Tutoriel sur la journalisation

Version 2.7.16

Guido van Rossum
and the Python development team

octobre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

1 Les bases de l'utilisation du module *logging* **2**
- 1.1 Quand utiliser *logging* . . . 2
- 1.2 Un exemple simple . . . 3
- 1.3 Enregistrer les évènements dans un fichier . . . 3
- 1.4 Employer *logging* à partir de différents modules . . . 4
- 1.5 Journalisation de données variables . . . 5
- 1.6 Modifier le format du message affiché . . . 5
- 1.7 Afficher l'horodatage dans les messages . . . 5
- 1.8 Étapes suivantes . . . 6

2 Usage avancé de Logging **6**
- 2.1 Flux du processus de journalisation . . . 7
- 2.2 Loggers . . . 8
- 2.3 Handlers . . . 8
- 2.4 Formatters . . . 9
- 2.5 Configuration de *Logging* . . . 9
- 2.6 Comportement par défaut (si aucune configuration n'est fournie) . . . 12
- 2.7 Configuration de la journalisation pour une bibliothèque . . . 13

3 Niveaux de journalisation **13**
- 3.1 Niveaux personnalisés . . . 14

4 Gestionnaires utiles **14**

5 Exceptions levées par la journalisation **15**

6 Utilisation d'objets arbitraires comme messages **15**

7 Optimisation **16**

Index **17**

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

1 Les bases de l'utilisation du module *logging*

La journalisation (*logging* en anglais) est une façon de suivre les événements qui ont lieu durant le fonctionnement d'un logiciel. Le développeur du logiciel ajoute des appels à l'outil de journalisation dans son code pour indiquer que certains événements ont eu lieu. Un événement est décrit par un message descriptif, qui peut éventuellement contenir des données variables (c'est-à-dire qui peuvent être différentes pour chaque occurrence de l'événement). Un événement a aussi une importance que le développeur lui attribue ; cette importance peut aussi être appelée *niveau* ou *sévérité*.

1.1 Quand utiliser *logging*

Le module *logging* fournit un ensemble de fonctions de commodités pour une utilisation simple du module. Ce sont les fonctions `debug()`, `info()`, `warning()`, `error()` et `critical()`. Pour déterminer quand employer la journalisation, voyez la table ci-dessous, qui vous indique, pour chaque tâche parmi les plus communes, l'outil approprié.

Tâche que vous souhaitez mener	Le meilleur outil pour cette tâche
Affiche la sortie console d'un script en ligne de commande ou d'un programme lors de son utilisation ordinaire	`print()`
Rapporter des évènements qui ont lieu au cours du fonctionnement normal d'un programme (par exemple pour suivre un statut ou examiner des dysfonctionnements)	`logging.info()` (ou `logging.debug()` pour une sortie très détaillée à visée diagnostique)
Émettre un avertissement (*warning* en anglais) en relation avec un évènement particulier au cours du fonctionnement d'un programme	`warnings.warn()` dans le code de la bibliothèque si le problème est évitable et l'application cliente doit être modifiée pour éliminer cet avertissement `logging.warning()` si l'application cliente ne peut rien faire pour corriger la situation mais l'évènement devrait quand même être noté
Rapporter une erreur lors d'un évènement particulier en cours d'exécution	Lever une exception
Rapporter la suppression d'une erreur sans lever d'exception (par exemple pour la gestion d'erreur d'un processus de long terme sur un serveur)	`logging.error()`, `logging.exception()` ou `logging.critical()`, au mieux, selon l'erreur spécifique et le domaine d'application

Les fonctions de journalisation sont nommées d'après le niveau ou la sévérité des évènements qu'elles suivent. Les niveaux standards et leurs applications sont décrits ci-dessous (par ordre croissant de sévérité) :

Niveau	Quand il est utilisé
`DEBUG`	Information détaillée, intéressante seulement lorsqu'on diagnostique un problème.
`INFO`	Confirmation que tout fonctionne comme prévu.
`WARNING`	L'indication que quelque chose d'inattendu a eu lieu, ou de la possibilité d'un problème dans un futur proche (par exemple « espace disque faible »). Le logiciel fonctionne encore normalement.
`ERROR`	Du fait d'un problème plus sérieux, le logiciel n'a pas été capable de réaliser une tâche.
`CRITICAL`	Une erreur sérieuse, indiquant que le programme lui-même pourrait être incapable de continuer à fonctionner.

Le niveau par défaut est `WARNING`, ce qui signifie que seuls les évènements de ce niveau et au-dessus sont suivis, sauf si le paquet *logging* est configuré pour faire autrement.

Les évènements suivis peuvent être gérés de différentes façons. La manière la plus simple est de les afficher dans la console. Une autre méthode commune est de les écrire dans un fichier.

1.2 Un exemple simple

Un exemple très simple est :

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

Si vous entrez ces lignes dans un script que vous exécutez, vous verrez :

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

affiché dans la console. Le message `INFO` n'apparaît pas parce que le niveau par défaut est `WARNING`. Le message affiché inclut l'indication du niveau et la description de l'évènement fournie dans l'appel à *logging*, ici « Watch out ! ». Ne vous préoccupez pas de la partie « *root* » pour le moment : nous détaillerons ce point plus bas. La sortie elle-même peut être formatée de multiples manières si besoin. Les options de formatage seront aussi expliquées plus bas.

