
Recettes pour la journalisation

Version 3.10.16

Guido van Rossum
and the Python development team

décembre 07, 2024

Python Software Foundation
Email : docs@python.org

Table des matières

1	Journalisation dans plusieurs modules	3
2	Journalisation avec des fils d'exécution multiples	4
3	Plusieurs gestionnaires et formateurs	5
4	Journalisation vers plusieurs destinations	6
5	Personnalisation du niveau de journalisation	7
6	Exemple d'un serveur de configuration	10
7	Utilisation de gestionnaires bloquants	11
8	Envoi et réception d'événements de journalisation à travers le réseau	12
8.1	Journalisation en production à l'aide d'un connecteur en écoute sur le réseau	14
9	Ajout d'informations contextuelles dans la journalisation	15
9.1	Utilisation d'adaptateurs de journalisation pour transmettre des informations contextuelles	15
9.2	Utilisation de filtres pour transmettre des informations contextuelles	16
10	Use of <code>contextvars</code>	17
11	Imparting contextual information in handlers	21
12	Journalisation vers un fichier unique à partir de plusieurs processus	22
12.1	Utilisation de <code>concurrent.futures.ProcessPoolExecutor</code>	26
12.2	Déploiement d'applications Web avec <i>Gunicorn</i> et <i>uWSGI</i>	26
13	Utilisation du roulement de fichiers	26
14	Utilisation d'autres styles de formatage	27
15	Personnalisation de <code>LogRecord</code>	30
16	Dérivation de <code>QueueHandler</code> – un exemple de <i>ZeroMQ</i>	31
17	Dérivation de <code>QueueListener</code> – un exemple de <i>ZeroMQ</i>	31

18 Exemple de configuration basée sur un dictionnaire	32
19 Utilisation d'un rotateur et d'un nom pour personnaliser la rotation des journaux	33
20 Exemple plus élaboré avec traitement en parallèle	34
21 Insertion d'une BOM dans les messages envoyés à un SysLogHandler	37
22 Journalisation structurée	38
23 Personnalisation des gestionnaires avec dictConfig()	39
24 Using particular formatting styles throughout your application	41
24.1 Using LogRecord factories	42
24.2 Using custom message objects	42
25 Configuring filters with dictConfig()	43
26 Customized exception formatting	44
27 Speaking logging messages	45
28 Buffering logging messages and outputting them conditionally	46
29 Sending logging messages to email, with buffering	48
30 Formatting times using UTC (GMT) via configuration	49
31 Using a context manager for selective logging	50
32 A CLI application starter template	52
33 A Qt GUI for logging	54
34 Logging to syslog with RFC5424 support	58
35 How to treat a logger like an output stream	59
36 Patterns to avoid	61
36.1 Opening the same log file multiple times	61
36.2 Using loggers as attributes in a class or passing them as parameters	62
36.3 Adding handlers other than NullHandler to a logger in a library	62
36.4 Creating a lot of loggers	62
37 Autres ressources	62
Index	63

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

Cette page contient des recettes relatives à la journalisation qui se sont avérées utiles par le passé. Pour des liens vers le tutoriel et des informations de référence, consultez *ces autres ressources*.

1 Journalisation dans plusieurs modules

Deux appels à `logging.getLogger('unLogger')` renvoient toujours une référence vers le même objet de journalisation. C'est valable à l'intérieur d'un module, mais aussi dans des modules différents pour autant que ce soit le même processus de l'interpréteur Python. En plus, le code d'une application peut définir et configurer une journalisation parente dans un module et créer (mais pas configurer) une journalisation fille dans un module séparé. Les appels à la journalisation fille passeront alors à la journalisation parente. Voici un module principal :

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s
↪')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

Voici un module auxiliaire :

```
import logging

# create logger
module_logger = logging.getLogger('spam_application.auxiliary')

class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

La sortie ressemble à ceci :

```

2005-03-23 23:47:11,663 - spam_application - INFO -
    creating an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary - INFO -
    creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
    created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
    calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary - INFO -
    doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary - INFO -
    done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
    finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
    calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
    received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
    done with auxiliary_module.some_function()

```

2 Journalisation avec des fils d'exécution multiples

La journalisation avec des fils d'exécution multiples ne requiert pas d'effort particulier. L'exemple suivant montre comment journaliser depuis le fil principal (c.-à-d. initial) et un autre fil :

```

import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%(relativeCreated)6d
↪ %(threadName)s %(message)s')
    info = {'stop': False}
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()

```

À l'exécution, le script doit afficher quelque chose comme ça :

```

0 Thread-1 Hi from myfunc
3 MainThread Hello from main
505 Thread-1 Hi from myfunc
755 MainThread Hello from main

```

(suite sur la page suivante)

```

1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc

```

Les entrées de journalisation sont entrelacées, comme on pouvait s'y attendre. Cette approche fonctionne aussi avec plus de fils que dans l'exemple, bien sûr.

3 Plusieurs gestionnaires et formateurs

Les gestionnaires de journalisation sont des objets Python ordinaires. La méthode `addHandler()` n'est pas limitée, en nombre minimum ou maximum, en gestionnaires que vous pouvez ajouter. Parfois, il peut être utile pour une application de journaliser tous les messages quels que soient leurs niveaux vers un fichier texte, tout en journalisant les erreurs (et plus grave) dans la console. Pour ce faire, configurez simplement les gestionnaires de manière adéquate. Les appels de journalisation dans le code de l'application resteront les mêmes. Voici une légère modification de l'exemple précédent dans une configuration au niveau du module :

```

import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s
↪')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

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

```

Notez que le code de « l'application » ignore la multiplicité des gestionnaires. Les modifications consistent simplement en l'ajout et la configuration d'un nouveau gestionnaire appelé *fh*.

La possibilité de créer de nouveaux gestionnaires avec des filtres sur un niveau de gravité supérieur ou inférieur peut être très utile lors de l'écriture ou du test d'une application. Au lieu d'utiliser de nombreuses instructions `print` pour le débogage, utilisez `logger.debug` : contrairement aux instructions `print`, que vous devrez supprimer ou commenter plus tard, les instructions `logger.debug` peuvent demeurer telles quelles dans le code source et

restent dormantes jusqu'à ce que vous en ayez à nouveau besoin. À ce moment-là, il suffit de modifier le niveau de gravité de la journalisation ou du gestionnaire pour déboguer.

4 Journalisation vers plusieurs destinations

Supposons que vous souhaitiez journaliser dans la console et dans un fichier avec différents formats de messages et avec différents critères. Supposons que vous souhaitiez consigner les messages de niveau DEBUG et supérieur dans le fichier, et les messages de niveau INFO et supérieur dans la console. Supposons également que le fichier doit contenir des horodatages, mais pas les messages de la console. Voici comment y parvenir :

```
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/tmp/myapp.log',
                    filemode='w')

# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

Quand vous le lancez, vous devez voir

```
root      : INFO      Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO      How quickly daft jumping zebras vex.
myapp.area2 : WARNING   Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR     The five boxing wizards jump quickly.
```

et, dans le fichier, vous devez trouver

```
10-22 22:19 root      INFO      Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1 DEBUG     Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1 INFO      How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2 WARNING   Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2 ERROR     The five boxing wizards jump quickly.
```

Comme vous pouvez le constater, le message DEBUG n'apparaît que dans le fichier. Les autres messages sont envoyés vers les deux destinations.

Cet exemple utilise la console et des gestionnaires de fichier, mais vous pouvez utiliser et combiner autant de gestionnaires que de besoin.

Notez que le choix du nom de fichier `journal /tmp/myapp.log` ci-dessus implique l'utilisation d'un emplacement standard pour les fichiers temporaires sur les systèmes POSIX. Sous Windows, vous devrez peut-être choisir un nom de répertoire différent pour le journal – assurez-vous simplement que le répertoire existe et que vous disposez des autorisations nécessaires pour créer et mettre à jour des fichiers dans celui-ci.

5 Personnalisation du niveau de journalisation

Il peut arriver que vous souhaitiez que vos gestionnaires journalisent légèrement différemment de la gestion standard des niveaux, où tous les niveaux au-dessus d'un seuil sont traités par un gestionnaire. Pour ce faire, vous devez utiliser des filtres. Examinons un scénario dans lequel vous souhaitez organiser les choses comme suit :

- envoyer les messages de niveau INFO et WARNING à `sys.stdout`;
- envoyer les messages de niveau ERROR et au-dessus à `sys.stderr`;
- envoyer les messages de niveau DEBUG et au-dessus vers le fichier `app.log`.

Supposons que vous configurez la journalisation avec le contenu au format JSON suivant :

```
{
  "version": 1,
  "disable_existing_loggers": false,
  "formatters": {
    "simple": {
      "format": "%(levelname)-8s - %(message)s"
    }
  },
  "handlers": {
    "stdout": {
      "class": "logging.StreamHandler",
      "level": "INFO",
      "formatter": "simple",
      "stream": "ext://sys.stdout"
    },
    "stderr": {
      "class": "logging.StreamHandler",
      "level": "ERROR",
      "formatter": "simple",
      "stream": "ext://sys.stderr"
    },
    "file": {
      "class": "logging.FileHandler",
      "formatter": "simple",
      "filename": "app.log",
      "mode": "w"
    }
  },
  "root": {
    "level": "DEBUG",
    "handlers": [
      "stderr",
      "stdout",
      "file"
    ]
  }
}
```

Cette configuration fait *presque* ce que nous voulons, sauf que `sys.stdout` affiche les messages de gravité ERROR et supérieurs ainsi que les messages INFO et WARNING. Pour éviter cela, nous pouvons configurer un filtre qui exclut ces messages et l'ajouter au gestionnaire approprié. Effectuons la configuration en ajoutant une section `filters` parallèle à `formatters` et `handlers` :

```
{
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    }
}
```

et en changeant la section du gestionnaire `stdout` pour ajouter le filtre :

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

Un filtre n'est qu'une fonction, nous pouvons donc définir un `filter_maker` (une fonction fabrique) comme suit :

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

    def filter(record):
        return record.levelno <= level

    return filter
```

Elle convertit la chaîne transmise en argument vers un niveau numérique puis renvoie une fonction qui ne renvoie `True` que si le niveau de l'enregistrement transmis est inférieur ou égal au niveau spécifié. Notez que dans cet exemple, nous avons défini `filter_maker` dans un script de test `main.py` qui est exécuté depuis la ligne de commande, donc son module est `__main__` – d'où le `__main__.filter_maker` dans la configuration du filtre. Vous devez le changer si vous la définissez dans un module différent.

Avec le filtre, nous pouvons exécuter `main.py` dont voici la version complète :

```
import json
import logging
import logging.config

CONFIG = '''
{
    "version": 1,
    "disable_existing_loggers": false,
    "formatters": {
        "simple": {
            "format": "%(levelname)-8s - %(message)s"
        }
    },
    "filters": {
        "warnings_and_below": {
            "()" : "__main__.filter_maker",
            "level": "WARNING"
        }
    },
    "handlers": {
        "stdout": {
            "class": "logging.StreamHandler",
            "level": "INFO",
```

(suite sur la page suivante)


```

        "formatter": "simple",
        "stream": "ext://sys.stdout",
        "filters": ["warnings_and_below"]
    },
    "stderr": {
        "class": "logging.StreamHandler",
        "level": "ERROR",
        "formatter": "simple",
        "stream": "ext://sys.stderr"
    },
    "file": {
        "class": "logging.FileHandler",
        "formatter": "simple",
        "filename": "app.log",
        "mode": "w"
    }
},
"root": {
    "level": "DEBUG",
    "handlers": [
        "stderr",
        "stdout",
        "file"
    ]
}
}
'''

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

    def filter(record):
        return record.levelno <= level

    return filter

logging.config.dictConfig(json.loads(CONFIG))
logging.debug('A DEBUG message')
logging.info('An INFO message')
logging.warning('A WARNING message')
logging.error('An ERROR message')
logging.critical('A CRITICAL message')

```

Et après l'avoir exécuté comme ceci :

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

Nous obtenons le résultat attendu :

