
Tutoriel sur la journalisation

Version 3.10.18

Guido van Rossum
and the Python development team

juillet 08, 2025

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 événements 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	9
2.4	Formatters	10
2.5	Configuration de logging	10
2.6	Comportement par défaut (si aucune configuration n'est fournie)	13
2.7	Configuration de la journalisation pour une bibliothèque	14
3	Niveaux de journalisation	14
3.1	Niveaux personnalisés	15
4	Gestionnaires utiles	15
5	Exceptions levées par la journalisation	16
6	Utilisation d'objets arbitraires comme messages	17
7	Optimisation	17
	Index	19

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	<code>print()</code>
Rapporter des événements qui ont lieu au cours du fonctionnement normal d'un programme (par exemple pour suivre un statut ou examiner des dysfonctionnements)	<code>logging.info()</code> (ou <code>logging.debug()</code> pour une sortie très détaillée à visée diagnostique)
Émettre un avertissement (<i>warning</i> en anglais) en relation avec un événement particulier au cours du fonctionnement d'un programme	<code>warnings.warn()</code> dans le code de la bibliothèque si le problème est évitable et l'application cliente doit être modifiée pour éliminer cet avertissement <code>logging.warning()</code> 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 événement 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)	<code>logging.error()</code> , <code>logging.exception()</code> ou <code>logging.critical()</code> , 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 événements 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 événements de ce niveau et au-dessus sont suivis, sauf si le paquet *logging* est configuré pour faire autrement.

Les événements 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'événement 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 événements dans un fichier

A very common situation is that of recording logging events in a file, so let's look at that next. Be sure to try the following in a newly started Python interpreter, and don't just continue from the session described above :

```
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')
```

Modifié dans la version 3.9 : L'argument *encoding* à été rajouté. Dans les versions précédentes de Python, ou si non spécifié, l'encodage utilisé est la valeur par défaut utilisée par `open()`. Bien que non montré dans l'exemple au dessus, un argument **errors** peut aussi être passé, qui détermine comment les erreurs d'encodage sont gérées. Pour voir les valeurs disponibles et par défaut, consultez la documentation de `open()`.

Maintenant, si nous ouvrons le fichier et lisons ce qui s'y trouve, on trouvera les messages de log :

```
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
```

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.

Si vous souhaitez régler le niveau de journalisation à partir d'une option de la ligne de commande comme :

```
--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, ...)
```

The call to `basicConfig()` should come *before* any calls to `debug()`, `info()`, etc. Otherwise, those functions will call `basicConfig()` for you with the default options. As it's intended as a one-off simple configuration facility, only the first call will actually do anything : subsequent calls are effectively no-ops.

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 *filename*, 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')
```

Si vous exécutez *myapp.py*, vous verrez ceci dans *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, référez-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!
```

Comme vous pouvez le voir, l'inclusion des données variables dans le message de description de l'évènement emploie le vieux style de formatage avec %. C'est pour assurer la rétrocompatibilité : le module `logging` est antérieur aux nouvelles options de formatage comme `str.format()` ou `string.Template`. Ces nouvelles options de formatage sont gérées, mais leur exploration sort du cadre de ce tutoriel, voyez `formatting-styles` pour plus d'information.

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')
```

ce qui affiche :

```
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)s ' dans votre chaîne de formatage :

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

ce qui affichera quelque chose comme :

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

Le format par défaut de l'horodatage (comme ci-dessus) est donné par la norme ISO8601 ou **RFC 3339**. Pour plus de contrôle sur le formatage de l'horodatage, vous pouvez fournir à `basicConfig` un argument *datefmt*, comme dans l'exemple suivant :

```
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.')
```

