを使って、イベントがロガーのハンドラに渡されるかが決められます。

子ロガーはメッセージを親ロガーのハンドラに伝えます。このため、アプリケーションが使っているすべてのロガーのためのハンドラを定義して設定する必要はありません。トップレベルのロガーのためのハンドラだけ設定しておいて必要に応じて子ロガーを作成すれば十分です。(しかし、ロガーの *propagate* 属性を `False` に設定することで、伝播を抑制できます。)

2.3 ハンドラ

Handler オブジェクトは適切なログメッセージを (ログメッセージの深刻度に基づいて) ハンドラの指定された出力先に振り分けることに責任を持ちます。**Logger** オブジェクトには `addHandler()` メソッドで 0 個以上のハンドラを追加することができます。例として、あるアプリケーションがすべてのログメッセージをログファイルに、`error` 以上のすべてのログメッセージを標準出力に、`critical` のメッセージはすべてメールアドレスに、それぞれ送りたいとします。この場合、3 つの個別のハンドラがそれぞれの深刻度と宛先に応じて必要になります。

このライブラリには多くのハンドラが用意されています (**便利なハンドラ** を参照してください) が、このチュートリアルでは `StreamHandler` と `FileHandler` だけを例に取り上げます。

アプリケーション開発者にとってハンドラを扱う上で気にするべきメソッドは極々限られています。組み込みのハンドラオブジェクトを使う (つまり自作ハンドラを作らない) 開発者に関係あるハンドラのメソッドは、次の設定用のメソッドだけでしょう:

- `setLevel()` メソッドは、ロガーオブジェクトの場合と同様に、適切な出力先に振り分けられるべき最も低い深刻度を指定します。なぜ 2 つも `setLevel()` メソッドがあるのでしょうか? ロガーで設定されるレベルは、付随するハンドラにどんな深刻度のメッセージを渡すか決めます。ハンドラで設定されるレベルは、ハンドラがどのメッセージを送るべきか決めます。
- `setFormatter()` でこのハンドラが使用する `Formatter` オブジェクトを選択します。
- `addFilter()` および `removeFilter()` はそれぞれハンドラへのフィルタオブジェクトの設定と解除を行います。

アプリケーションのコード中では `Handler` のインスタンスを直接インスタンス化して使ってはなりません。代わりに、`Handler` クラスはすべてのハンドラが持つべきインターフェイスを定義し、 subclasses が使える (もしくはオーバーライドできる) いくつかのデフォルトの振る舞いを規定します。

2.4 フォーマッタ

フォーマッタオブジェクトは最終的なログメッセージの順序、構造および内容を設定します。基底クラスである `logging.Handler` とは違って、アプリケーションのコードはフォーマッタクラスをインスタンス化しても構いません。特別な振る舞いをさせたいアプリケーションではフォーマッタのサブクラスを使う可能性もあります。コンストラクタは三つのオプション引数を取ります -- メッセージのフォーマット文字列、日付のフォーマット文字列、スタイル標識です。

```
logging.Formatter.__init__(fmt=None, datefmt=None, style='%')
```

フォーマット文字列に何も渡さない場合は、デフォルトで行メッセージが利用されます。また、日付フォーマットに何も渡さない場合は、デフォルトで以下のフォーマットが利用されます:

```
%Y-%m-%d %H:%M:%S
```

で、最後にミリ秒が付きます。style は '%', '{', '\$' のいずれかです。一つ指定されなければ、'%' が使われます。

style が '%' の場合、メッセージフォーマット文字列では `%(dictionary key)s` 形式の置換文字列が使われます; キーに指定できる属性名は `logrecord-attributes` に文書化されています。style が '{' の場合、メッセージフォーマット文字列は (キーワード引数を使う) `str.format()` と互換となります。style が '\$' の場合、メッセージフォーマット文字列は `string.Template.substitute()` で期待されているものと一致します。

バージョン 3.2 で変更: style パラメータが追加されました。

次のメッセージフォーマット文字列は、人が読みやすい形式の時刻、メッセージの深刻度、およびメッセージの内容を、順番に出力します:

```
'%(asctime)s - %(levelname)s - %(message)s'
```

フォーマッタは、ユーザが設定できる関数を使って、生成時刻をタプルに記録します。デフォルトでは、`time.localtime()` が使われます。特定のフォーマッタインスタンスに対してこれを変更するには、インスタンスの `converter` 属性を `time.localtime()` や `time.gmtime()` と同じ署名をもつ関数に設定してください。すべてのフォーマッタインスタンスに対してこれを変更するには、例えば全てのロギング時刻を GMT で表示するには、フォーマッタクラスの `converter` 属性を (GMT 表示の `time.gmtime` に) 設定してください。