```

$ more *.log
::::::::::::
app.log
::::::::::::
DEBUG      - A DEBUG message
INFO       - An INFO message
WARNING    - A WARNING message
ERROR      - An ERROR message
CRITICAL   - A CRITICAL message
::::::::::::
stderr.log
::::::::::::

```

```
ERROR      - An ERROR message
CRITICAL   - A CRITICAL message
:::
stdout.log
:::
INFO       - An INFO message
WARNING    - A WARNING message
```

6 Exemple d'un serveur de configuration

Voici un exemple de module mettant en œuvre la configuration de la journalisation *via* un serveur :

```
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warning('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
    # cleanup
    logging.config.stopListening()
    t.join()
```

Et voici un script qui, à partir d'un nom de fichier, commence par envoyer la taille du fichier encodée en binaire (comme il se doit), puis envoie ce fichier au serveur pour définir la nouvelle configuration de journalisation :

```
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
```

```
s.close()
print('complete')
```

7 Utilisation de gestionnaires bloquants

Parfois, il est nécessaire que les gestionnaires de journalisation fassent leur travail sans bloquer le fil d'exécution qui émet des événements. C'est généralement le cas dans les applications Web, mais aussi bien sûr dans d'autres scénarios.

Un gestionnaire classiquement lent est le `SMTPHandler` : l'envoi d'e-mails peut prendre beaucoup de temps, pour un certain nombre de raisons indépendantes du développeur (par exemple, une infrastructure de messagerie ou de réseau peu performante). Mais n'importe quel autre gestionnaire utilisant le réseau ou presque peut aussi s'avérer bloquant : même une simple opération `SocketHandler` peut faire une requête DNS implicite et être ainsi très lente (cette requête peut être enfouie profondément dans le code de la bibliothèque d'accès réseau, sous la couche Python, et hors de votre contrôle).

Une solution consiste à utiliser une approche en deux parties. Pour la première partie, affectez un seul `QueueHandler` à la journalisation des fils d'exécution critiques pour les performances. Ils écrivent simplement dans leur file d'attente, qui peut être dimensionnée à une capacité suffisamment grande ou initialisée sans limite supérieure en taille. L'écriture dans la file d'attente est généralement acceptée rapidement, mais nous vous conseillons quand même de prévoir d'intercepter l'exception `queue.Full` par précaution dans votre code. Si vous développez une bibliothèque avec des fils d'exécution critiques pour les performances, documentez-le bien (avec une suggestion de n'affecter que des `QueueHandlers` à votre journalisation) pour faciliter le travail des développeurs qui utilisent votre code.

La deuxième partie de la solution est la classe `QueueListener`, conçue comme l'homologue de `QueueHandler`. Un `QueueListener` est très simple : vous lui passez une file d'attente et des gestionnaires, et il lance un fil d'exécution interne qui scrute la file d'attente pour récupérer les événements envoyés par les `QueueHandlers` (ou toute autre source de `LogRecords`, d'ailleurs). Les `LogRecords` sont supprimés de la file d'attente et transmis aux gestionnaires pour traitement.

L'avantage d'avoir une classe `QueueListener` séparée est que vous pouvez utiliser la même instance pour servir plusieurs `QueueHandlers`. Cela consomme moins de ressources que des instances de gestionnaires réparties chacune dans un fil d'exécution séparé.

Voici un exemple d'utilisation de ces deux classes (les importations sont omises) :

```
que = queue.Queue(-1)  # no limit on size
queue_handler = QueueHandler(que)
handler = logging.StreamHandler()
listener = QueueListener(que, handler)
root = logging.getLogger()
root.addHandler(queue_handler)
formatter = logging.Formatter('%(threadName)s: %(message)s')
handler.setFormatter(formatter)
listener.start()
# The log output will display the thread which generated
# the event (the main thread) rather than the internal
# thread which monitors the internal queue. This is what
# you want to happen.
root.warning('Look out!')
listener.stop()
```

ce qui produit ceci à l'exécution :

```
MainThread: Look out!
```

Note : bien que la discussion précédente n'aborde pas spécifiquement l'utilisation de code asynchrone, mais concerne les gestionnaires de journalisation lents, notez que lors de la journalisation à partir de code asynchrone, les gestion-

naires qui écrivent vers le réseau ou même dans des fichiers peuvent entraîner des problèmes (blocage de la boucle d'événements) car une certaine journalisation est faite à partir du code natif de `asyncio`. Il peut être préférable, si un code asynchrone est utilisé dans une application, d'utiliser l'approche ci-dessus pour la journalisation, de sorte que tout ce qui utilise du code bloquant ne s'exécute que dans le thread `QueueListener`.

Modifié dans la version 3.5 : Avant Python 3.5, la classe `QueueListener` passait chaque message reçu de la file d'attente à chaque gestionnaire avec lequel l'instance avait été initialisée (on supposait que le filtrage de niveau était entièrement effectué de l'autre côté, au niveau de l'alimentation de la file d'attente). Depuis Python 3.5, le comportement peut être modifié en passant l'argument par mot-clé `respect_handler_level=True` au constructeur. Dans ce cas, la `QueueListener` compare le niveau de chaque message avec le niveau défini dans chaque gestionnaire et ne transmet le message que si c'est opportun.

8 Envoi et réception d'événements de journalisation à travers le réseau

Supposons que vous souhaitiez envoyer des événements de journalisation sur un réseau et les traiter à la réception. Une façon simple de faire est d'attacher une instance `SocketHandler` à la journalisation racine de l'émetteur :

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                                              logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

Vous pouvez configurer le récepteur en utilisant le module `socketserver`. Voici un exemple élémentaire :

```
import pickle
import logging
import logging.handlers
import socketserver
import struct

class LogRecordStreamHandler(socketserver.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """
```

(suite sur la page suivante)

```

def handle(self):
    """
    Handle multiple requests - each expected to be a 4-byte length,
    followed by the LogRecord in pickle format. Logs the record
    according to whatever policy is configured locally.
    """
    while True:
        chunk = self.connection.recv(4)
        if len(chunk) < 4:
            break
        slen = struct.unpack('>L', chunk)[0]
        chunk = self.connection.recv(slen)
        while len(chunk) < slen:
            chunk = chunk + self.connection.recv(slen - len(chunk))
        obj = self.unPickle(chunk)
        record = logging.makeLogRecord(obj)
        self.handleLogRecord(record)

def unPickle(self, data):
    return pickle.loads(data)

def handleLogRecord(self, record):
    # if a name is specified, we use the named logger rather than the one
    # implied by the record.
    if self.server.logname is not None:
        name = self.server.logname
    else:
        name = record.name
    logger = logging.getLogger(name)
    # N.B. EVERY record gets logged. This is because Logger.handle
    # is normally called AFTER logger-level filtering. If you want
    # to do filtering, do it at the client end to save wasting
    # cycles and network bandwidth!
    logger.handle(record)

class LogRecordSocketReceiver(socketserver.ThreadingTCPServer):
    """
    Simple TCP socket-based logging receiver suitable for testing.
    """

    allow_reuse_address = True

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
                 handler=LogRecordStreamHandler):
        socketserver.ThreadingTCPServer.__init__(self, (host, port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                      [], [],
                                      self.timeout)

            if rd:
                self.handle_request()
            abort = self.abort

```

```
def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

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

Lancez d'abord le serveur, puis le client. Côté client, rien ne s'affiche sur la console ; côté serveur, vous devez voir quelque chose comme ça :

```
About to start TCP server...
59 root          INFO      Jackdaws love my big sphinx of quartz.
59 myapp.area1    DEBUG     Quick zephyrs blow, vexing daft Jim.
69 myapp.area1    INFO      How quickly daft jumping zebras vex.
69 myapp.area2    WARNING   Jail zesty vixen who grabbed pay from quack.
69 myapp.area2    ERROR     The five boxing wizards jump quickly.
```

Notez que `pickle` introduit des problèmes de sécurité dans certains scénarios. Si vous êtes concerné, vous pouvez utiliser une sérialisation alternative en surchargeant la méthode `makePickle()` par votre propre implémentation, ainsi qu'en adaptant le script ci-dessus pour utiliser votre sérialisation.

8.1 Journalisation en production à l'aide d'un connecteur en écoute sur le réseau

Pour de la journalisation en production *via* un connecteur réseau en écoute, il est probable que vous ayez besoin d'utiliser un outil de surveillance tel que [Supervisor](#). Vous trouverez dans ce [Gist](#) des gabarits pour assurer cette fonction avec *Supervisor*. Il est composé des fichiers suivants :

Fichier	Objectif
<code>prepare.sh</code>	Script Bash pour préparer l'environnement de test
<code>supervisor.conf</code>	Fichier de configuration de <i>Supervisor</i> , avec les entrées pour le connecteur en écoute et l'application web multi-processus
<code>ensure_app.sh</code>	Script Bash pour s'assurer que <i>Supervisor</i> fonctionne bien avec la configuration ci-dessus
<code>log_listener.py</code>	Programme en écoute sur le réseau qui reçoit les événements de journalisation et les enregistre dans un fichier
<code>main.py</code>	Application web simple qui journalise <i>via</i> un connecteur réseau
<code>webapp.json</code>	Fichier JSON de configuration de l'application web
<code>client.py</code>	Script Python qui interagit avec l'application web pour effectuer des appels à la journalisation

L'application Web utilise [Gunicorn](#), qui est un serveur d'applications Web populaire qui démarre plusieurs processus de travail pour gérer les demandes. Cet exemple de configuration montre comment les processus peuvent écrire dans le même fichier journal sans entrer en conflit les uns avec les autres — ils passent tous par le connecteur en écoute.

Pour tester ces fichiers, suivez cette procédure dans un environnement POSIX :

1. Téléchargez l'archive ZIP du [Gist](#) à l'aide du bouton *Download ZIP*.
2. Décompressez les fichiers de l'archive dans un répertoire de travail.
3. Dans le répertoire de travail, exécutez le script de préparation par `bash prepare.sh`. Cela crée un sous-répertoire `run` pour contenir les fichiers journaux et ceux relatifs à *Supervisor*, ainsi qu'un sous-répertoire `venv` pour contenir un environnement virtuel dans lequel `bottle`, `gunicorn` et `supervisor` sont installés.
4. Exécutez `bash ensure_app.sh` pour vous assurer que *Supervisor* s'exécute avec la configuration ci-dessus.
5. Exécutez `venv/bin/python client.py` pour tester l'application Web, ce qui entraîne l'écriture d'enregistrements dans le journal.

6. Inspectez les fichiers journaux dans le sous-répertoire `run`. Vous devriez voir les lignes de journal les plus récentes dans les fichiers de type `app.log*`. Ils ne seront pas dans un ordre particulier, car ils ont été traités par les différents processus de travail de manière non déterministe.
7. Vous pouvez arrêter le connecteur en écoute et l'application Web en exécutant `venv/bin/supervisorctl -c supervisor.conf shutdown`.

Vous devrez peut-être modifier les fichiers de configuration dans le cas peu probable où les ports configurés entrent en conflit avec autre chose dans votre environnement de test.

9 Ajout d'informations contextuelles dans la journalisation

Dans certains cas, vous pouvez souhaiter que la journalisation contienne des informations contextuelles en plus des paramètres transmis à l'appel de journalisation. Par exemple, dans une application réseau, il peut être souhaitable de consigner des informations spécifiques au client dans le journal (par exemple, le nom d'utilisateur ou l'adresse IP du client distant). Bien que vous puissiez utiliser le paramètre *extra* pour y parvenir, il n'est pas toujours pratique de transmettre les informations de cette manière. Il peut être aussi tentant de créer des instances `Logger` connexion par connexion, mais ce n'est pas une bonne idée car ces instances `Logger` ne sont pas éliminées par le ramasse-miettes. Même si ce point n'est pas problématique en soi si la journalisation est configurée avec plusieurs niveaux de granularité, cela peut devenir difficile de gérer un nombre potentiellement illimité d'instances de `Logger`.