1.3 Enregistrer les évènements dans un fichier

Il est très commun d'enregistrer les évènements dans un fichier, c'est donc ce que nous allons regarder maintenant. Il faut essayer ce qui suit avec un interpréteur Python nouvellement démarré, ne poursuivez pas la session commencée ci-dessus :

```
import logging
logging.basicConfig(filename='example.log',level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

And now if we open the file and look at what we have, we should find the log messages :

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
```

Cet exemple montre aussi comment on peut régler le niveau de journalisation qui sert de seuil pour le suivi. Dans ce cas, comme nous avons réglé le seuil à `DEBUG`, tous les messages ont été écrits.

If you want to set the logging level from a command-line option such as :

```
--log=INFO
```

et que vous passez ensuite la valeur du paramètre donné à l'option *–log* dans une variable *loglevel*, vous pouvez utiliser :

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

de manière à obtenir la valeur à passer à `basicConfig()` à travers l'argument *level*. Vous pouvez vérifier que l'utilisateur n'a fait aucune erreur pour la valeur de ce paramètre, comme dans l'exemple ci-dessous :

```
# 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, ...)
```

L'appel à `basicConfig()` doit être fait *avant* un appel à `debug()`, `info()`, etc. Comme l'objectif est d'avoir un outil de configuration simple et d'usage unique, seul le premier appel aura un effet, les appels suivants ne font rien.

Si vous exécutez le script plusieurs fois, les messages des exécutions successives sont ajoutés au fichier *example.log*. Si vous voulez que chaque exécution reprenne un fichier vierge, sans conserver les messages des exécutions précédentes, vous pouvez spécifier l'argument *filemode*, en changeant l'appel à l'exemple précédent par :

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

La sortie est identique à la précédente, mais le texte n'est plus ajouté au fichier de log, donc les messages des exécutions précédentes sont perdus.

1.4 Employer *logging* à partir de différents modules

Si votre programme est composé de plusieurs modules, voici une façon d'organiser l'outil de journalisation :

```
# 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')
```

If you run *myapp.py*, you should see this in *myapp.log* :

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

ce qui est normalement ce à quoi vous vous attendiez. Vous pouvez généraliser cela à plusieurs modules, en employant le motif de *mylib.py*. Remarquez qu'avec cette méthode simple, vous ne pourrez pas savoir, en lisant le fichier de log, *d'où* viennent les messages dans votre application, sauf dans la description de l'évènement. Si vous voulez suivre la localisation des messages, référerez-vous à la documentation avancée *Usage avancé de Logging*.

1.5 Journalisation de données variables

Pour enregistrer des données variables, utilisez une chaîne formatée dans le message de description de l'évènement et ajoutez les données variables comme argument. Par exemple :

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

affichera :

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

As you can see, merging of variable data into the event description message uses the old, %-style of string formatting. This is for backwards compatibility : the logging package pre-dates newer formatting options such as `str.format()` and `string.Template`. These newer formatting options *are* supported, but exploring them is outside the scope of this tutorial.

1.6 Modifier le format du message affiché

Pour changer le format utilisé pour afficher le message, vous devez préciser le format que vous souhaitez employer :

```
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')
```

which would print :

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

Notez que le *root* qui apparaissait dans les exemples précédents a disparu. Pour voir l'ensemble des éléments qui peuvent apparaître dans la chaîne de format, référez-vous à la documentation pour logrecord-attributes. Pour une utilisation simple, vous avez seulement besoin du *levelname* (la sévérité), du *message* (la description de l'évènement, avec les données variables) et peut-être du moment auquel l'évènement a eu lieu. Nous décrivons ces points dans la prochaine section.

1.7 Afficher l'horodatage dans les messages

Pour afficher la date ou le temps d'un évènement, ajoutez *%(asctime)* dans votre chaîne de formatage :

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

which should print something like this :

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

The default format for date/time display (shown above) is ISO8601. If you need more control over the formatting of the date/time, provide a *datefmt* argument to `basicConfig`, as in this example :

```
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.')
```

which would display something like this :

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

Le format de *datefmt* est le même que celui de `time.strftime()`.

1.8 Étapes suivantes

Nous concluons ainsi le tutoriel basique. Il devrait suffire à vous mettre le pied à l'étrier pour utiliser *logging*. Le module *logging* a beaucoup d'autre cordes à son arc, mais pour en profiter au maximum, vous devez prendre le temps de lire les sections suivantes. Si vous êtes prêt, servez-vous votre boisson préférée et poursuivons.

If your logging needs are simple, then use the above examples to incorporate logging into your own scripts, and if you run into problems or don't understand something, please post a question on the comp.lang.python Usenet group (available at https://groups.google.com/group/comp.lang.python) and you should receive help before too long.

Vous êtes encore là ? Vous pouvez lire les prochaines sections, qui donnent un peu plus de détails que l'introduction ci-dessus. Après ça, vous pouvez jeter un œil à logging-cookbook.

2 Usage avancé de Logging

La bibliothèque de journalisation adopte une approche modulaire et offre différentes catégories de composants : *loggers*, *handlers*, *filters* et *formatters*.