ce qui affichera quelque chose comme :

```
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'autres 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.

Si vos besoins avec `logging` sont simples, vous pouvez incorporer les exemples ci-dessus dans vos scripts. Si vous rencontrez des difficultés ou s'il y a quelque chose que vous ne comprenez pas, vous pouvez poser une question sur le groupe Usenet `comp.lang.python` (accessible à <https://groups.google.com/forum/#!forum/comp.lang.python>), on vous répondra rapidement.

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 (créés 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 enregistreurs 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.

Il est bien sûr possible d'enregistrer des messages pour des destinations différentes. Ce paquet permet d'écrire des entrées de journal dans des fichiers, des ressources HTTP GET/POST, par courriel via SMTP, des connecteurs (*socket* en anglais) génériques, des files d'attente, ou des mécanismes d'enregistrement spécifiques au système d'exploitation, comme *syslog* ou le journal d'événements de Windows NT. Les destinations sont servies par des classes *handler*. Vous pouvez créer votre propre classe de destination si vous avez des besoins spéciaux qui ne sont couverts par aucune classe *handler* prédéfinie.

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.

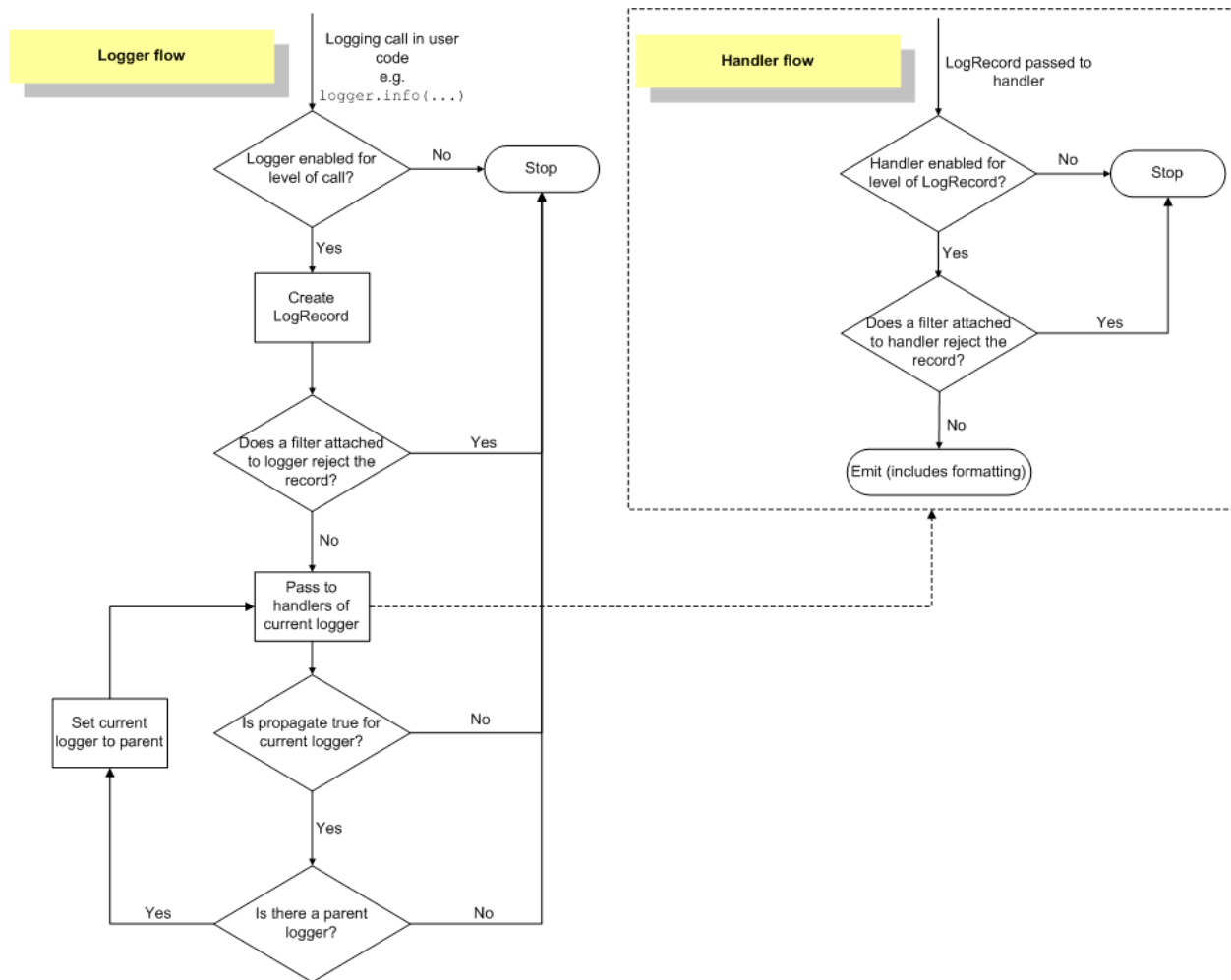
Le format par défaut des messages est défini par `basicConfig()` comme suit :

```
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 événement 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 :