9.1 Utilisation d'adaptateurs de journalisation pour transmettre des informations contextuelles

Un moyen simple de transmettre des informations contextuelles accompagnant les informations de journalisation consiste à utiliser la classe `LoggerAdapter`. Cette classe est conçue pour ressembler à un `Logger`, de sorte que vous pouvez appeler `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()` et `log()`. Ces méthodes ont les mêmes signatures que leurs homologues dans `Logger`, vous pouvez donc utiliser les deux types d'instances de manière interchangeable.

Lorsque vous créez une instance de `LoggerAdapter`, vous lui transmettez une instance de `Logger` et un objet dictionnaire qui contient vos informations contextuelles. Lorsque vous appelez l'une des méthodes de journalisation sur une instance de `LoggerAdapter`, elle délègue l'appel à l'instance sous-jacente de `Logger` transmise à son constructeur et s'arrange pour intégrer les informations contextuelles dans l'appel délégué. Voici un extrait du code de `LoggerAdapter` :

```
def debug(self, msg, /, *args, **kwargs):
    """
    Delegate a debug call to the underlying logger, after adding
    contextual information from this adapter instance.
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

Les informations contextuelles sont ajoutées dans la méthode `process()` de `LoggerAdapter`. On lui passe le message et les arguments par mot-clé de l'appel de journalisation, et elle en renvoie des versions (potentiellement) modifiées à utiliser pour la journalisation sous-jacente. L'implémentation par défaut de cette méthode laisse le message seul, mais insère une clé *extra* dans l'argument par mot-clé dont la valeur est l'objet dictionnaire passé au constructeur. Bien sûr, si vous avez passé un argument par mot-clé *extra* dans l'appel à l'adaptateur, il est écrasé silencieusement.

L'avantage d'utiliser *extra* est que les valeurs de l'objet dictionnaire sont fusionnées dans le `__dict__` de l'instance `LogRecord`, ce qui vous permet d'utiliser des chaînes personnalisées avec vos instances `Formatter` qui connaissent les clés de l'objet dictionnaire. Si vous avez besoin d'une méthode différente, par exemple si vous souhaitez ajouter des informations contextuelles avant ou après la chaîne de message, il vous suffit de surcharger `LoggerAdapter` et de remplacer `process()` pour faire ce dont vous avez besoin. Voici un exemple simple :

```
class CustomAdapter(logging.LoggerAdapter):
    """
    This example adapter expects the passed in dict-like object to have a
    'connid' key, whose value in brackets is prepended to the log message.
    """
    def process(self, msg, kwargs):
        return ' [%s] %s' % (self.extra['connid'], msg), kwargs
```

que vous pouvez utiliser comme ceci :

```
logger = logging.getLogger(__name__)
adapter = CustomAdapter(logger, {'connid': some_conn_id})
```

Ainsi, tout événement journalisé aura la valeur de `some_conn_id` insérée en début de message de journalisation.

Utilisation d'objets autres que les dictionnaires pour passer des informations contextuelles

Il n'est pas obligatoire de passer un dictionnaire réel à un `LoggerAdapter`, vous pouvez passer une instance d'une classe qui implémente `__getitem__` et `__iter__` pour qu'il ressemble à un dictionnaire du point de vue de la journalisation. C'est utile si vous souhaitez générer des valeurs de manière dynamique (alors que les valeurs d'un dictionnaire seraient constantes).

9.2 Utilisation de filtres pour transmettre des informations contextuelles

Un `Filter` défini par l'utilisateur peut aussi ajouter des informations contextuelles à la journalisation. Les instances de `Filter` sont autorisées à modifier les `LogRecords` qui leur sont transmis, y compris par l'ajout d'attributs supplémentaires qui peuvent ensuite être intégrés à la journalisation en utilisant une chaîne de formatage appropriée ou, si nécessaire, un `Formatter` personnalisé.

Par exemple, dans une application Web, la requête en cours de traitement (ou du moins ce qu'elle contient d'intéressant) peut être stockée dans une variable locale au fil d'exécution (`threading.local`), puis utilisée dans un `Filter` pour ajouter, par exemple, des informations relatives à la requête (par exemple, l'adresse IP distante et le nom de l'utilisateur) au `LogRecord`, en utilisant les noms d'attribut `ip` et `user` comme dans l'exemple `LoggerAdapter` ci-dessus. Dans ce cas, la même chaîne de formatage peut être utilisée pour obtenir une sortie similaire à celle indiquée ci-dessus. Voici un exemple de script :

```
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into the log.

    Rather than use actual contextual information, we just use random
    data in this demo.
    """
    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

    def filter(self, record):
        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
        return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.
    ↪CRITICAL)
```

(suite sur la page suivante)


```

logging.basicConfig(level=logging.DEBUG,
                    format='% (asctime)-15s %(name)-5s %(levelname)-8s IP:
↳ %(ip)-15s User: %(user)-8s %(message)s')
a1 = logging.getLogger('a.b.c')
a2 = logging.getLogger('d.e.f')

f = ContextFilter()
a1.addFilter(f)
a2.addFilter(f)
a1.debug('A debug message')
a1.info('An info message with %s', 'some parameters')
for x in range(10):
    lvl = choice(levels)
    lvlname = logging.getLevelName(lvl)
    a2.log(lvl, 'A message at %s level with %d %s', lvlname, 2, 'parameters')

```

qui, à l'exécution, produit quelque chose comme ça :

```

2010-09-06 22:38:15,292 a.b.c DEBUG      IP: 123.231.231.123 User: fred      A debug_
↳ message
2010-09-06 22:38:15,300 a.b.c INFO       IP: 192.168.0.1      User: sheila    An info_
↳ message with some parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL  IP: 127.0.0.1      User: sheila    A_
↳ message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR     IP: 127.0.0.1      User: jim       A_
↳ message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG     IP: 127.0.0.1      User: sheila    A_
↳ message at DEBUG level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR     IP: 123.231.231.123 User: fred      A_
↳ message at ERROR level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL  IP: 192.168.0.1      User: jim       A_
↳ message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL  IP: 127.0.0.1      User: sheila    A_
↳ message at CRITICAL level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG     IP: 192.168.0.1      User: jim       A_
↳ message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f ERROR     IP: 127.0.0.1      User: sheila    A_
↳ message at ERROR level with 2 parameters
2010-09-06 22:38:15,301 d.e.f DEBUG     IP: 123.231.231.123 User: fred      A_
↳ message at DEBUG level with 2 parameters
2010-09-06 22:38:15,301 d.e.f INFO      IP: 123.231.231.123 User: fred      A_
↳ message at INFO level with 2 parameters

```

10 Use of contextvars

Since Python 3.7, the `contextvars` module has provided context-local storage which works for both `threading` and `asyncio` processing needs. This type of storage may thus be generally preferable to `thread-locals`. The following example shows how, in a multi-threaded environment, logs can be populated with contextual information such as, for example, request attributes handled by web applications.

For the purposes of illustration, say that you have different web applications, each independent of the other but running in the same Python process and using a library common to them. How can each of these applications have their own log, where all logging messages from the library (and other request processing code) are directed to the appropriate application's log file, while including in the log additional contextual information such as client IP, HTTP request method and client username ?

Let's assume that the library can be simulated by the following code :

```
# webapplib.py
import logging
import time

logger = logging.getLogger(__name__)

def useful():
    # Just a representative event logged from the library
    logger.debug('Hello from webapplib!')
    # Just sleep for a bit so other threads get to run
    time.sleep(0.01)
```

We can simulate the multiple web applications by means of two simple classes, Request and WebApp. These simulate how real threaded web applications work - each request is handled by a thread :

```
# main.py
import argparse
from contextvars import ContextVar
import logging
import os
from random import choice
import threading
import webapplib

logger = logging.getLogger(__name__)
root = logging.getLogger()
root.setLevel(logging.DEBUG)

class Request:
    """
    A simple dummy request class which just holds dummy HTTP request method,
    client IP address and client username
    """
    def __init__(self, method, ip, user):
        self.method = method
        self.ip = ip
        self.user = user

# A dummy set of requests which will be used in the simulation - we'll just pick
# from this list randomly. Note that all GET requests are from 192.168.2.XXX
# addresses, whereas POST requests are from 192.16.3.XXX addresses. Three users
# are represented in the sample requests.

REQUESTS = [
    Request('GET', '192.168.2.20', 'jim'),
    Request('POST', '192.168.3.20', 'fred'),
    Request('GET', '192.168.2.21', 'sheila'),
    Request('POST', '192.168.3.21', 'jim'),
    Request('GET', '192.168.2.22', 'fred'),
    Request('POST', '192.168.3.22', 'sheila'),
]

# Note that the format string includes references to request context information
# such as HTTP method, client IP and username

formatter = logging.Formatter('%(threadName)-11s %(appName)s %(name)-9s %(user)-6s
↳ %(ip)s %(method)-4s %(message)s')

# Create our context variables. These will be filled at the start of request
# processing, and used in the logging that happens during that processing

ctx_request = ContextVar('request')
```

(suite sur la page suivante)

```

ctx_appname = ContextVar('appname')

class InjectingFilter(logging.Filter):
    """
    A filter which injects context-specific information into logs and ensures
    that only information for a specific webapp is included in its log
    """
    def __init__(self, app):
        self.app = app

    def filter(self, record):
        request = ctx_request.get()
        record.method = request.method
        record.ip = request.ip
        record.user = request.user
        record.appName = appName = ctx_appname.get()
        return appName == self.app.name

class WebApp:
    """
    A dummy web application class which has its own handler and filter for a
    webapp-specific log.
    """
    def __init__(self, name):
        self.name = name
        handler = logging.FileHandler(name + '.log', 'w')
        f = InjectingFilter(self)
        handler.setFormatter(formatter)
        handler.addFilter(f)
        root.addHandler(handler)
        self.num_requests = 0

    def process_request(self, request):
        """
        This is the dummy method for processing a request. It's called on a
        different thread for every request. We store the context information into
        the context vars before doing anything else.
        """
        ctx_request.set(request)
        ctx_appname.set(self.name)
        self.num_requests += 1
        logger.debug('Request processing started')
        webapplib.useful()
        logger.debug('Request processing finished')

def main():
    fn = os.path.splitext(os.path.basename(__file__))[0]
    adhf = argparse.ArgumentDefaultsHelpFormatter
    ap = argparse.ArgumentParser(formatter_class=adhf, prog=fn,
                                description='Simulate a couple of web '
                                             'applications handling some '
                                             'requests, showing how request '
                                             'context can be used to '
                                             'populate logs')

    aa = ap.add_argument
    aa('--count', '-c', type=int, default=100, help='How many requests to simulate
    →')
    options = ap.parse_args()

    # Create the dummy webapps and put them in a list which we can use to select
    # from randomly

```

```

app1 = WebApp('app1')
app2 = WebApp('app2')
apps = [app1, app2]
threads = []
# Add a common handler which will capture all events
handler = logging.FileHandler('app.log', 'w')
handler.setFormatter(formatter)
root.addHandler(handler)

# Generate calls to process requests
for i in range(options.count):
    try:
        # Pick an app at random and a request for it to process
        app = choice(apps)
        request = choice(REQUESTS)
        # Process the request in its own thread
        t = threading.Thread(target=app.process_request, args=(request,))
        threads.append(t)
        t.start()
    except KeyboardInterrupt:
        break

# Wait for the threads to terminate
for t in threads:
    t.join()

for app in apps:
    print('%s processed %s requests' % (app.name, app.num_requests))

if __name__ == '__main__':
    main()

```

If you run the above, you should find that roughly half the requests go into `app1.log` and the rest into `app2.log`, and the all the requests are logged to `app.log`. Each webapp-specific log will contain only log entries for only that webapp, and the request information will be displayed consistently in the log (i.e. the information in each dummy request will always appear together in a log line). This is illustrated by the following shell output :