- Les enregistreurs (*loggers* en anglais) exposent l'interface que le code de l'application utilise directement.
- Les gestionnaires (*handlers*) envoient les entrées de journal (crées par les *loggers*) vers les destinations voulues.
- Les filtres (*filters*) fournissent un moyen de choisir finement quelles entrées de journal doivent être sorties.
- Les formateurs (*formatters*) spécifient la structure de l'entrée de journal dans la sortie finale.

L'information relative à un événement est passée entre *loggers*, *handlers* et *formatters* dans une instance de la classe `LogRecord`.

La journalisation est réalisée en appelant les méthodes d'instance de la classe `Logger` (que l'on appelle ci-dessous *loggers*). Chaque instance a un nom et les instances sont organisées conceptuellement comme des hiérarchies dans l'espace de nommage, en utilisant un point comme séparateur. Par exemple, un *logger* appelé *scan* est le parent des *loggers scan.text*, *scan.html* et *scan.pdf*. Les noms des *loggers* peuvent être ce que vous voulez et indiquent le sous-domaine d'une application depuis lequel le message enregistré a été émis.

Une bonne convention lorsqu'on nomme les *loggers* est d'utiliser un *logger* au niveau du module, dans chaque module qui emploie *logging*, nommé de la façon suivante :

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

Cela signifie que le nom d'un *logger* se rapporte à la hiérarchie du paquet et des modules, et il est évident de voir où un événement a été enregistré simplement en regardant le nom du *logger*.

La racine de la hiérarchie des *loggers* est appelée le *root logger*. C'est le *logger* utilisé par les fonctions `debug()`, `info()`, `warning()`, `error()` et `critical()`, qui appelle en fait les méthodes du même nom de l'objet *root logger*. Les fonctions et les méthodes ont la même signature. Le nom du *root logger* est affiché comme « *root* » dans la sortie.

It is, of course, possible to log messages to different destinations. Support is included in the package for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, or OS-specific logging mechanisms

such as syslog or the Windows NT event log. Destinations are served by *handler* classes. You can create your own log destination class if you have special requirements not met by any of the built-in handler classes.

Par défaut, aucune destination n'est prédéfinie pour les messages de journalisation. Vous pouvez définir une destination (comme la console ou un fichier) en utilisant `basicConfig()` comme dans les exemples donnés dans le tutoriel. Si vous appelez les fonctions `debug()`, `info()`, `warning()`, `error()` et `critical()`, celles-ci vérifient si une destination a été définie ; si ce n'est pas le cas, la destination est assignée à la console (`sys.stderr`) avec un format par défaut pour le message affiché, avant d'être déléguée au *logger* racine, qui sort le message.

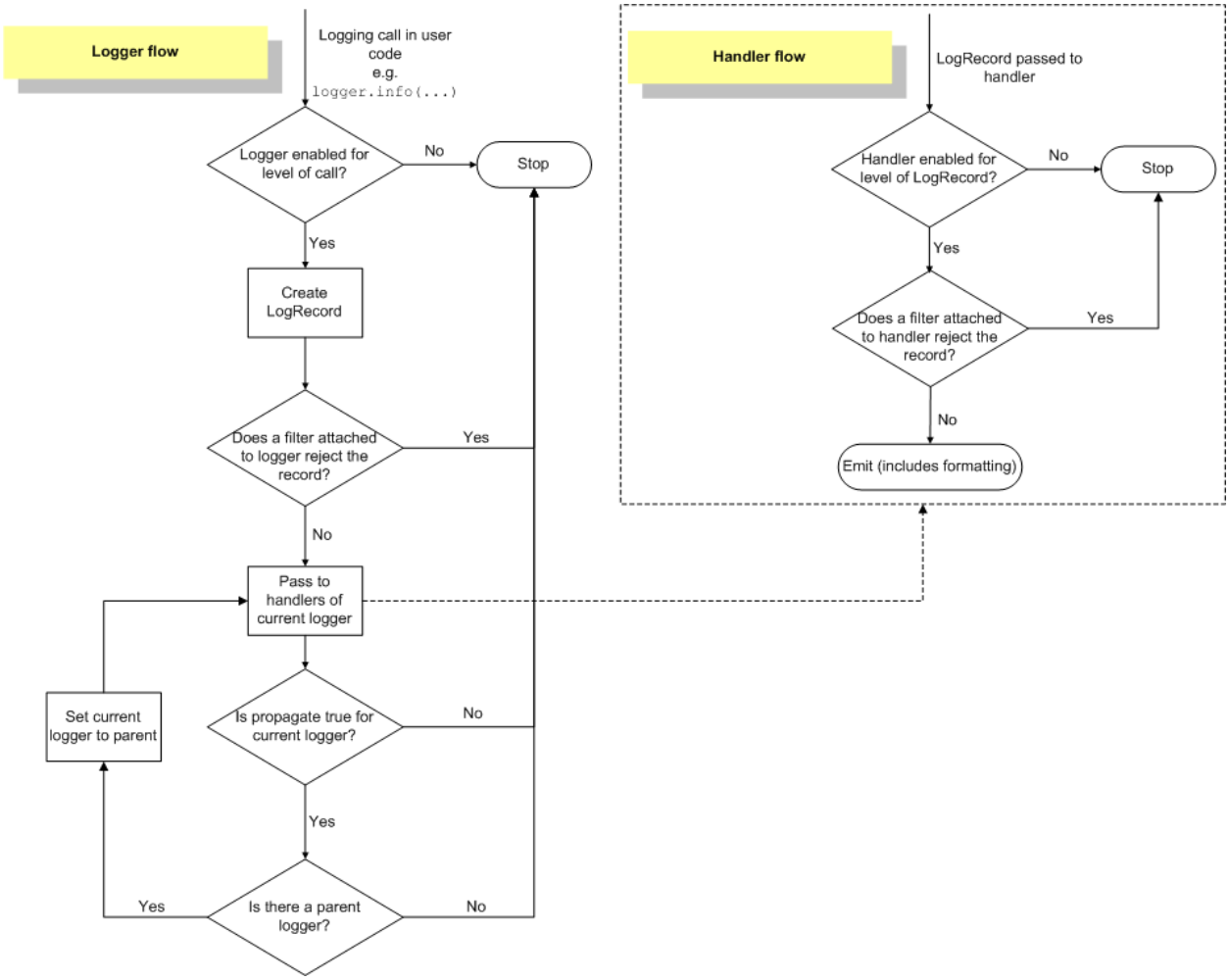
The default format set by `basicConfig()` for messages is :