2.5 ロギングの環境設定

プログラマは、ロギングを 3 種類の方法で設定できます:

1. 上述の設定メソッドを呼び出す Python コードを明示的に使って、ロガー、ハンドラ、そしてフォーマッタを生成する。
2. ロギング設定ファイルを作り、それを `fileConfig()` 関数を使って読み込む。
3. 設定情報の辞書を作り、それを `dictConfig()` 関数に渡す。

最後の 2 つの選択肢については、logging-config-api で解説しています。以下の例では、Python コードを使って、とても簡単なロガー、コンソールハンドラ、そして簡単なフォーマッタを設定しています:

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

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

このモジュールを実行すると、コマンドラインによって以下の出力がなされます:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

以下の Python モジュールは、ロガー、ハンドラ、フォーマッタをほとんど上述の例と同じように生成していますが、オブジェクト名だけが異なります:

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')
```

(次のページに続く)

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

これは 'logging.conf' ファイルに記載する例です:

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=
```

出力は、設定ファイルに基づく例とだいたい同じです:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

この通り、設定ファイルの方法は、主に設定とコードが分かれ、非コードがロギングプロパティを変えやすくなるという点で、Python コードの方法より少し優れています。

警告: `fileConfig()` 関数はデフォルト引数 `disable_existing_loggers` を取り、後方互換性のためにデフォルト値は `True` になっています。これはあなたの望むものかもしれませんが、そうでないかもしれません。というのは、設定で明示的に指定したクラス (もしくはその親クラス) を除いて、`fileConfig()` が呼び出される前に存在した非ルートロガーを無効化してしまうからです。より詳細なことはリファレンスを参照し、望むならこの引数に `False` を指定してください。

`dictConfig()` に渡した辞書で、キー `disable_existing_loggers` にブール値を指定することができ、辞書で指定しなかった場合はデフォルトで `True` と解釈されます。こうすると上で説明したロガー無効化が動作しますが、これはあなたが望んだものでないかもしれません - その場合は、明示的にキーに `False` を指定してください。

なお、設定ファイルで参照されるクラス名は、`logging` モジュールに対して相対であるか、通常のインポート機構を使って解決される絶対である値でなければなりません。従って、(`logging` モジュールに相対な) `WatchedFileHandler` または (Python インポートパスとして `mypackage` が使えるとき、パッケージ `mypackage` のモジュール `mymodule` で定義されたクラスに) `mypackage.mymodule.MyHandler` のどちらかが使えます。

Python 3.2 では、ロギングを設定するのに新しく、辞書に設定情報を持たせる手段が導入されました。これは、上で概説した設定ファイルに基づく方法による機能の上位版を提供し、新しいアプリケーションやデプロイにはこのメソッドが推奨されます。Python の辞書を使って設定情報を保持し、辞書は他の用途にも使えるので、設定の選択肢が広がります。例えば、JSON フォーマットの設定ファイルや、YAML 処理機能があれば YAML フォーマットのファイルを使って、設定辞書を構成できます。また、もちろん、Python コードで辞書を構成し、ソケットを通して pickle 化された形式を受け取るなど、アプリケーションで意味があるいかなるやり方でも使えます。

以下は上記と同じ設定を、辞書形式に基づく YAML フォーマットとして書いたものです:

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
```

(次のページに続く)