```

~/logging-contextual-webapp$ python main.py
app1 processed 51 requests
app2 processed 49 requests
~/logging-contextual-webapp$ wc -l *.log
 153 app1.log
 147 app2.log
 300 app.log
 600 total
~/logging-contextual-webapp$ head -3 app1.log
Thread-3 (process_request) app1 __main__ jim 192.168.3.21 POST Request
↪processing started
Thread-3 (process_request) app1 webapplib jim 192.168.3.21 POST Hello from
↪webapplib!
Thread-5 (process_request) app1 __main__ jim 192.168.3.21 POST Request
↪processing started
~/logging-contextual-webapp$ head -3 app2.log
Thread-1 (process_request) app2 __main__ sheila 192.168.2.21 GET Request
↪processing started
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET Hello from
↪webapplib!
Thread-2 (process_request) app2 __main__ jim 192.168.2.20 GET Request
↪processing started
~/logging-contextual-webapp$ head app.log
Thread-1 (process_request) app2 __main__ sheila 192.168.2.21 GET Request
↪processing started

```

(suite sur la page suivante)

(suite de la page précédente)

```
Thread-1 (process_request) app2 webapplib sheila 192.168.2.21 GET Hello from_
↳webapplib!
Thread-2 (process_request) app2 __main__ jim 192.168.2.20 GET Request_
↳processing started
Thread-3 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
Thread-2 (process_request) app2 webapplib jim 192.168.2.20 GET Hello from_
↳webapplib!
Thread-3 (process_request) app1 webapplib jim 192.168.3.21 POST Hello from_
↳webapplib!
Thread-4 (process_request) app2 __main__ fred 192.168.2.22 GET Request_
↳processing started
Thread-5 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
Thread-4 (process_request) app2 webapplib fred 192.168.2.22 GET Hello from_
↳webapplib!
Thread-6 (process_request) app1 __main__ jim 192.168.3.21 POST Request_
↳processing started
~/logging-contextual-webapp$ grep app1 app1.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app2.log | wc -l
147
~/logging-contextual-webapp$ grep app1 app.log | wc -l
153
~/logging-contextual-webapp$ grep app2 app.log | wc -l
147
```

11 Imparting contextual information in handlers

Chaque Handler possède sa propre chaîne de filtres. Si vous souhaitez ajouter des informations contextuelles à un LogRecord sans impacter d'autres gestionnaires, vous pouvez utiliser un filtre qui renvoie un nouveau LogRecord au lieu de le modifier sur place, comme dans le script suivant :

```
import copy
import logging

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

if __name__ == '__main__':
    logger = logging.getLogger()
    logger.setLevel(logging.INFO)
    handler = logging.StreamHandler()
    formatter = logging.Formatter('%(message)s from %(user)-8s')
    handler.setFormatter(formatter)
    handler.addFilter(filter)
    logger.addHandler(handler)

    logger.info('A log message')
```

12 Journalisation vers un fichier unique à partir de plusieurs processus

La journalisation est fiable avec les programmes à fils d'exécution multiples (thread-safe) : rien n'empêche plusieurs fils d'exécution de journaliser dans le même fichier, du moment que ces fils d'exécution font partie du même processus. En revanche, il n'existe aucun moyen standard de sérialiser l'accès à un seul fichier sur plusieurs processus en Python. Si vous avez besoin de vous connecter à un seul fichier à partir de plusieurs processus, une façon de le faire est de faire en sorte que tous les processus se connectent à un `SocketHandler`, et d'avoir un processus séparé qui implémente un serveur qui lit à partir de ce connecteur et écrit les journaux dans le fichier (si vous préférez, vous pouvez dédier un fil d'exécution dans l'un des processus existants pour exécuter cette tâche). [Cette section](#) documente cette approche plus en détail et inclut un connecteur en écoute réseau fonctionnel qui peut être utilisé comme point de départ pour l'adapter à vos propres applications.

Vous pouvez également écrire votre propre gestionnaire en utilisant la classe `Lock` du module `multiprocessing` pour sérialiser l'accès au fichier depuis vos processus. Les actuels `FileHandler` et sous-classes n'utilisent pas `multiprocessing` pour le moment, même s'ils pourraient le faire à l'avenir. Notez qu'à l'heure actuelle, le module `multiprocessing` ne fournit pas un verrouillage fonctionnel pour toutes les plates-formes (voir <https://bugs.python.org/issue3770>).

Autrement, vous pouvez utiliser une `Queue` et un `QueueHandler` pour envoyer tous les événements de journalisation à l'un des processus de votre application multi-processus. L'exemple de script suivant montre comment procéder ; dans l'exemple, un processus d'écoute distinct écoute les événements envoyés par les autres processus et les journalise en fonction de sa propre configuration de journalisation. Bien que l'exemple ne montre qu'une seule façon de faire (par exemple, vous pouvez utiliser un fil d'exécution d'écoute plutôt qu'un processus d'écoute séparé – l'implémentation serait analogue), il permet des configurations de journalisation complètement différentes pour celui qui écoute ainsi que pour les autres processus de votre application, et peut être utilisé comme base pour répondre à vos propres exigences :

```
# You'll need these imports in your own code
import logging
import logging.handlers
import multiprocessing

# Next two import lines for this demo only
from random import choice, random
import time

#
# Because you'll want to define the logging configurations for listener and
# ↪workers, the
# listener and worker process functions take a configurer parameter which is a
# ↪callable
# for configuring logging for that process. These functions are also passed the
# ↪queue,
# which they use for communication.
#
# In practice, you can configure the listener however you want, but note that in
# ↪this
# simple example, the listener does not apply level or filter logic to received
# ↪records.
# In practice, you would probably want to do this logic in the worker processes,
# ↪to avoid
# sending events which would be filtered out between processes.
#
# The size of the rotated files is made small so you can see the results easily.
def listener_configurer():
    root = logging.getLogger()
    h = logging.handlers.RotatingFileHandler('mptest.log', 'a', 300, 10)
    f = logging.Formatter('%(asctime)s %(processName)-10s %(name)s %(levelname)-8s
    ↪%(message)s')
```

(suite sur la page suivante)

```

h.setFormatter(f)
root.addHandler(h)

# This is the listener process top-level loop: wait for logging events
# (LogRecords) on the queue and handle them, quit when you get a None for a
# LogRecord.
def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            record = queue.get()
            if record is None: # We send this as a sentinel to tell the listener
                ↪to quit.
                break
            logger = logging.getLogger(record.name)
            logger.handle(record) # No level or filter logic applied - just do it!
        except Exception:
            import sys, traceback
            print('Whoops! Problem:', file=sys.stderr)
            traceback.print_exc(file=sys.stderr)

# Arrays used for random selections in this demo
LEVELS = [logging.DEBUG, logging.INFO, logging.WARNING,
           logging.ERROR, logging.CRITICAL]

LOGGERS = ['a.b.c', 'd.e.f']

MESSAGES = [
    'Random message #1',
    'Random message #2',
    'Random message #3',
]

# The worker configuration is done at the start of the worker process run.
# Note that on Windows you can't rely on fork semantics, so each process
# will run the logging configuration code when it starts.
def worker_configurer(queue):
    h = logging.handlers.QueueHandler(queue) # Just the one handler needed
    root = logging.getLogger()
    root.addHandler(h)
    # send all messages, for demo; no other level or filter logic applied.
    root.setLevel(logging.DEBUG)

# This is the worker process top-level loop, which just logs ten events with
# random intervening delays before terminating.
# The print messages are just so you know it's doing something!
def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print('Worker started: %s' % name)
    for i in range(10):
        time.sleep(random())
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print('Worker finished: %s' % name)

# Here's where the demo gets orchestrated. Create the queue, create and start
# the listener, create ten workers and start them, wait for them to finish,

```

```

# then send a None to the queue to tell the listener to finish.
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(target=listener_process,
                                      args=(queue, listener_configurer))

    listener.start()
    workers = []
    for i in range(10):
        worker = multiprocessing.Process(target=worker_process,
                                      args=(queue, worker_configurer))

        workers.append(worker)
        worker.start()
    for w in workers:
        w.join()
    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()

```

Une variante du script ci-dessus conserve la journalisation dans le processus principal, dans un fil séparé :

```

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

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

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

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

```



```

    }
},
'handlers': {
    'console': {
        'class': 'logging.StreamHandler',
        'level': 'INFO',
    },
    'file': {
        'class': 'logging.FileHandler',
        'filename': 'mplog.log',
        'mode': 'w',
        'formatter': 'detailed',
    },
    'foofile': {
        'class': 'logging.FileHandler',
        'filename': 'mplog-foo.log',
        'mode': 'w',
        'formatter': 'detailed',
    },
    'errors': {
        'class': 'logging.FileHandler',
        'filename': 'mplog-errors.log',
        'mode': 'w',
        'level': 'ERROR',
        'formatter': 'detailed',
    },
},
'loggers': {
    'foo': {
        'handlers': ['foofile']
    }
},
'root': {
    'level': 'DEBUG',
    'handlers': ['console', 'file', 'errors']
},
}
workers = []
for i in range(5):
    wp = Process(target=worker_process, name='worker %d' % (i + 1), args=(q,))
    workers.append(wp)
    wp.start()
logging.config.dictConfig(d)
lp = threading.Thread(target=logger_thread, args=(q,))
lp.start()
# At this point, the main process could do some useful work of its own
# Once it's done that, it can wait for the workers to terminate...
for wp in workers:
    wp.join()
# And now tell the logging thread to finish up, too
q.put(None)
lp.join()

```

Cette variante montre comment appliquer la configuration pour des enregistreurs particuliers - par exemple l'enregistreur `foo` a un gestionnaire spécial qui stocke tous les événements du sous-système `foo` dans un fichier `mplog-foo.log`. C'est utilisé par le mécanisme de journalisation dans le processus principal (même si les événements de journalisation sont générés dans les processus de travail) pour diriger les messages vers les destinations appropriées.

12.1 Utilisation de `concurrent.futures.ProcessPoolExecutor`

Si vous souhaitez utiliser `concurrent.futures.ProcessPoolExecutor` pour démarrer vos processus de travail, vous devez créer la file d'attente légèrement différemment. À la place de

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

vous devez écrire

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

et vous pouvez alors remplacer la création du processus de travail telle que :

```
workers = []
for i in range(10):
    worker = multiprocessing.Process(target=worker_process,
                                    args=(queue, worker_configurer))
    workers.append(worker)
    worker.start()
for w in workers:
    w.join()
```

par celle-ci (souvenez-vous d'importer au préalable `concurrent.futures`) :

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

12.2 Déploiement d'applications Web avec *Gunicorn* et *uWSGI*

Lors du déploiement d'applications Web qui utilisent *Gunicorn* ou *uWSGI* (ou équivalent), plusieurs processus de travail sont créés pour traiter les requêtes des clients. Dans de tels environnements, évitez de créer des gestionnaires à fichiers directement dans votre application Web. Au lieu de cela, utilisez un `SocketHandler` pour journaliser depuis l'application Web vers gestionnaire réseau à l'écoute dans un processus séparé. Cela peut être configuré à l'aide d'un outil de gestion de processus tel que *Supervisor* (voir *Journalisation en production à l'aide d'un connecteur en écoute sur le réseau* pour plus de détails).

13 Utilisation du roulement de fichiers

Parfois, vous souhaitez laisser un fichier de journalisation grossir jusqu'à une certaine taille, puis ouvrir un nouveau fichier et vous y enregistrer les nouveaux événements. Vous souhaitez peut-être conserver un certain nombre de ces fichiers et, lorsque ce nombre de fichiers aura été créé, faire rouler les fichiers afin que le nombre de fichiers et la taille des fichiers restent tous deux limités. Pour ce cas d'usage, `RotatingFileHandler` est inclus dans le paquet de journalisation :

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
```

(suite sur la page suivante)

```

LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)

```

Vous devez obtenir 6 fichiers séparés, chacun contenant une partie de l'historique de journalisation de l'application :

```

logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5

```

Le fichier de journalisation actuel est toujours `logging_rotatingfile_example.out`, et chaque fois qu'il atteint la taille limite, il est renommé avec le suffixe `.1`. Chacun des fichiers de sauvegarde existants est renommé pour incrémenter le suffixe (`.1` devient `.2`, etc.) et le fichier `.6` est effacé.