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

Vous pouvez modifier ce comportement en passant une chaîne de formatage à `basicConfig()` par l'argument nommé *format*. Consultez formatter-objects pour toutes les options de construction de cette chaîne de formatage.

2.1 Flux du processus de journalisation

Le flux des informations associées à un évènement dans les *loggers* et les *handlers* est illustré dans le diagramme suivant.



2.2 Loggers

Les objets de classe `Logger` ont un rôle triple. Premièrement, ils exposent plusieurs méthodes au code de l'application, de manière à ce qu'elle puisse enregistrer des messages en cours d'exécution. Deuxièmement, les objets *logger* déterminent sur quel message agir selon leur sévérité (à partir des filtres par défaut) ou selon les objets *filter* associés. Troisièmement, les objets *logger* transmettent les messages pertinents à tous les *handlers* concernés.

Les méthodes des objets *logger* les plus utilisées appartiennent à deux catégories : la configuration et l'envoi de messages.

Voici les méthodes de configuration les plus communes :

- `Logger.setLevel()` spécifie le plus bas niveau de sévérité qu'un *logger* traitera. Ainsi, *debug* est le niveau de sévérité défini par défaut le plus bas et *critical* est le plus haut. Par exemple, si le niveau de sévérité est INFO, le *logger* ne traite que les messages de niveau INFO, WARNING, ERROR et CRITICAL ; il ignore les messages de niveau DEBUG.
- `Logger.addHandler()` et `Logger.removeHandler()` ajoutent ou enlèvent des objets *handlers* au *logger*. Les objets *handlers* sont expliqués plus en détail dans *Handlers*.
- `Logger.addFilter()` et `Logger.removeFilter()` ajoutent ou enlèvent des objets *filter* au *logger*. Les objets *filters* sont expliqués plus en détail dans filter.

Comme nous l'expliquons aux deux derniers paragraphes de cette section, vous n'avez pas besoin de faire appel à ces méthodes à chaque fois que vous créez un *logger*.

Une fois que l'objet *logger* est correctement configuré, les méthodes suivantes permettent de créer un message :

- `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()`, and `Logger.critical()` all create log records with a message and a level that corresponds to their respective method names. The message is actually a format string, which may contain the standard string substitution syntax of `%s`, `%d`, `%f`, and so on. The rest of their arguments is a list of objects that correspond with the substitution fields in the message. With regard to `**kwargs`, the logging methods care only about a keyword of `exc_info` and use it to determine whether to log exception information.
- `Logger.exception()` crée un message similaire à `Logger.error()`. La différence est que `Logger.exception()` ajoute la trace de la pile d'exécution au message. On ne peut appeler cette méthode qu'à l'intérieur d'un bloc de gestion d'exception.
- `Logger.log()` prend le niveau de sévérité comme argument explicite. C'est un peu plus verbeux pour enregistrer des messages que d'utiliser les méthodes plus pratiques décrites si dessus, mais c'est ce qui permet d'enregistrer des messages pour des niveaux de sévérité définis par l'utilisateur.

`getLogger()` renvoie une référence à un objet *logger* du nom spécifié si celui-ci est donné en argument. Dans le cas contraire, se sera le *logger root*. Ces noms sont des structures hiérarchiques séparées par des points. Des appels répétés à `getLogger()` avec le même nom renvoient une référence au même objet *logger*. Les *loggers* qui sont plus bas dans cette liste hiérarchique sont des enfants des *loggers* plus haut dans la liste. Par exemple, si un *logger* a le nom `foo`, les *loggers* avec les noms `foo.bar`, `foo.bar.baz`, et `foo.bam` sont tous des descendants de `foo`.

On associe aux *loggers* un concept de *niveau effectif*. Si aucun niveau n'est explicitement défini pour un *logger*, c'est le niveau du parent qui est utilisé comme niveau effectif. Si le parent n'a pas de niveau défini, c'est celui de *son* parent qui est considéré, et ainsi de suite ; on examine tous les ancêtres jusqu'à ce qu'un niveau explicite soit trouvé. Le *logger root* a toujours un niveau explicite (`WARNING` par défaut). Quand le *logger* traite un événement, c'est ce niveau effectif qui est utilisé pour déterminer si cet événement est transmis à ses *handlers*.