```
root:
  level: DEBUG
  handlers: [console]
```

辞書を使ったロギングについて詳細は、`logging-config-api` を参照してください。

2.6 環境設定が与えられないとどうなるか

ロギング環境設定を与えられないと、ロギングイベントを出力しなければならないのに、イベントを出力するハンドラが見つからないことがあります。この状況での `logging` パッケージの振る舞いは、Python のバージョンに依ります。

Python 3.2 より前のバージョンでは、振る舞いは以下の通りです:

- `logging.raiseExceptions` が `False` (製品モード) なら、イベントは黙って捨てられます。
- `logging.raiseExceptions` が `True` (開発モード) なら、メッセージ 'No handlers could be found for logger X.Y.Z' が一度表示されます。

Python 3.2 以降バージョンでは、振る舞いは以下の通りです:

- イベントは、`logging.lastResort` に格納された「最終手段ハンドラ」を使用して出力されます。この内部的なハンドラはどんなロガーにも関係しておらず、イベント記述メッセージを現在の `sys.stderr` の値に書く `StreamHandler` のように動作します (したがって、あらゆるリダイレクトの効果が反映されます)。メッセージに対してフォーマットは行われません - 裸のイベント記述メッセージだけが印刷されます。ハンドラのレベルは `WARNING` にセットされ、これより重大度が大きなすべてのイベントが出力されます。

3.2 より前の動作にするために、`logging.lastResort` を `None` に設定することもできます。

2.7 ライブラリのためのロギングの設定

ロギングを使うライブラリを開発するときは、ライブラリがどのようにロギングを使うのか、例えば使われているロガーの名前などを、ドキュメントにしておくべきです。ロギングの設定については、いくつか考えておくべきこともあります。使っているアプリケーションがロギングを使っていなくて、ライブラリコードがロギングを呼び出すと、(前の節で解説したように) 重大度 `WARNING` 以上のイベントが、`sys.stderr` に表示されます。これが最高のデフォルトの振る舞いと見なされます。

何らかの理由で、ロギング設定がなされていないときにメッセージを表示 **させたくない** なら、あなたのライブラリのトップレベルロガーに何もしないハンドラを取り付けられます。このハンドラは何も処理しないというだけで、全てのライブラリイベントに対してハンドラが見つかるので、メッセージが表示されることを防げます。ライブラリのユーザがアプリケーションでの使用のためにロギングを設定したら、それはおそらくハンドラを追加す

る設定でしょうが、そしてレベルが適切に設定されたら、ライブラリコード内でなされたログイングの呼び出しは、通常通りそのハンドラに出力を送るようになります。

何もしないハンドラ `NullHandler` (Python 3.1 以降) は、`logging` パッケージに含まれます。このハンドラのインスタンスを、(ログイング設定がなされていないときにライブラリのログイベントを `sys.stderr` に出力させたくないなら) ライブラリの名前空間で使われるトップレベルロガーに取り付けられます。ライブラリ `foo` によるすべてのログイングが、`'foo.x'`, `'foo.x.y'` その他に該当する名前のロガーによってなされるなら:

```
import logging
logging.getLogger('foo').addHandler(logging.NullHandler())
```

とすれば望んだ効果が得られるでしょう。組織が複数のライブラリを作り出すなら、指定されるロガー名は単に `'foo'` ではなく、`'orgname.foo'` になります。

注釈: ライブラリのロガーには、`NullHandler` 以外のハンドラを追加しない ことを強く推奨します。これは、ハンドラの設定が、あなたのライブラリを使うアプリケーション開発者にも伝播するからです。アプリケーション開発者は、対象となる聴衆と、そのアプリケーションにどのハンドラが最も適しているかを知っています。ハンドラを 'ボンネットの中で' 加えてしまうと、ユニットテストをして必要に応じたログを送達する能力に干渉しかねません。

3 ログイングレベル

ログレベルの数値は以下の表のように与えられています。これらは基本的に自分でレベルを定義したい人のためのもので、定義するレベルを既存のレベルの間に位置づけるためには具体的な値が必要になります。もし数値が他のレベルと同じだったら、既存の値は上書きされその名前は失われます。

レベル	数値
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

レベルはロガーに関連付けることもでき、開発者が設定することも、保存されたログ記録設定を読み込む際に設定することもできます。ロガーに対してログ記録メソッドが呼び出されると、ロガーは自らのレベルとメソッド呼び出しに関連付けられたレベルを比較します。ロガーのレベルがメソッド呼び出しのレベルよりも高い場合、実際のログメッセージは生成されません。これはログ出力の冗長性を制御するための基本的なメカニズムです。

ログ記録されるメッセージは `LogRecord` クラスのインスタンスとしてエンコードされます。ロガーがあるイベントを実際にログ出力すると決定した場合、ログメッセージから `LogRecord` インスタンスが生成されます。

ログ記録されるメッセージは、ハンドラ (*handlers*) を通してディスパッチ機構にかけられます。ハンドラは `Handler` クラスのサブクラスのインスタンスで、ログ記録された (`LogRecord` 形式の) メッセージが、そのメッセージの伝達対象となる相手 (エンドユーザ、サポートデスクのスタッフ、システム管理者、開発者) に行き着くようにする役割を持ちます。ハンドラには特定の出力先を意図された `LogRecord` インスタンスが渡されます。各ロガーは 0 個以上のハンドラを (`Logger` の `addHandler()` メソッド) で関連付けることができます。ロガーに直接関連付けられたハンドラに加えて、**ロガーの上位にあるロガーすべてに関連付けられたハンドラ** がメッセージを処理する際に呼び出されます (ただしロガーの *propagate* フラグが `false` 値にセットされている場合を除きます。その場合は、祖先ハンドラへの伝搬はそこで止まります)。

ロガーと同様に、ハンドラは関連付けられたレベルを持つことができます。ハンドラのレベルはロガーのレベルと同じ方法で、フィルタとして働きます。ハンドラがあるイベントを実際に処理すると決定した場合、`emit()` メソッドが使われ、メッセージを出力先に送信します。ほとんどのユーザ定義の `Handler` のサブクラスで、この `emit()` をオーバーライドする必要があるでしょう。

3.1 カスタムレベル

独自のレベルを定義することは可能ですが、必須ではなく、実経験上は既存のレベルが選ばれます。しかし、カスタムレベルが必要だと確信するなら、レベルの定義には多大な注意を払うべきで、**ライブラリの開発の際、カスタムレベルを定義することはとても悪いアイデア** になり得ます。これは、複数のライブラリの作者がみな独自のカスタムレベルを定義すると、与えられた数値が異なるライブラリで異なる意味になりえるため、開発者がこれを制御または解釈するのが難しくなるからです。

4 便利なハンドラ

基底の `Handler` クラスに加え、多くの便利なサブクラスが提供されています:

1. `StreamHandler` インスタンスは、メッセージをストリーム (ファイル風オブジェクト) に送ります。
2. `FileHandler` インスタンスは、メッセージをディスクファイルに送ります。
3. `BaseRotatingHandler` は、ある地点でログファイルを循環させるハンドラの基底クラスです。これを直接インスタンス化することは意図されていません。代わりに、`RotatingFileHandler` や `TimedRotatingFileHandler` を使用してください。
4. `RotatingFileHandler` インスタンスは、メッセージをディスクファイルに送り、最大ログファイル数とログファイル循環をサポートします。
5. `TimedRotatingFileHandler` インスタンスは、メッセージをディスクファイルに送り、ログファイルを特定時間のインターバルで循環します。

6. `SocketHandler` インスタンスは、TCP/IP ソケットにメッセージを送ります。バージョン 3.4 から、Unix ドメインソケットもサポートされます。
7. `DatagramHandler` インスタンスは UDP ソケットにメッセージを送ります。バージョン 3.4 から、Unix ドメインソケットもサポートされます。
8. `SMTPHandler` インスタンスは、メッセージを指示された email アドレスに送ります。
9. `SysLogHandler` インスタンスは、メッセージを、必要ならばリモートマシンの、Unix syslog daemon に送ります。
10. `NTEventLogHandler` インスタンスは、メッセージを Windows NT/2000/XP イベントログに送ります。
11. `MemoryHandler` インスタンスは、メッセージを、特定の基準が満たされる度に流される、メモリ中のバッファに送ります。
12. `HTTPHandler` インスタンスは、メッセージを、GET または POST セマンティクスを使って、HTTP サーバに送ります。
13. `WatchedFileHandler` インスタンスは、ロギングする先のファイルを監視します。ファイルが変更されると、そのファイルは閉じられ、ファイル名を使って再び開かれます。このハンドラは Unix 系のシステムにのみ便利です。Windows は、使われている基の機構をサポートしていません。
14. `QueueHandler` インスタンスは、`queue` モジュールや `multiprocessing` モジュールなどで実装されているキューにメッセージを送ります。instances send messages to a queue, such as those implemented in the `queue` or `multiprocessing` modules.
15. `NullHandler` インスタンスは、エラーメッセージに対して何もしません。これは、ライブラリ開発者がロギングを使いたいが、ライブラリのユーザがロギングを設定してなくても 'No handlers could be found for logger XXX' メッセージを表示させたくない場合に使われます。詳しい情報は [ライブラリのためのロギングの設定](#) を参照してください。

バージョン 3.1 で追加: `NullHandler` クラス。

バージョン 3.2 で追加: `QueueHandler` クラス。

コア logging パッケージで、`NullHandler`、`StreamHandler` および `FileHandler` クラスが定義されています。その他のハンドラは、サブモジュールの `logging.handlers` で定義されています。(環境設定機能のためのサブモジュール、`logging.config` もあります。)

ログメッセージは、`Formatter` クラスのインスタンスを通してフォーマット化してから表示されます。このインスタンスは、% 演算子と辞書で使うのに適切なフォーマット文字列で初期化されます。

複数のメッセージを一括してフォーマット化するには、`BufferingFormatter` が使えます。(一連の文字列のそれぞれに適用される) フォーマット文字列に加え、ヘッダとトレーラフォーマット文字列も提供されています。

ロガーレベルおよび/またはハンドラレベルに基づくフィルタリングで十分でなければ、`Filter` のインスタンスを `Logger` と `Handler` インスタンスの両方に (`addFilter()` を通して) 加えることができます。メッセージの処

理を続ける前に、ロガーもハンドラも、全てのフィルタに許可を求めます。フィルタのいずれかが偽値を返したら、メッセージの処理は続けられません。

基本の `Filter` 機能では、特定のロガー名でのフィルタリングをできます。この機能が使われると、指名されたロガーに送られたメッセージとその子だけがフィルタを通り、その他は落とされます。

5 ログ記録中に発生する例外

`logging` パッケージは、ログを生成している間に起こる例外を飲み込むように設計されています。これは、ログ記録イベントを扱っている間に発生するエラー（ログ記録の設定ミス、ネットワークまたは他の同様のエラー）によってログ記録を使用するアプリケーションが早期に終了しないようにするためです。

`SystemExit` と `KeyboardInterrupt` 例外は決して飲み込まれません。`Handler` サブクラスの `emit()` メソッドの間に起こる他の例外は、`handleError()` メソッドに渡されます。

`Handler` の `handleError()` のデフォルト実装は、モジュールレベル変数 `raiseExceptions` が設定されているかどうかチェックします。設定されているなら、トレースバックが `sys.stderr` に出力されます。設定されていないなら、例外は飲み込まれます。

注釈: `raiseExceptions` のデフォルト値は `True` です。これは、開発の間に起こるどんな例外についても通常は通知してほしいからです。実運用環境では `raiseExceptions` を `False` に設定することをお勧めします。

6 任意のオブジェクトをメッセージに使用する

前の節とそこで挙げた例では、イベントを記録するときに渡されたメッセージが文字列であると仮定していました。しかし、これは唯一の可能性ではありません。メッセージとして任意のオブジェクトを渡すことができます。そして、ロギングシステムがそのオブジェクトを文字列表現に変換する必要があるとき、オブジェクトの `__str__()` メソッドが呼び出されます。実際、そうしたければ、文字列表現を計算することを完全に避けることができます - 例えば、`SocketHandler` は、イベントを `pickle` してネットワーク上で送信することでログ出力します。

7 最適化

`message` 引数の整形は、必要になるまで延期されます。しかしながら、ログ記録メソッドに渡す引数を計算するだけでもコストがかかる場合があります。ロガーが単にイベントを捨てるなら、その計算を避けたいと考えるかもしれません。どうするかを決定するために `isEnabledFor()` メソッドを呼ぶことができます。このメソッドは引数にレベルを取って、そのレベルの呼び出しに対して `Logger` がイベントを生成するなら `true` を返します。このようにコードを書くことができます:

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(),
                 expensive_func2())
```