De toute évidence, la longueur du journal définie dans cet exemple est beaucoup trop petite. À vous de définir `max-Bytes` à une valeur appropriée.

14 Utilisation d'autres styles de formatage

Lorsque la journalisation a été ajoutée à la bibliothèque standard Python, la seule façon de formater les messages avec un contenu variable était d'utiliser la méthode de formatage avec « % ». Depuis, Python s'est enrichi de deux nouvelles méthodes de formatage : `string.Template` (ajouté dans Python 2.4) et `str.format()` (ajouté dans Python 2.6).

La journalisation (à partir de la version 3.2) offre une meilleure prise en charge de ces deux styles de formatage supplémentaires. La classe `Formatter` a été améliorée pour accepter un paramètre par mot-clé facultatif supplémentaire nommé `style`. La valeur par défaut est `'%'`, les autres valeurs possibles étant `'{'` et `'$'`, qui correspondent aux deux autres styles de formatage. La rétrocompatibilité est maintenue par défaut (comme vous vous en doutez) mais, en spécifiant explicitement un paramètre de style, vous avez la possibilité de spécifier des chaînes de format qui fonctionnent avec `str.format()` ou `string.Template`. Voici un exemple de session interactive en console pour montrer les possibilités :

```

>>> import logging
>>> root = logging.getLogger()
>>> root.setLevel(logging.DEBUG)
>>> handler = logging.StreamHandler()
>>> bf = logging.Formatter('{asctime} {name} {levelname:8s} {message}',
...                         style='{')
>>> handler.setFormatter(bf)
>>> root.addHandler(handler)
>>> logger = logging.getLogger('foo.bar')
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:11:55,341 foo.bar DEBUG      This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:12:11,526 foo.bar CRITICAL This is a CRITICAL message

```

(suite sur la page suivante)

```
>>> df = logging.Formatter('$asctime $name ${levelname} $message',
...                         style='$')
>>> handler.setFormatter(df)
>>> logger.debug('This is a DEBUG message')
2010-10-28 15:13:06,924 foo.bar DEBUG This is a DEBUG message
>>> logger.critical('This is a CRITICAL message')
2010-10-28 15:13:11,494 foo.bar CRITICAL This is a CRITICAL message
>>>
```

Notez que le formatage des messages de journalisation est, au final, complètement indépendant de la façon dont un message de journalisation individuel est construit. Vous pouvez toujours utiliser formatage *via* « % », comme ici :

```
>>> logger.error('This is an%s %s %s', 'other,', 'ERROR,', 'message')
2010-10-28 15:19:29,833 foo.bar ERROR This is another, ERROR, message
>>>
```

Les appels de journalisation (`logger.debug()`, `logger.info()` etc.) ne prennent que des paramètres positionnels pour le message de journalisation lui-même, les paramètres par mots-clés étant utilisés uniquement pour déterminer comment gérer le message réel (par exemple, le paramètre par mot-clé `exc_info` indique que les informations de trace doivent être enregistrées, ou le paramètre par mot-clé `extra` indique des informations contextuelles supplémentaires à ajouter au journal). Vous ne pouvez donc pas inclure dans les appels de journalisation à l'aide de la syntaxe `str.format()` ou `string.Template`, car le paquet de journalisation utilise le formatage *via* « % » en interne pour fusionner la chaîne de format et les arguments de variables. Il n'est pas possible de changer ça tout en préservant la rétrocompatibilité puisque tous les appels de journalisation dans le code pré-existant utilisent des chaînes au format « % ».

Il existe cependant un moyen d'utiliser le formatage *via* « {} » et « \$ » pour vos messages de journalisation. Rappelez-vous que, pour un message, vous pouvez utiliser un objet arbitraire comme chaîne de format de message, et que le package de journalisation appelle `str()` sur cet objet pour fabriquer la chaîne finale. Considérez les deux classes suivantes :

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

L'une ou l'autre peut être utilisée à la place d'une chaîne de format `"%(message)s"` ou `"{message}"` ou `"$message"`, afin de mettre en forme *via* « { } » ou « \$ » la partie « message réel » qui apparaît dans la sortie de journal formatée. Il est un peu lourd d'utiliser les noms de classe chaque fois que vous voulez journaliser quelque chose, mais ça devient acceptable si vous utilisez un alias tel que `__` (double trait de soulignement — à ne pas confondre avec `_`, le trait de soulignement unique utilisé comme alias pour `gettext.gettext()` ou ses homologues).

Les classes ci-dessus ne sont pas incluses dans Python, bien qu'elles soient assez faciles à copier et coller dans votre propre code. Elles peuvent être utilisées comme suit (en supposant qu'elles soient déclarées dans un module appelé `wherever`) :

```

>>> from wherever import BraceMessage as __
>>> print(__('Message with {0} {name}', 2, name='placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})',
...         point=p))
Message with coordinates: (0.50, 0.50)
>>> from wherever import DollarMessage as __
>>> print(__('Message with $num $what', num=2, what='placeholders'))
Message with 2 placeholders
>>>

```

Alors que les exemples ci-dessus utilisent `print()` pour montrer comment fonctionne le formatage, utilisez bien sûr `logger.debug()` ou similaire pour journaliser avec cette approche.

Une chose à noter est qu'il n'y a pas de perte de performance significative avec cette approche : le formatage réel ne se produit pas lorsque vous effectuez l'appel de journalisation, mais lorsque (et si) le message journalisé est réellement sur le point d'être écrit dans un journal par un gestionnaire. Ainsi, la seule chose légèrement inhabituelle qui pourrait vous perturber est que les parenthèses entourent la chaîne de format et les arguments, pas seulement la chaîne de format. C'est parce que la notation `__` n'est que du sucre syntaxique pour un appel de constructeur à l'une des classes `XXXMessage`.

Si vous préférez, vous pouvez utiliser un `LoggerAdapter` pour obtenir un effet similaire à ce qui précède, comme dans l'exemple suivant :

```

import logging

class Message:
    def __init__(self, fmt, args):
        self.fmt = fmt
        self.args = args

    def __str__(self):
        return self.fmt.format(*self.args)

class StyleAdapter(logging.LoggerAdapter):
    def __init__(self, logger, extra=None):
        super().__init__(logger, extra or {})

    def log(self, level, msg, /, *args, **kwargs):
        if self.isEnabledFor(level):
            msg, kwargs = self.process(msg, kwargs)
            self.logger._log(level, Message(msg, args), (), **kwargs)

logger = StyleAdapter(logging.getLogger(__name__))

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

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

```

Le script ci-dessus journalise le message `Hello, world!` quand il est lancé avec Python 3.2 ou ultérieur.

15 Personnalisation de LogRecord

Chaque événement de journalisation est représenté par une instance `LogRecord`. Lorsqu'un événement est enregistré et non filtré en raison du niveau d'un enregistreur, un `LogRecord` est créé, rempli avec les informations de l'événement, puis transmis aux gestionnaires de cet enregistreur (et ses ancêtres, jusqu'à et y compris l'enregistreur où la propagation vers le haut de la hiérarchie est désactivée). Avant Python 3.2, il n'y avait que deux endroits où cette création était effectuée :

- `Logger.makeRecord()`, qui est appelée dans le processus normal de journalisation d'un événement. Elle appelait `LogRecord` directement pour créer une instance.
- `makeLogRecord()`, qui est appelée avec un dictionnaire contenant des attributs à ajouter au `LogRecord`. Elle est généralement invoquée lorsqu'un dictionnaire approprié a été reçu par le réseau (par exemple, sous forme de *pickle* via un `SocketHandler`, ou sous format JSON via un `HTTPHandler`).

Cela signifiait généralement que, si vous deviez faire quelque chose de spécial avec un `LogRecord`, vous deviez faire l'une des choses suivantes.

- Créer votre propre sous-classe `Logger`, surchargeant `Logger.makeRecord()`, et la personnaliser à l'aide de `setLoggerClass()` avant que les enregistreurs qui vous intéressaient ne soient instanciés.
- Ajouter un `Filter` à un enregistreur ou un gestionnaire, qui effectuait la manipulation spéciale nécessaire dont vous aviez besoin lorsque sa méthode `filter()` était appelée.

La première approche est un peu lourde dans le scénario où (disons) plusieurs bibliothèques différentes veulent faire des choses différentes. Chacun essaie de définir sa propre sous-classe `Logger`, et celui qui l'a fait en dernier gagne.

La seconde approche fonctionne raisonnablement bien dans de nombreux cas, mais ne vous permet pas, par exemple, d'utiliser une sous-classe spécialisée de `LogRecord`. Les développeurs de bibliothèques peuvent définir un filtre approprié sur leurs enregistreurs, mais ils doivent se rappeler de le faire chaque fois qu'ils introduisent un nouvel enregistreur (ce qu'ils font simplement en ajoutant de nouveaux paquets ou modules et en écrivant :

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

au niveau des modules). C'est probablement trop de choses auxquelles penser. Les développeurs pourraient également ajouter le filtre à un `NullHandler` attaché à leur enregistreur de niveau supérieur, mais cela ne serait pas invoqué si un développeur d'application attachait un gestionnaire à un enregistreur de bibliothèque de niveau inférieur — donc la sortie de ce gestionnaire ne refléterait pas les intentions du développeur de la bibliothèque.

Dans Python 3.2 et ultérieurs, la création de `LogRecord` est effectuée via une fabrique, que vous pouvez spécifier. La fabrique est juste un callable que vous pouvez définir avec `setLogRecordFactory()`, et interroger avec `getLogRecordFactory()`. La fabrique est invoquée avec la même signature que le constructeur `LogRecord`, car `LogRecord` est le paramètre par défaut de la fabrique.

Cette approche permet à une fabrique personnalisée de contrôler tous les aspects de la création d'un `LogRecord`. Par exemple, vous pouvez renvoyer une sous-classe ou simplement ajouter des attributs supplémentaires à l'enregistrement une fois créé, en utilisant un modèle similaire à celui-ci :

```
old_factory = logging.getLogRecordFactory()

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

logging.setLogRecordFactory(record_factory)
```

Ce modèle permet à différentes bibliothèques d'enchaîner des fabriques, et tant qu'elles n'écrasent pas les attributs des autres ou n'écrasent pas involontairement les attributs fournis en standard, il ne devrait pas y avoir de surprise. Cependant, il faut garder à l'esprit que chaque maillon de la chaîne ajoute une surcharge d'exécution à toutes les opérations de journalisation, et la technique ne doit être utilisée que lorsque l'utilisation d'un `Filter` ne permet pas d'obtenir le résultat souhaité.

16 Dérivation de *QueueHandler* – un exemple de *ZeroMQ*

Vous pouvez utiliser une sous-classe *QueueHandler* pour envoyer des messages à d'autres types de files d'attente, par exemple un connecteur *ZeroMQ publish*. Dans l'exemple ci-dessous, le connecteur est créé séparément et transmis au gestionnaire (en tant que file d'attente) :

```
import zmq    # using pyzmq, the Python binding for ZeroMQ
import json   # for serializing records portably

ctx = zmq.Context()
sock = zmq.Socket(ctx, zmq.PUB)    # or zmq.PUSH, or other suitable value
sock.bind('tcp://*:5556')          # or wherever

class ZeroMQSocketHandler(QueueHandler):
    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

handler = ZeroMQSocketHandler(sock)
```

Bien sûr, il existe d'autres manières de faire, par exemple en transmettant les données nécessaires au gestionnaire pour créer le connecteur :

```
class ZeroMQSocketHandler(QueueHandler):
    def __init__(self, uri, socktype=zmq.PUB, ctx=None):
        self.ctx = ctx or zmq.Context()
        socket = zmq.Socket(self.ctx, socktype)
        socket.bind(uri)
        super().__init__(socket)

    def enqueue(self, record):
        self.queue.send_json(record.__dict__)

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

17 Dérivation de *QueueListener* – un exemple de *ZeroMQ*

Vous pouvez également dériver *QueueListener* pour obtenir des messages d'autres types de files d'attente, par exemple un connecteur *ZeroMQ subscribe*. Voici un exemple :

```
class ZeroMQSocketListener(QueueListener):
    def __init__(self, uri, /, *handlers, **kwargs):
        self.ctx = kwargs.get('ctx') or zmq.Context()
        socket = zmq.Socket(self.ctx, zmq.SUB)
        socket.setsockopt_string(zmq.SUBSCRIBE, '')    # subscribe to everything
        socket.connect(uri)
        super().__init__(socket, *handlers, **kwargs)

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

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.