Les *loggers* fils font remonter leurs messages aux *handlers* associés à leurs *loggers* parents. De ce fait, il n'est pas nécessaire de définir et configurer des *handlers* pour tous les *loggers* employés par une application. Il suffit de configurer les *handlers* pour un *logger* de haut niveau et de créer des *loggers* fils quand c'est nécessaire (on peut cependant empêcher la propagation aux ancêtres des messages en donnant la valeur `False` à l'attribut *propagate* d'un *logger*).

2.3 Handlers

Les objets de type `Handler` sont responsables de la distribution des messages (selon leur niveau de sévérité) vers les destinations spécifiées pour ce *handler*. Les objets `Logger` peuvent ajouter des objets *handler* à eux-mêmes en appelant

`addHandler()`. Pour donner un exemple, une application peut envoyer tous les messages dans un fichier journal, tous les messages de niveau *error* ou supérieur vers la sortie standard, et tous les messages de niveau *critical* vers une adresse de courriel. Dans ce scenario, nous avons besoin de trois *handlers*, responsable chacun d'envoyer des messages d'une sévérité donnée vers une destination donnée.

La bibliothèque standard inclut déjà un bon nombre de types de gestionnaires (voir *Gestionnaires utiles*) ; le tutoriel utilise surtout `StreamHandler` et `FileHandler` dans ses exemples.

Peu de méthodes des objets *handlers* sont intéressantes pour les développeurs. Les seules méthodes intéressantes lorsqu'on utilise les objets *handlers* natifs (c'est à dire si l'on ne crée pas d'"*handler* personnalisé) sont les méthodes de configuration suivantes :

- La méthode `setLevel()`, comme celle des objets *logger* permet de spécifier le plus bas niveau de sévérité qui sera distribué à la destination appropriée. Pourquoi y a-t-il deux méthodes `setLevel()` ? Le niveau défini dans le *logger* détermine quelle sévérité doit avoir un message pour être transmis à ses *handlers*. Le niveau mis pour chaque *handler* détermine quels messages seront envoyés aux destinations.
- `setFormatter()` sélectionne l'objet *Formatter* utilisé par cet *handler*.
- `addFilter()` et `removeFilter()` configurent et respectivement dé-configurent des objets *filter* sur les *handlers*.

Le code d'une application ne devrait ni instancier, ni utiliser d'instances de la classe `Handler`. La classe `Handler` est plutôt d'une classe de base qui définit l'interface que tous les gestionnaires doivent avoir et établit les comportements par défaut que les classes filles peuvent employer (ou redéfinir).

2.4 Formatters

Formatter objects configure the final order, structure, and contents of the log message. Unlike the base `logging.Handler` class, application code may instantiate formatter classes, although you could likely subclass the formatter if your application needs special behavior. The constructor takes two optional arguments – a message format string and a date format string.

`logging.Formatter.__init__`(*fmt=None*, *datefmt=None*)

If there is no message format string, the default is to use the raw message. If there is no date format string, the default date format is :

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

with the milliseconds tacked on at the end.

The message format string uses `%(<dictionary key>)s` styled string substitution ; the possible keys are documented in logrecord-attributes.

La chaîne de formatage de message suivante enregistrera le temps dans un format lisible par les humains, la sévérité du message et son contenu, dans cet ordre :

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

Les *formatters* emploient une fonction configurable par l'utilisateur pour convertir le temps de création d'une entrée de journal en un tuple. Par défaut, `time.localtime()` est employé ; pour changer cela pour une instance particulière de *formatter*, assignez une fonction avec la même signature que `time.localtime()` ou `time.gmtime()` à l'attribut `converter` de cette instance. Pour changer cela pour tous les *formatters*, par exemple si vous voulez que tous votre horodatage soit affiché en GMT, changez l'attribut `converter` de la classe `Formatter` en `time.gmtime`.

2.5 Configuration de *Logging*

On peut configurer *logging* de trois façons :

1. Créer des *loggers*, *handlers* et *formatters* explicitement en utilisant du code Python qui appelle les méthodes de configuration listées ci-dessus.
2. Creating a logging config file and reading it using the `fileConfig()` function.
3. Créer un dictionnaire d'informations de configuration et le passer à la fonction `dictConfig()`.

Pour la documentation de référence de ces deux dernières options, voyez logging-config-api. L'exemple suivant configure un *logger* très simple, un *handler* employant la console, et un *formatter* simple en utilisant du code Python :

```
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.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

L'exécution de ce module via la ligne de commande produit la sortie suivante :

```
$ 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
```

Le module Python suivant crée un *logger*, un *handler* et un *formatter* identiques à ceux de l'exemple détaillé au-dessus, au nom des objets près :

```
import logging
import logging.config

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

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

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

(suite sur la page suivante)

(suite de la page précédente)

```
logger.error('error message')
logger.critical('critical message')
```

Here is the logging.conf file :

```
[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=
```

La sortie est presque identique à celle de l'exemple qui n'est pas basé sur un fichier de configuration :

```
$ 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
```

Vous pouvez constatez les avantages de l'approche par fichier de configuration par rapport à celle du code Python, principalement la séparation de la configuration et du code, et la possibilité pour une personne qui ne code pas de modifier facilement les propriétés de *logging*.

