1 時間でわかる！〜 Logging Tutorial〜

for Python 3.6

このスライドについて

公式のドキュメントを自分なりにまとめただけですので，詳細を学びたい人は以下のページをどうぞ.

- logging - Python 用ロギング機能
- Python3.6: Logging HOWTO

Logging HOWTO

logging チュートリアルの準備

Logging とは？

> 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

- 今回のスライドでは logging モジュールの使い方を理解してもらうことを目標とします

有効レベルについて

ロギング関数は，そのイベントの重大度から名前がつけられる
(以下に重大度が増す順に有効レベルを記述)

レベル	使いどき
DEBUG	問題の診断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	想定外の問題が近く起こりそうであることの表示
ERROR	より重大な問題により，ある機能が実行できないこと
CRITICAL	実行ができないような重大なエラー

簡単な例

- logging.{debug, ..., critical} でログ出力を行う
- デフォルトのレベルは WARNING なので，それ未満のレベルのログは現れない

```
import logging

logging.warning('disk space low')
logging.info('start application')
```

- 有効レベルの設定には，logging.basicConfig の level を指定する.
- 例えばレベルを DEBUG にしたければ，ログ出力を行う前に以下を挿入する

```
logging.basicConfig(level=logging.DEBUG)
```

ファイルへの出力

- logging.basicConfig の filename を指定する
- デフォルトでファイルに追記するので，上書きしたければ filemode を 'w' で指定する
- 有効レベル以上のログがファイルに書き込まれる

```
import logging

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

logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

表示メッセージのフォーマット

- basicConfig の format を指定することで可能
- 日付の表示を書き換えたい時は datefmt を追加で指定する

```
logging.basicConfig(
      format='[%(levelname)s]%(asctime)s\n%(message)s',
      datefmt='%Y/%m/%d %H:%M:%S')
```

主な Log Record 属性（詳細はこちら）

- %(message)s：ログメッセージを表示する
- %(levelname)s：有効レベルを表示する
- %(asctime)s：イベントの日付と時刻を表示する．datefmt のフォーマットは time.strftime を参照
- %(name)s:ロギングに使われたロガーの名前

Logging HOWTO

ここから logging チュートリアル

何故ここからがチュートリアル？

- 今までの説明でのロギングは全てルートロガーを使ってる．そのため，複数ファイルにまたがったロギングを行った時**どのログが何処のものかわかりずらい**という問題がある．
- なので，もっと踏み込んで学習しましょうという話

logging ライブラリのモジュール

モジュール	説明
ロガー	アプリケーションコードが直接使うインタフェースを公開します
ハンドラ	(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	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	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, handle, filter, formatter 間でやりとりされる

ロギングの実行

- ロガーには階層構造があり，根源はルートロガーと呼ばれる
- ロガーはデフォルトでルートロガーを生成し，関数 dubug(), ..., critical() によって使われる

```
import logging
logger = logging.getLogger()
logger.info('Hello, World!')
```

ロガーの命名

- ロガーにはそれぞれ名前があり，名前空間階層構造にドットをセパレータとして概念的に並べれられている
- 名前は自由につけて良いが，以下のようにモジュールレベルロガーを用いると良い

```
from logging import getLogger
logger = getLogger(__name__)
```

- これによりロガー名だけで直感的に何処でイベントのログが取られたかわかるようになる

logging.Logger

Logger オブジェクトの仕事は大きく３つに分かれる

1. アプリ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から呼べるようにする
2. どのメッセージに対して作用するか，深刻度やフィルタオブジェクトに基づいて決定する
3. 関心を持っている全てのログハンドラに関連するログメッセージを回送する

Logger の設定

1. ロガーの名前を getLogger() の引数に与える
2. ロガーの有効レベルを setLevel() で追加する
3. ハンドラオブジェクト，フィルタオブジェクトを addHandler(), addFilter() で追加する

ロガー毎に有効レベル，ハンドラ，フィルタを毎回呼ばなくても良い

> - ロガーの有効レベルは未設定の場合は親ロガーを参照する. ルートロガーのデフォルトは WARNING
> - 子ロガーはメッセージを親ロガーのハンドラに伝えるので，トップレベルロガーのためのハンドラだけ設定し，必要に応じて子ロガーでも作成すれば良い. （ロガーの propagate 属性を False にすれば，伝播を抑制できる）

logging.Handler

- Handler オブジェクトは適切なログメッセージを指定された出力先に振り分ける役割を持つ
- Logger オブジェクトに addHandler() でハンドラを追加する
- Handler には 有効レベル, Formatter, Filter を設定可能
    - setLevel(), setFormatter(), addFilter()