18 Exemple de configuration basée sur un dictionnaire

Vous trouverez ci-dessous un exemple de dictionnaire de configuration de journalisation – il est tiré de la [documentation du projet Django](#). Ce dictionnaire est passé à `dictConfig()` pour activer la configuration :

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d
↪ %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level': 'DEBUG',
            'class': 'django.utils.log.NullHandler',
        },
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers': ['null'],
            'propagate': True,
            'level': 'INFO',
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```


Pour plus d'informations sur cette configuration, vous pouvez consulter la [section correspondante](#) de la documentation de *Django*.

19 Utilisation d'un rotateur et d'un nom pour personnaliser la rotation des journaux

L'extrait de code suivant fournit un exemple de la façon dont vous pouvez définir un nom et un rotateur, avec la compression par *zlib* du journal :

```
import gzip
import logging
import logging.handlers
import os
import shutil

def namer(name):
    return name + ".gz"

def rotator(source, dest):
    with open(source, 'rb') as f_in:
        with gzip.open(dest, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)
    os.remove(source)

rh = logging.handlers.RotatingFileHandler('rotated.log', maxBytes=128,
↳backupCount=5)
rh.rotator = rotator
rh.namer = namer

root = logging.getLogger()
root.setLevel(logging.INFO)
root.addHandler(rh)
f = logging.Formatter('%(asctime)s %(message)s')
rh.setFormatter(f)
for i in range(1000):
    root.info(f'Message no. {i + 1}')
```

Après l'avoir exécuté, vous verrez six nouveaux fichiers, dont cinq sont compressés :

```
$ ls rotated.log*
rotated.log      rotated.log.2.gz  rotated.log.4.gz
rotated.log.1.gz rotated.log.3.gz  rotated.log.5.gz
$ zcat rotated.log.1.gz
2023-01-20 02:28:17,767 Message no. 996
2023-01-20 02:28:17,767 Message no. 997
2023-01-20 02:28:17,767 Message no. 998
```

20 Exemple plus élaboré avec traitement en parallèle

L'exemple suivant que nous allons étudier montre comment la journalisation peut être utilisée, à l'aide de fichiers de configuration, pour un programme effectuant des traitements parallèles. Les configurations sont assez simples, mais servent à illustrer comment des configurations plus complexes pourraient être implémentées dans un scénario multi-processus réel.

Dans l'exemple, le processus principal génère un processus d'écoute et des processus de travail. Chacun des processus, le principal, l'auditeur (*listener_process* dans l'exemple) et les processus de travail (*worker_process* dans l'exemple) ont trois configurations distinctes (les processus de travail partagent tous la même configuration). Nous pouvons voir la journalisation dans le processus principal, comment les processus de travail se connectent à un *QueueHandler* et comment l'auditeur implémente un *QueueListener* avec une configuration de journalisation plus complexe, et s'arrange pour envoyer les événements reçus *via* la file d'attente aux gestionnaires spécifiés dans la configuration. Notez que ces configurations sont purement illustratives, mais vous devriez pouvoir adapter cet exemple à votre propre scénario.

Voici le script – les chaînes de documentation et les commentaires expliquent (en anglais), le principe de fonctionnement :

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

class MyHandler:
    """
    A simple handler for logging events. It runs in the listener process and
    dispatches events to loggers based on the name in the received record,
    which then get dispatched, by the logging system, to the handlers
    configured for those loggers.
    """

    def handle(self, record):
        if record.name == "root":
            logger = logging.getLogger()
        else:
            logger = logging.getLogger(record.name)

        if logger.isEnabledFor(record.levelno):
            # The process name is transformed just to show that it's the listener
            # doing the logging to files and console
            record.processName = '%s (for %s)' % (current_process().name, record.
↪processName)
            logger.handle(record)

def listener_process(q, stop_event, config):
    """
    This could be done in the main process, but is just done in a separate
    process for illustrative purposes.

    This initialises logging according to the specified configuration,
    starts the listener and waits for the main process to signal completion
    via the event. The listener is then stopped, and the process exits.
    """
    logging.config.dictConfig(config)
    listener = logging.handlers.QueueListener(q, MyHandler())
    listener.start()
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured in the
```

(suite sur la page suivante)

```

        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    stop_event.wait()
    listener.stop()

def worker_process(config):
    """
    A number of these are spawned for the purpose of illustration. In
    practice, they could be a heterogeneous bunch of processes rather than
    ones which are identical to each other.

    This initialises logging according to the specified configuration,
    and logs a hundred messages with random levels to randomly selected
    loggers.

    A small sleep is added to allow other processes a chance to run. This
    is not strictly needed, but it mixes the output from the different
    processes a bit more than if it's left out.
    """
    logging.config.dictConfig(config)
    levels = [logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
              logging.CRITICAL]
    loggers = ['foo', 'foo.bar', 'foo.bar.baz',
               'spam', 'spam.ham', 'spam.ham.eggs']
    if os.name == 'posix':
        # On POSIX, the setup logger will have been configured in the
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
        # exist in the child, so it would be created and the message
        # would appear - hence the "if posix" clause.
        logger = logging.getLogger('setup')
        logger.critical('Should not appear, because of disabled logger ...')
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)
        time.sleep(0.01)

def main():
    q = Queue()
    # The main process gets a simple configuration which prints to the console.
    config_initial = {
        'version': 1,
        'handlers': {
            'console': {
                'class': 'logging.StreamHandler',
                'level': 'INFO'
            }
        },
        'root': {
            'handlers': ['console'],
            'level': 'DEBUG'
        }
    }
    # The worker process configuration is just a QueueHandler attached to the

```

```

# root logger, which allows all messages to be sent to the queue.
# We disable existing loggers to disable the "setup" logger used in the
# parent process. This is needed on POSIX because the logger will
# be there in the child following a fork().
config_worker = {
    'version': 1,
    'disable_existing_loggers': True,
    'handlers': {
        'queue': {
            'class': 'logging.handlers.QueueHandler',
            'queue': q
        }
    },
    'root': {
        'handlers': ['queue'],
        'level': 'DEBUG'
    }
}

# The listener process configuration shows that the full flexibility of
# logging configuration is available to dispatch events to handlers however
# you want.
# We disable existing loggers to disable the "setup" logger used in the
# parent process. This is needed on POSIX because the logger will
# be there in the child following a fork().
config_listener = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'detailed': {
            'class': 'logging.Formatter',
            'format': '%(asctime)s %(name)-15s %(levelname)-8s %(processName)-
↪10s %(message)s'
        },
        'simple': {
            'class': 'logging.Formatter',
            'format': '%(name)-15s %(levelname)-8s %(processName)-10s
↪%(message)s'
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'formatter': 'simple',
            'level': 'INFO'
        },
        'file': {
            'class': 'logging.FileHandler',
            'filename': 'mplog.log',
            'mode': 'w',
            'formatter': 'detailed'
        },
        'foofile': {
            'class': 'logging.FileHandler',
            'filename': 'mplog-foo.log',
            'mode': 'w',
            'formatter': 'detailed'
        },
        'errors': {
            'class': 'logging.FileHandler',
            'filename': 'mplog-errors.log',
            'mode': 'w',

```

```

        'formatter': 'detailed',
        'level': 'ERROR'
    },
    'loggers': {
        'foo': {
            'handlers': ['foofile']
        }
    },
    'root': {
        'handlers': ['console', 'file', 'errors'],
        'level': 'DEBUG'
    }
}
# Log some initial events, just to show that logging in the parent works
# normally.
logging.config.dictConfig(config_initial)
logger = logging.getLogger('setup')
logger.info('About to create workers ...')
workers = []
for i in range(5):
    wp = Process(target=worker_process, name='worker %d' % (i + 1),
                 args=(config_worker,))
    workers.append(wp)
    wp.start()
    logger.info('Started worker: %s', wp.name)
logger.info('About to create listener ...')
stop_event = Event()
lp = Process(target=listener_process, name='listener',
             args=(q, stop_event, config_listener))
lp.start()
logger.info('Started listener')
# We now hang around for the workers to finish their work.
for wp in workers:
    wp.join()
# Workers all done, listening can now stop.
# Logging in the parent still works normally.
logger.info('Telling listener to stop ...')
stop_event.set()
lp.join()
logger.info('All done.')

if __name__ == '__main__':
    main()

```

21 Insertion d'une **BOM** dans les messages envoyés à un **SysLogHandler**

La **RFC 5424** requiert qu'un message Unicode soit envoyé à un démon *syslog* sous la forme d'un ensemble d'octets ayant la structure suivante : un composant ASCII pur facultatif, suivi d'une marque d'ordre d'octet (**BOM** pour *Byte Order Mark* en anglais) UTF-8, suivie de contenu Unicode UTF-8 (voir la **la spécification correspondante**).

Dans Python 3.1, du code a été ajouté à **SysLogHandler** pour insérer une **BOM** dans le message mais, malheureusement, il a été implémenté de manière incorrecte, la **BOM** apparaissant au début du message et n'autorisant donc aucun composant ASCII pur à être placé devant.

Comme ce comportement est inadéquat, le code incorrect d'insertion de la **BOM** a été supprimé de Python 3.2.4 et ultérieurs. Cependant, il n'est pas remplacé et, si vous voulez produire des messages conformes **RFC 5424** qui

incluent une *BOM*, une séquence facultative en ASCII pur avant et un Unicode arbitraire après, encodé en UTF-8, alors vous devez appliquer ce qui suit :

1. Adjoignez une instance `Formatter` à votre instance `SysLogHandler`, avec une chaîne de format telle que :

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

Le point de code Unicode U+FEFF, lorsqu'il est encodé en UTF-8, est encodé comme une *BOM* UTF-8 – la chaîne d'octets `b'\xef\xbb\xbf'`.

2. Remplacez la section ASCII par les caractères de votre choix, mais assurez-vous que les données qui y apparaissent après la substitution sont toujours ASCII (ainsi elles resteront inchangées après l'encodage UTF-8).
3. Remplacez la section Unicode par le contenu de votre choix ; si les données qui y apparaissent après la substitution contiennent des caractères en dehors de la plage ASCII, c'est pris en charge – elles seront encodées en UTF-8.

Le message formaté *sera* encodé en UTF-8 par `SysLogHandler`. Si vous suivez les règles ci-dessus, vous devriez pouvoir produire des messages conformes à la [RFC 5424](#). Si vous ne le faites pas, la journalisation ne se plaindra peut-être pas, mais vos messages ne seront pas conformes à la RFC 5424 et votre démon *syslog* est susceptible de se plaindre.

22 Journalisation structurée

Bien que la plupart des messages de journalisation soient destinés à être lus par des humains, et donc difficilement analysables par la machine, il peut arriver que vous souhaitiez produire des messages dans un format structuré dans le but d'être analysé par un programme (sans avoir besoin d'expressions régulières complexes pour analyser le message du journal). C'est simple à réaliser en utilisant le paquet de journalisation. Il existe plusieurs façons d'y parvenir et voici une approche simple qui utilise JSON pour sérialiser l'événement de manière à être analysable par une machine :

```
import json
import logging

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message, json.dumps(self.kwargs))

_ = StructuredMessage # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123, fnum=123.456))
```

Si vous lancez le script ci-dessus, il imprime :

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz", "foo": "bar"}
```

Notez que l'ordre des éléments peut être différent en fonction de la version de Python utilisée.