Avertissement : The `fileConfig()` function takes a default parameter, `disable_existing_loggers`, which defaults to `True` for reasons of backward compatibility. This may or may not be what you want, since it will cause any loggers existing before the `fileConfig()` call to be disabled unless they (or an ancestor) are explicitly named in the configuration. Please refer to the reference documentation for more information, and specify `False` for this parameter if you wish.

The dictionary passed to `dictConfig()` can also specify a Boolean value with key `disable_existing_loggers`, which if not specified explicitly in the dictionary also defaults to being

interpreted as `True`. This leads to the logger-disabling behaviour described above, which may not be what you want - in which case, provide the key explicitly with a value of `False`.

Notez que les noms de classe référencés dans le fichier de configuration doivent être relatifs au module *logging*, ou des valeurs absolues qui peuvent être résolues à travers les mécanismes d'importation habituels. Ainsi, on peut soit utiliser `WatchedFileHandler` (relativement au module *logging*) ou `mypackage.mymodule.MyHandler` (pour une classe définie dans le paquet `mypackage` et le module `mymodule`, si `mypackage` est disponible dans les chemins d'importation de Python).

In Python 2.7, a new means of configuring logging has been introduced, using dictionaries to hold configuration information. This provides a superset of the functionality of the config-file-based approach outlined above, and is the recommended configuration method for new applications and deployments. Because a Python dictionary is used to hold configuration information, and since you can populate that dictionary using different means, you have more options for configuration. For example, you can use a configuration file in JSON format, or, if you have access to YAML processing functionality, a file in YAML format, to populate the configuration dictionary. Or, of course, you can construct the dictionary in Python code, receive it in pickled form over a socket, or use whatever approach makes sense for your application.

Here's an example of the same configuration as above, in YAML format for the new dictionary-based approach :

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

Pour plus d'informations sur la journalisation à l'aide d'un dictionnaire, consultez logging-config-api.

2.6 Comportement par défaut (si aucune configuration n'est fournie)

Si aucune configuration de journalisation n'est fournie, il est possible d'avoir une situation où un événement doit faire l'objet d'une journalisation, mais où aucun gestionnaire ne peut être trouvé pour tracer l'événement. Le comportement du paquet `logging` dans ces circonstances dépend de la version Python.

For Python 2.x, the behaviour is as follows :

— Si *logging.raiseExceptions* vaut `False` (mode production), l'événement est silencieusement abandonné.

— Si *logging.raiseExceptions* vaut `True` (mode de développement), un message *No handlers could be found for logger X.Y.Z* est écrit sur la sortie standard une fois.

2.7 Configuration de la journalisation pour une bibliothèque

When developing a library which uses logging, you should take care to document how the library uses logging - for example, the names of loggers used. Some consideration also needs to be given to its logging configuration. If the using application does not configure logging, and library code makes logging calls, then (as described in the previous section) an error message will be printed to `sys.stderr`.

If for some reason you *don't* want this message printed in the absence of any logging configuration, you can attach a do-nothing handler to the top-level logger for your library. This avoids the message being printed, since a handler will be always be found for the library's events : it just doesn't produce any output. If the library user configures logging for application use, presumably that configuration will add some handlers, and if levels are suitably configured then logging calls made in library code will send output to those handlers, as normal.

A do-nothing handler is included in the logging package : `NullHandler` (since Python 2.7). An instance of this handler could be added to the top-level logger of the logging namespace used by the library (*if* you want to prevent an error message being output to `sys.stderr` in the absence of logging configuration). If all logging by a library *foo* is done using loggers with names matching "foo.x", "foo.x.y", etc. then the code :

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

doit avoir l'effet désiré. Si une organisation produit un certain nombre de bibliothèques, le nom de l'enregistreur spécifié peut être `orgname.foo` plutôt que simplement `foo`.

Note : Il est vivement conseillé de ne *pas ajouter de gestionnaires autres que* `NullHandler` *aux enregistreurs de votre bibliothèque*. Cela est dû au fait que la configuration des gestionnaires est la prérogative du développeur d'applications qui utilise votre bibliothèque. Le développeur d'applications connaît le public cible et les gestionnaires les plus appropriés pour ses applications : si vous ajoutez des gestionnaires « sous le manteau », vous pourriez bien interférer avec les tests unitaires et la journalisation qui convient à ses exigences.

---

3 Niveaux de journalisation

Les valeurs numériques des niveaux de journalisation sont données dans le tableau suivant. Celles-ci n'ont d'intérêt que si vous voulez définir vos propres niveaux, avec des valeurs spécifiques par rapport aux niveaux prédéfinis. Si vous définissez un niveau avec la même valeur numérique, il écrase la valeur prédéfinie ; le nom prédéfini est perdu.

Niveau	Valeur numérique
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

Les niveaux peuvent également être associés à des enregistreurs, étant définis soit par le développeur, soit par le chargement d'une configuration de journalisation enregistrée. Lorsqu'une méthode de journalisation est appelée sur un enregistreur, l'enregistreur compare son propre niveau avec le niveau associé à l'appel de méthode. Si le niveau de l'enregistreur est supérieur à l'appel de méthode, aucun message de journalisation n'est réellement généré. C'est le mécanisme de base contrôlant la verbosité de la sortie de journalisation.

Les messages de journalisation sont codés en tant qu'instances de `LogRecord`. Lorsqu'un enregistreur décide de réellement enregistrer un événement, une instance de `LogRecord` est créée à partir du message de journalisation.