- Les méthodes `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()` et `Logger.critical()` créent des entrées de journal avec un message et un niveau correspondant à leur nom. Le message est en fait une chaîne de caractères qui peut contenir la syntaxe standard de substitution de chaînes de caractères : `%s`, `%d`, `%f`, etc. L'argument suivant est une liste des objets correspondant aux champs à substituer dans le message. En ce qui concerne `**kwargs`, les méthodes de logging ne tiennent compte que du mot clef `exc_info` et l'utilisent pour déterminer s'il faut enregistrer les informations associées à une exception.
- `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 ci-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 scénario, 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 de *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 ce *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

Les objets *formatter* configurent l'ordre final, la structure et le contenu du message. Contrairement à la classe de base `logging.Handler`, le code d'une application peut instancier un objet de classe *formatter*, même si vous pouvez toujours sous-classer *formatter* si vous avez besoin d'un comportement spécial dans votre application. Le constructeur a trois arguments optionnels : une chaîne de formatage du message, un chaîne de formatage de la date et un indicateur de style.

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

S'il n'y a pas de chaîne de formatage, la chaîne brute est utilisée par défaut. S'il n'y a pas de chaîne de formatage de date, le format de date par défaut est :

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

with the milliseconds tacked on at the end. The *style* is one of `'%'`, `'{'`, or `'$'`. If one of these is not specified, then `'%'` will be used.

If the style is `'%'`, the message format string uses `%(<dictionary key>)s` styled string substitution; the possible keys are documented in `logrecord-attributes`. If the style is `'{'`, the message format string is assumed to be compatible with `str.format()` (using keyword arguments), while if the style is `'$'` then the message format string should conform to what is expected by `string.Template.substitute()`.

Modifié dans la version 3.2 : Ajout du paramètre *style*.

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 *n*-uplet. 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. Créer un fichier de configuration de `logging` et le lire en employant la fonction `fileConfig()`.
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)
```

(suite sur la page suivante)

```
# 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')
```

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

Voici le fichier *logging.conf* :

```
[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

```

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 constater 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 : La fonction `fileConfig()` accepte un paramètre par défaut `disable_existing_loggers`, qui vaut `True` par défaut pour des raisons de compatibilité ascendante. Ce n'est pas forcément ce que vous souhaitez : en effet, tous les *loggers* créés avant l'appel à `fileConfig()` seront désactivés sauf si eux-mêmes (ou l'un de leurs parents) sont explicitement nommés dans le fichier de configuration. Veuillez vous reporter à la documentation pour plus de détails, et donner la valeur `False` à ce paramètre si vous le souhaitez.

Le dictionnaire passé à `dictConfig()` peut aussi spécifier une valeur Booléenne pour la clef `disable_existing_loggers`. Si cette valeur n'est pas donnée, elle est interprétée comme vraie par défaut. Cela conduit au comportement de désactivation des *loggers* décrit ci-dessus, qui n'est pas forcément celui que vous souhaitez ; dans ce cas, donnez explicitement la valeur `False` à cette clef.

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).

Dans Python 3.2, un nouveau moyen de configuration de la journalisation a été introduit, à l'aide de dictionnaires pour contenir les informations de configuration. Cela fournit un sur-ensemble de la fonctionnalité décrite ci-dessus basée sur un fichier de configuration et c'est la méthode recommandée pour les nouvelles applications et les déploiements. Étant donné qu'un dictionnaire Python est utilisé pour contenir des informations de configuration et que vous pouvez remplir ce dictionnaire à l'aide de différents moyens, vous avez plus d'options pour la configuration. Par exemple, vous pouvez utiliser

un fichier de configuration au format JSON ou, si vous avez accès à la fonctionnalité de traitement YAML, un fichier au format YAML, pour remplir le dictionnaire de configuration. Ou bien sûr, vous pouvez construire le dictionnaire dans le code Python, le recevoir sous forme de *pickle* sur un connecteur, ou utiliser n'importe quelle approche suivant la logique de votre application.

Voici un exemple définissant la même configuration que ci-dessus, au format YAML pour le dictionnaire correspondant à cette nouvelle approche :

```
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.

Pour les versions de Python antérieures à 3.2, le comportement est le suivant :

- 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.