Si vous avez besoin d'un traitement plus spécifique, vous pouvez utiliser un encodeur JSON personnalisé, comme dans l'exemple complet suivant :

```
import json
import logging

class Encoder(json.JSONEncoder):
    def default(self, o):
```

(suite sur la page suivante)

```

    if isinstance(o, set):
        return tuple(o)
    elif isinstance(o, str):
        return o.encode('unicode_escape').decode('ascii')
    return super().default(o)

class StructuredMessage:
    def __init__(self, message, /, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage  # optional, to improve readability

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value={1, 2, 3}, snowman='\u2603'))

if __name__ == '__main__':
    main()

```

Quand vous exécutez le script ci-dessus, il imprime :

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```

Notez que l'ordre des éléments peut être différent en fonction de la version de Python utilisée.

23 Personnalisation des gestionnaires avec dictConfig()

Il arrive de souhaiter personnaliser les gestionnaires de journalisation d'une manière particulière et, en utilisant `dictConfig()`, vous pourrez peut-être le faire sans avoir à dériver les classes mères. Par exemple, supposons que vous souhaitiez définir le propriétaire d'un fichier journal. Dans un environnement POSIX, cela se fait facilement en utilisant `shutil.chown()`, mais les gestionnaires de fichiers de la bibliothèque standard n'offrent pas cette gestion nativement. Vous pouvez personnaliser la création du gestionnaire à l'aide d'une fonction simple telle que :

```

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)

```

Vous pouvez ensuite spécifier, dans une configuration de journalisation transmise à `dictConfig()`, qu'un gestionnaire de journalisation soit créé en appelant cette fonction :

```

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {

```

(suite sur la page suivante)

```

    'file':{
        # The values below are popped from this dictionary and
        # used to create the handler, set the handler's level and
        # its formatter.
        '(): owned_file_handler,
        'level': 'DEBUG',
        'formatter': 'default',
        # The values below are passed to the handler creator callable
        # as keyword arguments.
        'owner': ['pulse', 'pulse'],
        'filename': 'chowntest.log',
        'mode': 'w',
        'encoding': 'utf-8',
    },
},
'root': {
    'handlers': ['file'],
    'level': 'DEBUG',
},
}

```

Dans cet exemple, nous définissons le propriétaire à l'utilisateur et au groupe pulse, uniquement à des fins d'illustration. Rassemblons le tout dans un script `chowntest.py` :

```

import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
            shutil.chown(filename, *owner)
        return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '(): owned_file_handler,
            'level': 'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

```



```

}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')

```

Pour l'exécuter, vous devrez probablement le faire en tant que root :

```

$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log

```

Notez que cet exemple utilise Python 3.3 car c'est là que `shutil.chown()` fait son apparition. Cette approche devrait fonctionner avec n'importe quelle version de Python prenant en charge `dictConfig()` – à savoir, Python 2.7, 3.2 ou version ultérieure. Avec les versions antérieures à la 3.3, vous devrez implémenter le changement de propriétaire réel en utilisant par exemple `os.chown()`.

En pratique, la fonction de création de gestionnaire peut être située dans un module utilitaire ailleurs dans votre projet. Au lieu de cette ligne dans la configuration :

```
'()': owned_file_handler,
```

vous pouvez écrire par exemple :

```
'()': 'ext://project.util.owned_file_handler',
```

où `project.util` peut être remplacé par le nom réel du paquet où réside la fonction. Dans le script étudié ci-dessus, l'utilisation de `'ext://__main__.owned_file_handler'` devrait fonctionner. Dans le cas présent, l'appelable réel est résolu par `dictConfig()` à partir de la spécification `ext://`.

Nous espérons qu'à partir de cet exemple vous saurez implémenter d'autres types de modification de fichier – par ex. définir des bits d'autorisation POSIX spécifiques – de la même manière, en utilisant `os.chmod()`.

Bien sûr, l'approche pourrait également être étendue à des types de gestionnaires autres qu'un `FileHandler` – par exemple, l'un des gestionnaires à roulement de fichiers ou un autre type de gestionnaire complètement différent.

24 Using particular formatting styles throughout your application

In Python 3.2, the `Formatter` gained a `style` keyword parameter which, while defaulting to `%` for backward compatibility, allowed the specification of `{` or `$` to support the formatting approaches supported by `str.format()` and `string.Template`. Note that this governs the formatting of logging messages for final output to logs, and is completely orthogonal to how an individual logging message is constructed.

Logging calls (`debug()`, `info()` etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the `exc_info` keyword parameter to indicate that traceback information should be logged, or the `extra` keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using `str.format()` or `string.Template` syntax, because internally the logging package uses `%`-formatting to merge the format string and the variable arguments. There would be no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using `%`-format strings.

There have been suggestions to associate format styles with specific loggers, but that approach also runs into backward compatibility problems because any existing code could be using a given logger name and using `%`-formatting.

For logging to work interoperably between any third-party libraries and your code, decisions about formatting need to be made at the level of the individual logging call. This opens up a couple of ways in which alternative formatting styles can be accommodated.

24.1 Using LogRecord factories

In Python 3.2, along with the `Formatter` changes mentioned above, the logging package gained the ability to allow users to set their own `LogRecord` subclasses, using the `setLogRecordFactory()` function. You can use this to set your own subclass of `LogRecord`, which does the Right Thing by overriding the `getMessage()` method. The base class implementation of this method is where the `msg % args` formatting happens, and where you can substitute your alternate formatting; however, you should be careful to support all formatting styles and allow %-formatting as the default, to ensure interoperability with other code. Care should also be taken to call `str(self.msg)`, just as the base implementation does.

Refer to the reference documentation on `setLogRecordFactory()` and `LogRecord` for more information.

24.2 Using custom message objects

There is another, perhaps simpler way that you can use {}- and \$-formatting to construct your individual log messages. You may recall (from arbitrary-object-messages) that when logging you can use an arbitrary object as a message format string, and that the logging package will call `str()` on that object to get the actual format string. Consider the following two classes :

```
class BraceMessage:
    def __init__(self, fmt, /, *args, **kwargs):
        self.fmt = fmt
        self.args = args
        self.kwargs = kwargs

    def __str__(self):
        return self.fmt.format(*self.args, **self.kwargs)

class DollarMessage:
    def __init__(self, fmt, /, **kwargs):
        self.fmt = fmt
        self.kwargs = kwargs

    def __str__(self):
        from string import Template
        return Template(self.fmt).substitute(**self.kwargs)
```

Either of these can be used in place of a format string, to allow {}- or \$-formatting to be used to build the actual "message" part which appears in the formatted log output in place of "%(message)s" or "{message}" or "\$message". If you find it a little unwieldy to use the class names whenever you want to log something, you can make it more palatable if you use an alias such as `M` or `_` for the message (or perhaps `__`, if you are using `_` for localization).

Examples of this approach are given below. Firstly, formatting with `str.format()` :

```
>>> _ = BraceMessage
>>> print(_('Message with {0} {1}', 2, 'placeholders'))
Message with 2 placeholders
>>> class Point: pass
...
>>> p = Point()
>>> p.x = 0.5
>>> p.y = 0.5
>>> print(_('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
```

Secondly, formatting with `string.Template` :

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

One thing to note is that you pay no significant performance penalty with this approach : the actual formatting happens not when you make the logging call, but when (and if) the logged message is actually about to be output to a log by a handler. So the only slightly unusual thing which might trip you up is that the parentheses go around the format string and the arguments, not just the format string. That's because the `__` notation is just syntax sugar for a constructor call to one of the `XXXMessage` classes shown above.

25 Configuring filters with `dictConfig()`

You *can* configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it's only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example :

```
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow

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

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

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print :

```
changed: hello
```

which shows that the filter is working as configured.

A couple of extra points to note :

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form `ext://...` as described in `logging-config-dict-externalobj`. For example, you could have used the text `'ext://__main__.MyFilter'` instead of `MyFilter` in the above example.
- As well as for filters, this technique can also be used to configure custom handlers and formatters. See `logging-config-dict-userdef` for more information on how logging supports using user-defined objects in its configuration, and see the other cookbook recipe *Personnalisation des gestionnaires avec dictConfig()* above.

26 Customized exception formatting

There might be times when you want to do customized exception formatting - for argument's sake, let's say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example :

```
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super().formatException(exc_info)
        return repr(result) # or format into one line however you want to

    def format(self, record):
        s = super().format(record)
        if record.exc_text:
            s = s.replace('\n', '|') + '|'
        return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(message)s|',
                                  '%d/%m/%Y %H:%M:%S')
    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
        logging.exception('ZeroDivisionError: %s', e)

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

When run, this produces a file with exactly two lines :

```
28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: integer division or modulo by zero|
↳ 'Traceback (most recent call last):\n  File "logtest7.py", line 30, in main\n    x = 1 / 0\nZeroDivisionError: integer division or modulo by zero'|
```

(suite sur la page suivante)

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The `traceback` module may be helpful for more specialized needs.

27 Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text-to-speech (TTS) functionality available in your system, even if it doesn't have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using `subprocess`. It's assumed here that TTS command line programs won't expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it's acceptable to have the messages spoken one at a time rather than concurrently. The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the `espeak` TTS package is available :

```
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)

        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
    logging.debug('Goodbye')

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

When run, this script should say "Hello" and then "Goodbye" in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.

28 Buffering logging messages and outputting them conditionally

There might be situations where you want to log messages in a temporary area and only output them if a certain condition occurs. For example, you may want to start logging debug events in a function, and if the function completes without errors, you don't want to clutter the log with the collected debug information, but if there is an error, you want all the debug information to be output as well as the error.

Here is an example which shows how you could do this using a decorator for your functions where you want logging to behave this way. It makes use of the `logging.handlers.MemoryHandler`, which allows buffering of logged events until some condition occurs, at which point the buffered events are flushed - passed to another handler (the target handler) for processing. By default, the `MemoryHandler` flushed when its buffer gets filled up or an event whose level is greater than or equal to a specified threshold is seen. You can use this recipe with a more specialised subclass of `MemoryHandler` if you want custom flushing behavior.

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to `sys.stderr` to say what level it's about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at `ERROR` and `CRITICAL` levels - otherwise, it only logs at `DEBUG`, `INFO` and `WARNING` levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that's required. The decorator takes a logger as a parameter and attaches a memory handler for the duration of the call to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer (number of records buffered). These default to a `StreamHandler` which writes to `sys.stderr`, `logging.ERROR` and 100 respectively.

Here's the script :

```
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level, target=target_
↪ handler)

    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
                logger.removeHandler(handler)
        return wrapper

    return decorator

def write_line(s):
    sys.stderr.write('%s\n' % s)
```

(suite sur la page suivante)

```

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail

decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)

```

When this script is run, the following output should be observed :

```

Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
about to log at CRITICAL ...
Actually logged at CRITICAL

```

As you can see, actual logging output only occurs when an event is logged whose severity is ERROR or greater, but in that case, any previous events at lower severities are also logged.

You can of course use the conventional means of decoration :

```
@log_if_errors(logger)
```

```
def foo(fail=False):
    ...
```

29 Sending logging messages to email, with buffering

To illustrate how you can send log messages via email, so that a set number of messages are sent per email, you can subclass `BufferingHandler`. In the following example, which you can adapt to suit your specific needs, a simple test harness is provided which allows you to run the script with command line arguments specifying what you typically need to send things via SMTP. (Run the downloaded script with the `-h` argument to see the required and optional arguments.)