Les messages de journalisation sont soumis à un mécanisme d'expédition via l'utilisation de *handlers*, qui sont des instances de sous-classes de la classe `Handler`. Les gestionnaires sont chargés de s'assurer qu'un message journalisé (sous la forme d'un `LogRecord`) atterrit dans un emplacement particulier (ou un ensemble d'emplacements) qui est utile pour le public cible pour ce message (tels que les utilisateurs finaux, le personnel chargé de l'assistance aux utilisateurs, les administrateurs système ou les développeurs). Des instances de `LogRecord` adaptées à leur destination finale sont passées aux gestionnaires destinées à des destinations particulières. Chaque enregistreur peut avoir zéro, un ou plusieurs gestionnaires associés à celui-ci (via la méthode `addHandler()` de `Logger`). En plus de tous les gestionnaires directement associés à un enregistreur, *tous les gestionnaires associés à tous les ancêtres de l'enregistreur* sont appelés pour envoyer le message (à moins que l'indicateur *propager* pour un enregistreur soit défini sur la valeur `False`, auquel cas le passage à l'ancêtre gestionnaires s'arrête).

Tout comme pour les enregistreurs, les gestionnaires peuvent avoir des niveaux associés. Le niveau d'un gestionnaire agit comme un filtre de la même manière que le niveau d'un enregistreur. Si un gestionnaire décide de réellement distribuer un événement, la méthode `emit()` est utilisée pour envoyer le message à sa destination. La plupart des sous-classes définies par l'utilisateur de `Handler` devront remplacer ce `emit()`.

3.1 Niveaux personnalisés

La définition de vos propres niveaux est possible, mais ne devrait pas être nécessaire, car les niveaux existants ont été choisis par expérience. Cependant, si vous êtes convaincu que vous avez besoin de niveaux personnalisés, prenez grand soin à leur réalisation et il est pratiquement certain que c'est *une très mauvaise idée de définir des niveaux personnalisés si vous développez une bibliothèque*. Car si plusieurs auteurs de bibliothèque définissent tous leurs propres niveaux personnalisés, il y a une chance que la sortie de journalisation de ces multiples bibliothèques utilisées ensemble sera difficile pour le développeur à utiliser pour contrôler et/ou interpréter, car une valeur numérique donnée peut signifier des choses différentes pour différentes bibliothèques.

4 Gestionnaires utiles

En plus de la classe de base `Handler`, de nombreuses sous-classes utiles sont fournies :

1. Les instances `StreamHandler` envoient des messages aux flux (objets de type fichier).
2. Les instances `FileHandler` envoient des messages à des fichiers sur le disque.
3. `BaseRotatingHandler` est la classe de base pour les gestionnaires qui assurent la rotation des fichiers de journalisation à partir d'un certain point. Elle n'est pas destinée à être instanciée directement. Utilisez plutôt `RotatingFileHandler` ou `TimedRotatingFileHandler`.
4. Les instances `RotatingFileHandler` envoient des messages à des fichiers sur le disque, avec la prise en charge des tailles maximales de fichiers de journalisation et de la rotation des fichiers de journalisation.
5. `TimedRotatingFileHandler` instances send messages to disk files, rotating the log file at certain timed intervals.
6. `SocketHandler` instances send messages to TCP/IP sockets.
7. `DatagramHandler` instances send messages to UDP sockets.
8. Les instances de `SMTPHandler` envoient des messages à une adresse e-mail désignée.
9. Les instances de `SysLogHandler` envoient des messages à un *daemon syslog* UNIX, éventuellement sur un ordinateur distant.
10. Les instances de `NTEventLogHandler` envoient des messages à un journal des événements Windows NT/2000/XP.
11. Les instances de `MemoryHandler` envoient des messages à un tampon en mémoire, qui est vidé chaque fois que des critères spécifiques sont remplis.
12. Les instances de `HTTPHandler` envoient des messages à un serveur HTTP à l'aide de la sémantique `GET` ou `POST`.

13. Les instances de `WatchedFileHandler` surveillent le fichier sur lequel elles se connectent. Si le fichier change, il est fermé et rouvert à l'aide du nom de fichier. Ce gestionnaire n'est utile que sur les systèmes de type UNIX ; Windows ne prend pas en charge le mécanisme sous-jacent utilisé.
14. Les instances `NullHandler` ne font rien avec les messages d'erreur. Ils sont utilisés par les développeurs de bibliothèques qui veulent utiliser la journalisation, mais qui veulent éviter les messages de type *No handlers could be found for logger XXX*, affiché si celui qui utilise la bibliothèque n'a pas configuré la journalisation. Voir *Configuration de la journalisation pour une bibliothèque* pour plus d'informations.

Nouveau dans la version 2.7 : La classe `NullHandler`.

The `NullHandler`, `StreamHandler` and `FileHandler` classes are defined in the core logging package. The other handlers are defined in a sub- module, `logging.handlers`. (There is also another sub-module, `logging.config`, for configuration functionality.)

Les messages journalisés sont mis en forme pour la présentation via des instances de la classe `Formatter`. Ils sont initialisés avec une chaîne de format appropriée pour une utilisation avec l'opérateur % et un dictionnaire.

Pour formater plusieurs messages dans un lot, des instances de `BufferingFormatter` peuvent être utilisées. En plus de la chaîne de format (qui est appliquée à chaque message dans le lot), il existe des dispositions pour les chaînes de format d'en-tête et de fin.