このようにすると、ロガーの閾値が `DEBUG` より上に設定されている場合、`expensive_func1()` と `expensive_func2()` の呼び出しは行われません。

注釈: ある種のケースでは、`isEnabledFor()` それ自身があなたが期待するよりも高価になる場合があります (たとえば明示的なレベル指定がロガー階層の上位のみに設定されていて、ロガーが深くネストされているような場合です)。そのようなケース (あるいはタイトなループ内でメソッド呼び出しを避けたい場合) は、`isEnabledFor()` 結果をローカルにもしくはインスタンス変数としてキャッシュし、メソッドを毎回呼び出すかわりに使えば良いです。そのようなキャッシュ値は、(まったく一般的ではありませんが) ログ設定がアプリケーション実行中に動的に変更された場合にのみ再計算が必要でしょう。

これ以外にも、どんなログ情報が集められるかについてより正確なコントロールを必要とする、特定のアプリケーションでできる最適化があります。これは、ログ記録の間の不要な処理を避けるためにできることのリストです:

不要な情報	それを避ける方法
呼び出しがどこから行われたかに関する情報。	<code>logging._srcfile</code> を <code>None</code> にする。このことにより <code>sys._getframe()</code> 呼び出しを避けることが出来、PyPy のような環境 (<code>sys._getframe()</code> の高速化が出来ない) において高速化の役に立ちます。
スレッド情報。	<code>logging.logThreads</code> を <code>0</code> にする。
プロセス情報。	<code>logging.logProcesses</code> を <code>0</code> にする。

また、コア `logging` モジュールが基本的なハンドラだけを含んでいることに注意してください。`logging.handlers` と `logging.config` をインポートしなければ、余分なメモリを消費することはありません。

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。

ロギングクックブック

索引

アルファベット以外

`__init__()` (*logging.logging.Formatter* のメソッド), [12](#)