Dans Python 3.2 et ultérieur, le comportement est le suivant :

- L'événement est sorti à l'aide d'un « gestionnaire de dernier recours », stocké dans `logging.lastResort`. Ce gestionnaire interne n'est associé à aucun enregistreur et agit comme un `StreamHandler` qui écrit le message de description de l'événement vers la valeur actuelle de `sys.stderr` (par conséquent, en respectant les redirections qui peuvent être en vigueur). Aucun formatage n'est fait sur le message – juste le message de description de l'événement nu est affiché. Le niveau du gestionnaire est défini sur `WARNING`, de sorte que tous les événements de cette sévérité et plus seront écrits.

Pour obtenir un comportement antérieur à 3.2, `logging.lastResort` peut être mis à `None`.

2.7 Configuration de la journalisation pour une bibliothèque

Lors du développement d'une bibliothèque qui utilise la journalisation, vous devez prendre soin de documenter la façon dont la bibliothèque utilise la journalisation (par exemple, les noms des enregistreurs utilisés). Consacrez aussi un peu de temps à la configuration de la journalisation. Si l'application utilisant votre bibliothèque n'utilise pas la journalisation et que le code de la bibliothèque effectue des appels de journalisation, alors (comme décrit dans la section précédente), les événements de gravité `WARNING` et au-dessus seront écrits sur `sys.stderr`. Cela est considéré comme le meilleur comportement par défaut.

Si, pour une raison quelconque, vous ne voulez *pas* que ces messages soient affichés en l'absence de toute configuration de journalisation, vous pouvez attacher un gestionnaire *ne-fait-rien* à l'enregistreur de niveau supérieur de votre bibliothèque. Cela évite qu'un message ne soit écrit, puisqu'un gestionnaire sera toujours trouvé pour les événements de la bibliothèque, il ne produit tout simplement pas de sortie. Si celui qui utilise la bibliothèque configure la journalisation pour son application, il est vraisemblable que la configuration ajoutera certains gestionnaires et, si les niveaux sont convenablement configurés, alors la journalisation des appels effectués dans le code de bibliothèque enverra la sortie à ces gestionnaires, comme d'habitude.

Un gestionnaire *ne-fait-rien* est inclus dans le paquet de journalisation : `NullHandler` (depuis Python 3.1). Une instance de ce gestionnaire peut être ajoutée à l'enregistreur de niveau supérieur de l'espace de nommage de journalisation utilisé par la bibliothèque (si vous souhaitez empêcher la copie de la journalisation de votre bibliothèque dans `sys.stderr` en l'absence de configuration de journalisation). Si toute la journalisation par une bibliothèque *foo* est effectuée en utilisant des enregistreurs avec des noms correspondant à *foo.x*, *foo.x.y*, etc., alors le 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 enregis-

treur, 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. Les instances de `TimedRotatingFileHandler` envoient des messages aux fichiers de disque, en permutant le fichier `journal` à intervalles réguliers.
6. Les instances de `SocketHandler` envoient des messages aux connecteurs TCP/IP. Depuis 3.4, les connecteurs UNIX sont également pris en charge.
7. Les instances de `DatagramHandler` envoient des messages aux connecteurs UDP. Depuis 3.4, les connecteurs UNIX sont également pris en charge.
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 de `QueueHandler` envoient des messages à une file d'attente, telles que celles implémentées dans les modules `queue` ou `multiprocessing`.
15. 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 3.1 : La classe `NullHandler`.

Nouveau dans la version 3.2 : La classe `QueueHandler`.

Les classes `NullHandler`, `StreamHandler` et `FileHandler` sont définies dans le module de journalisation de base. Les autres gestionnaires sont définis dans un sous-module, `logging.handlers` (il existe également un autre sous-module, `logging.config`, pour la fonctionnalité de configuration).

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.	Définissez <code>logging._srcfile</code> à <code>None</code> . Cela évite d'appeler <code>sys._getframe()</code> , qui peut aider à accélérer votre code dans des environnements comme PyPy (qui ne peut pas accélérer le code qui utilise <code>sys._getframe()</code>).
Informations de <i>threading</i> .	Mettez <code>logging.logThreads</code> à <code>False</code> .
Identifiant du processus courant (résultat de <code>os.getpid()</code>)	Mettez <code>logging.logProcesses</code> à <code>False</code> .
Nom du processus actuel, si vous vous servez de <code>multiprocessing</code> pour gérer plusieurs processus à la fois	Mettez <code>logging.logMultiProcessing</code> à <code>False</code> .

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 alphabétique

`__init__()` (méthode `logging.logging.Formatter`), [10](#)

R

RFC

RFC 3339, [6](#)