Lorsque le filtrage basé sur le niveau de l'enregistreur et/ou le niveau du gestionnaire ne suffit pas, les instances de `Filter` peuvent être ajoutées aux deux instances de `Logger` et `Handler` (par le biais de leur méthode `addFilter()`). Avant de décider de traiter un message plus loin, les enregistreurs et les gestionnaires consultent tous leurs filtres pour obtenir l'autorisation. Si un filtre renvoie une valeur `False`, le traitement du message est arrêté.

La fonctionnalité de base `Filter` permet de filtrer par nom de *logger* spécifique. Si cette fonctionnalité est utilisée, les messages envoyés à l'enregistreur nommé et à ses enfants sont autorisés via le filtre et tous les autres sont abandonnés.

5 Exceptions levées par la journalisation

Le paquet de journalisation est conçu pour ne pas faire apparaître les exceptions qui se produisent lors de la journalisation en production. Il s'agit de sorte que les erreurs qui se produisent lors de la gestion des événements de journalisation (telles qu'une mauvaise configuration de la journalisation , une erreur réseau ou d'autres erreurs similaires) ne provoquent pas l'arrêt de l'application utilisant la journalisation.

Les exceptions `SystemExit` et `KeyboardInterrupt` ne sont jamais passées sous silence. Les autres exceptions qui se produisent pendant la méthode `emit()` d'une sous classe `Handler` sont passées à sa méthode `handleError()`.

L'implémentation par défaut de `handleError()` dans la classe `Handler` vérifie si une variable au niveau du module, `raiseExceptions`, est définie. Si cette valeur est définie, la trace de la pile d'appels est affichée sur `sys.stderr`. Si elle n'est pas définie, l'exception est passée sous silence.

Note : La valeur par défaut de `raiseExceptions` est `True`. C'est parce que pendant le développement, vous voulez généralement être notifié de toutes les exceptions qui se produisent. Il est conseillé de définir `raiseExceptions` à `False` pour une utilisation en production.

---

6 Utilisation d'objets arbitraires comme messages

Dans les sections et exemples précédents, il a été supposé que le message passé lors de la journalisation de l'événement est une chaîne. Cependant, ce n'est pas la seule possibilité. Vous pouvez passer un objet arbitraire en tant que message et sa méthode `__str__()` est appelée lorsque le système de journalisation doit le convertir en une représentation sous forme de chaîne. En fait, si vous le souhaitez, vous pouvez complètement éviter de calculer une représentation sous forme de

chaîne. Par exemple, les gestionnaires `SocketHandler` émettent un événement en lui appliquant *pickle* et en l'envoyant sur le réseau.

7 Optimisation

La mise en forme des arguments de message est différée jusqu'à ce qu'elle ne puisse pas être évitée. Toutefois, le calcul des arguments passés à la méthode de journalisation peut également être coûteux et vous voudrez peut-être éviter de le faire si l'enregistreur va simplement jeter votre événement. Pour décider de ce qu'il faut faire, vous pouvez appeler la méthode `isEnabledFor()` qui prend en argument le niveau et renvoie `True` si un événement est créé par l'enregistreur pour ce niveau d'appel. Vous pouvez écrire un code qui ressemble à ça :

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

de sorte que si le seuil du journaliseur est défini au-dessus de `DEBUG`, les appels à `expensive_func1()` et `expensive_func2()` ne sont jamais faits.

Note : Dans certains cas, `isEnabledFor()` peut être plus coûteux que vous le souhaitez (par exemple pour les enregistreurs profondément imbriqués où un niveau explicite n'est défini que dans la hiérarchie des enregistreurs). Dans de tels cas (ou si vous souhaitez éviter d'appeler une méthode dans des boucles optimisées), vous pouvez mettre en cache le résultat d'un appel à `isEnabledFor()` dans une variable locale ou d'instance, et l'utiliser au lieu d'appeler la méthode à chaque fois. Une telle valeur mise en cache ne doit être recalculée que lorsque la configuration de journalisation change dynamiquement pendant l'exécution de l'application (ce qui est rarement le cas).

Il existe d'autres optimisations qui peuvent être faites pour des applications spécifiques qui nécessitent un contrôle plus précis sur les informations de journalisation collectées. Voici une liste de choses que vous pouvez faire pour éviter le traitement pendant la journalisation dont vous n'avez pas besoin :

Ce que vous ne voulez pas collecter	Comment éviter de le collecter
Informations sur l'endroit où les appels ont été faits.	Set `logging._srcfile` to `None`. This avoids calling `sys._getframe()`, which may help to speed up your code in environments like PyPy (which can't speed up code that uses `sys._getframe()`).
Informations de *threading*.	Mettez `logging.logThreads` à `0`.
Informations sur le processus.	Mettez `logging.logProcesses` à `0`.

Notez également que le module de journalisation principale inclut uniquement les gestionnaires de base. Si vous n'importez pas `logging.handlers` et `logging.config`, ils ne prendront pas de mémoire.

Voir aussi :

Module `logging` Référence d'API pour le module de journalisation.

Module `logging.config` API de configuration pour le module de journalisation.

Module `logging.handlers` Gestionnaires utiles inclus avec le module de journalisation.

A logging cookbook

Index

Non-alphabetical

__init__() (*méthode logging.logging.Formatter*), 9