```
from logging import StreamHandler
handler = StreamHandler
logger.addHandler(handler)
```

Handler の種類

- コード中で Handler() を直接インスタンス化してはいけない．代わりに便利な子クラスがたくさんあるので，それらを使いましょう
    - （ハンドラーを自作することも可能）

ハンドラー	機能
StreamHandler	ログ出力を sys.stdout とかのストリームに送信する
FileHandler	ログ出力をディスク上のダイルに送信する
NullHandler	いかなる書式化も出力も行わない
SMTPHandler	SMTP を介してログ記録メッセージの送信をする

- ハンドラー一覧：
  https://docs.python.jp/3/library/logging.handlers.html

logging.Filter

- 規定のフィルタクラスは，ロガー階層構造内の特定地点の配下にあるイベントだけを許可する
- 例えば，Filter('A.B') はロガー 'A.B', 'A.B.D' などによる記録は許可するが，'A.BB', 'B.A.B' などは許可しない

```
from logging import Filter
filter = Filter('root.viewer')
logger = getLogger(__name__)
logger.addFilter(filter)
```

- また，Filter は渡された LogRecord を修正する事ができる
  - COOKBOOK の filter-contextual を参照

logging.Formatter

> class logging.Formatter(fmt=None, datefmt=None, style='%')

- コンストラクタは三つの引数をとる
  - 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
- 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る
- 日付フォーマットのデフォルトは `%Y-%m-%d %H:%M:%S`

```
fmtr = logging.Formatter(fmt='%(asctime)s: %(message)s')
```

ロギングの環境設定

```
import logging

logger = logging.getLogger('simpleExample')
logger.setLevel(logging.INFO)

stream_handler = logging.StreamHandler()
stream_handler.setLevel(logging.INFO)

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

stream_handler.setFormatter(formatter)
logger.addHandler(stream_handler)

# application code
logger.info('application.start()')
```

- ちょっと見辛いと思う人は，**ロギング設定ファイル**を使いましょう

ロギング設定ファイル

```
[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=
```

- 詳細は環境設定ファイルの書式を参照

ロギング設定ファイルの適用

- コードがすっきりする！
- 設定とコードが分かれ，非コーダがロギングプロパティを変え易い

```
from logging import getLogger
from logging.config import fileConfig

fileConfig('logging.conf')
logger = getLogger('simpleExample')

# application code
logger.info('application.start()')
```

まとめ

- ちょっとしたコードだったら print() とか logging.info() とかで良い
- 真面目に大きめなコードを書く場合は，Logger を活用しよう
- とりあえずログ記録を sys.stdout に流したければ，以下を書こう

```
import logging

logger = logging.getLogger(__name__)
handler = logging.StreamHandler()
logger.setLevel(logging.info)
handler.setLevel(logging.info)
logger.addHandler(handler)

logger.info('Hello, world!')
```

おまけ

logging.error() vs logging.exception()

レベル ERROR のメッセージは，logging.error()，logging.exception() で記録できる．logging.exceptions() はスタックトレースを一緒にダンプするので，例外ハンドラでのみ用いると良い．

```
tyr:
  dosomething()
except (KeyError, ValueError) as err:
  logger.exception('Error dosomething: %s', err)
```

> - Python の例外情報（Traceback）を Log に記録するには Logger.exception が便利って話

最適化

ログ引数メソッドに有効レベルの低い高コストな関数がある場合，以下のようにすると良いかもしれません.

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

> ある種のケースでは，isEnabledFor() 自身が高価になる可能性があります．そのような場合は，isEnablesFor() 結果をローカルにキャッシュし，メソッド呼び出しの代わりに使いましょう