```
import logging
import logging.handlers
import smtplib

class BufferingSMTPHandler(logging.handlers.BufferingHandler):
    def __init__(self, mailhost, port, username, password, fromaddr, toaddrs,
                  subject, capacity):
        logging.handlers.BufferingHandler.__init__(self, capacity)
        self.mailhost = mailhost
        self.mailport = port
        self.username = username
        self.password = password
        self.fromaddr = fromaddr
        if isinstance(toaddrs, str):
            toaddrs = [toaddrs]
        self.toaddrs = toaddrs
        self.subject = subject
        self.setFormatter(logging.Formatter("%(asctime)s %(levelname)-5s
→ %(message)s"))

    def flush(self):
        if len(self.buffer) > 0:
            try:
                smtp = smtplib.SMTP(self.mailhost, self.mailport)
                smtp.starttls()
                smtp.login(self.username, self.password)
                msg = "From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n" % (self.fromaddr,
→ ', '.join(self.toaddrs), self.subject)
                for record in self.buffer:
                    s = self.format(record)
                    msg = msg + s + "\r\n"
                smtp.sendmail(self.fromaddr, self.toaddrs, msg)
                smtp.quit()
            except Exception:
                if logging.raiseExceptions:
                    raise
                self.buffer = []

if __name__ == '__main__':
    import argparse

    ap = argparse.ArgumentParser()
    aa = ap.add_argument
    aa('host', metavar='HOST', help='SMTP server')
    aa('--port', '-p', type=int, default=587, help='SMTP port')
    aa('user', metavar='USER', help='SMTP username')
    aa('password', metavar='PASSWORD', help='SMTP password')
```

(suite sur la page suivante)


```

aa('to', metavar='TO', help='Addressee for emails')
aa('sender', metavar='SENDER', help='Sender email address')
aa('--subject', '-s',
    default='Test Logging email from Python logging module (buffering)',
    help='Subject of email')
options = ap.parse_args()
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
h = BufferingSMTPHandler(options.host, options.port, options.user,
                        options.password, options.sender,
                        options.to, options.subject, 10)

logger.addHandler(h)
for i in range(102):
    logger.info("Info index = %d", i)
h.flush()
h.close()

```

If you run this script and your SMTP server is correctly set up, you should find that it sends eleven emails to the addressee you specify. The first ten emails will each have ten log messages, and the eleventh will have two messages. That makes up 102 messages as specified in the script.

30 Formatting times using UTC (GMT) via configuration

Sometimes you want to format times using UTC, which can be done using a class such as `UTCFormatter`, shown below :

```

import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

```

and you can then use the `UTCFormatter` in your code instead of `Formatter`. If you want to do that via configuration, you can use the `dictConfig()` API with an approach illustrated by the following complete example :

```

import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        }
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',

```

```

    },
    'console2': {
        'class': 'logging.StreamHandler',
        'formatter': 'local',
    },
},
'root': {
    'handlers': ['console1', 'console2'],
}
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())

```

When this script is run, it should print something like :

```

2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29 2015
2015-10-17 13:53:29,501 The local time is Sat Oct 17 13:53:29 2015

```

showing how the time is formatted both as local time and UTC, one for each handler.

31 Using a context manager for selective logging

There are times when it would be useful to temporarily change the logging configuration and revert it back after doing something. For this, a context manager is the most obvious way of saving and restoring the logging context. Here is a simple example of such a context manager, which allows you to optionally change the logging level and add a logging handler purely in the scope of the context manager :

```

import logging
import sys

class LoggingContext:
    def __init__(self, logger, level=None, handler=None, close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions

```

If you specify a level value, the logger's level is set to that value in the scope of the with block covered by the context manager. If you specify a handler, it is added to the logger on entry to the block and removed on exit from the block. You can also ask the manager to close the handler for you on block exit - you could do this if you don't need the handler any more.

To illustrate how it works, we can add the following block of code to the above :

```
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
        logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h, close=True):
        logger.debug('5. This should appear twice - once on stderr and once on_
↪stdout.')
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

We initially set the logger's level to INFO, so message #1 appears and message #2 doesn't. We then change the level to DEBUG temporarily in the following with block, and so message #3 appears. After the block exits, the logger's level is restored to INFO and so message #4 doesn't appear. In the next with block, we set the level to DEBUG again but also add a handler writing to `sys.stdout`. Thus, message #5 appears twice on the console (once via `stderr` and once via `stdout`). After the with statement's completion, the status is as it was before so message #6 appears (like message #1) whereas message #7 doesn't (just like message #2).

If we run the resulting script, the result is as follows :

```
$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

If we run it again, but pipe `stderr` to `/dev/null`, we see the following, which is the only message written to `stdout` :

```
$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on stdout.
```

Once again, but piping `stdout` to `/dev/null`, we get :

```
$ python logctx.py >/dev/null
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

In this case, the message #5 printed to `stdout` doesn't appear, as expected.

Of course, the approach described here can be generalised, for example to attach logging filters temporarily. Note that the above code works in Python 2 as well as Python 3.

32 A CLI application starter template

Here's an example which shows how you can :

- Use a logging level based on command-line arguments
- Dispatch to multiple subcommands in separate files, all logging at the same level in a consistent way
- Make use of simple, minimal configuration

Suppose we have a command-line application whose job is to stop, start or restart some services. This could be organised for the purposes of illustration as a file `app.py` that is the main script for the application, with individual commands implemented in `start.py`, `stop.py` and `restart.py`. Suppose further that we want to control the verbosity of the application via a command-line argument, defaulting to logging.INFO. Here's one way that `app.py` could be written :

```
import argparse
import importlib
import logging
import os
import sys

def main(args=None):
    scriptname = os.path.basename(__file__)
    parser = argparse.ArgumentParser(scriptname)
    levels = ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
    parser.add_argument('--log-level', default='INFO', choices=levels)
    subparsers = parser.add_subparsers(dest='command',
                                      help='Available commands:')
    start_cmd = subparsers.add_parser('start', help='Start a service')
    start_cmd.add_argument('name', metavar='NAME',
                          help='Name of service to start')
    stop_cmd = subparsers.add_parser('stop',
                                    help='Stop one or more services')
    stop_cmd.add_argument('names', metavar='NAME', nargs='+',
                        help='Name of service to stop')
    restart_cmd = subparsers.add_parser('restart',
                                       help='Restart one or more services')
    restart_cmd.add_argument('names', metavar='NAME', nargs='+',
                          help='Name of service to restart')
    options = parser.parse_args()
    # the code to dispatch commands could all be in this file. For the purposes
    # of illustration only, we implement each command in a separate module.
    try:
        mod = importlib.import_module(options.command)
        cmd = getattr(mod, 'command')
    except (ImportError, AttributeError):
        print('Unable to find the code for command \'%s\'' % options.command)
        return 1
    # Could get fancy here and load configuration from file or dictionary
    logging.basicConfig(level=options.log_level,
                      format='%(levelname)s %(name)s %(message)s')
    cmd(options)

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

And the start, stop and restart commands can be implemented in separate modules, like so for starting :

```
# start.py
import logging

logger = logging.getLogger(__name__)

def command(options):
```

(suite sur la page suivante)

```

logger.debug('About to start %s', options.name)
# actually do the command processing here ...
logger.info('Started the \'%s\' service.', options.name)

```

and thus for stopping :

```

# stop.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to stop %s', services)
    # actually do the command processing here ...
    logger.info('Stopped the %s service%s.', services, plural)

```

and similarly for restarting :

```

# restart.py
import logging

logger = logging.getLogger(__name__)

def command(options):
    n = len(options.names)
    if n == 1:
        plural = ''
        services = '\'%s\'' % options.names[0]
    else:
        plural = 's'
        services = ', '.join('\'%s\'' % name for name in options.names)
        i = services.rfind(', ')
        services = services[:i] + ' and ' + services[i + 2:]
    logger.debug('About to restart %s', services)
    # actually do the command processing here ...
    logger.info('Restarted the %s service%s.', services, plural)

```

If we run this application with the default log level, we get output like this :

```

$ python app.py start foo
INFO start Started the 'foo' service.

$ python app.py stop foo bar
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py restart foo bar baz
INFO restart Restarted the 'foo', 'bar' and 'baz' services.

```

The first word is the logging level, and the second word is the module or package name of the place where the event was logged.

If we change the logging level, then we can change the information sent to the log. For example, if we want more information :

```
$ python app.py --log-level DEBUG start foo
DEBUG start About to start foo
INFO start Started the 'foo' service.

$ python app.py --log-level DEBUG stop foo bar
DEBUG stop About to stop 'foo' and 'bar'
INFO stop Stopped the 'foo' and 'bar' services.

$ python app.py --log-level DEBUG restart foo bar baz
DEBUG restart About to restart 'foo', 'bar' and 'baz'
INFO restart Restarted the 'foo', 'bar' and 'baz' services.
```

And if we want less :

```
$ python app.py --log-level WARNING start foo
$ python app.py --log-level WARNING stop foo bar
$ python app.py --log-level WARNING restart foo bar baz
```

In this case, the commands don't print anything to the console, since nothing at WARNING level or above is logged by them.

33 A Qt GUI for logging

A question that comes up from time to time is about how to log to a GUI application. The Qt framework is a popular cross-platform UI framework with Python bindings using PySide2 or PyQt5 libraries.

The following example shows how to log to a Qt GUI. This introduces a simple QtHandler class which takes a callable, which should be a slot in the main thread that does GUI updates. A worker thread is also created to show how you can log to the GUI from both the UI itself (via a button for manual logging) as well as a worker thread doing work in the background (here, just logging messages at random levels with random short delays in between).

The worker thread is implemented using Qt's QThread class rather than the threading module, as there are circumstances where one has to use QThread, which offers better integration with other Qt components.

The code should work with recent releases of either PySide2 or PyQt5. You should be able to adapt the approach to earlier versions of Qt. Please refer to the comments in the code snippet for more detailed information.

```
import datetime
import logging
import random
import sys
import time

# Deal with minor differences between PySide2 and PyQt5
try:
    from PySide2 import QtCore, QtGui, QtWidgets
    Signal = QtCore.Signal
    Slot = QtCore.Slot
except ImportError:
    from PyQt5 import QtCore, QtGui, QtWidgets
    Signal = QtCore.pyqtSignal
    Slot = QtCore.pyqtSlot

logger = logging.getLogger(__name__)

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

(suite sur la page suivante)

```

#
class Signaller(QtCore.QObject):
    signal = Signal(str, logging.LogRecord)

#
# Output to a Qt GUI is only supposed to happen on the main thread. So, this
# handler is designed to take a slot function which is set up to run in the main
# thread. In this example, the function takes a string argument which is a
# formatted log message, and the log record which generated it. The formatted
# string is just a convenience - you could format a string for output any way
# you like in the slot function itself.
#
# You specify the slot function to do whatever GUI updates you want. The handler
# doesn't know or care about specific UI elements.
#
class QtHandler(logging.Handler):
    def __init__(self, slotfunc, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.signaller = Signaller()
        self.signaller.signal.connect(slotfunc)

    def emit(self, record):
        s = self.format(record)
        self.signaller.signal.emit(s, record)

#
# This example uses QThreads, which means that the threads at the Python level
# are named something like "Dummy-1". The function below gets the Qt name of the
# current thread.
#
def ctname():
    return QtCore.QThread.currentThread().objectName()

#
# Used to generate random levels for logging.
#
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR,
          logging.CRITICAL)

#
# This worker class represents work that is done in a thread separate to the
# main thread. The way the thread is kicked off to do work is via a button press
# that connects to a slot in the worker.
#
# Because the default threadName value in the LogRecord isn't much use, we add
# a qThreadName which contains the QThread name as computed above, and pass that
# value in an "extra" dictionary which is used to update the LogRecord with the
# QThread name.
#
# This example worker just outputs messages sequentially, interspersed with
# random delays of the order of a few seconds.
#
class Worker(QtCore.QObject):
    @Slot()
    def start(self):
        extra = {'qThreadName': ctname()}
        logger.debug('Started work', extra=extra)
        i = 1
        # Let the thread run until interrupted. This allows reasonably clean
        # thread termination.

```

```

while not QtCore.QThread.currentThread().isInterrupted():
    delay = 0.5 + random.random() * 2
    time.sleep(delay)
    level = random.choice(LEVELS)
    logger.log(level, 'Message after delay of %3.1f: %d', delay, i,
→extra=extra)
    i += 1

#
# Implement a simple UI for this cookbook example. This contains:
#
# * A read-only text edit window which holds formatted log messages
# * A button to start work and log stuff in a separate thread
# * A button to log something from the main thread
# * A button to clear the log window
#
class Window(QtWidgets.QWidget):

    COLORS = {
        logging.DEBUG: 'black',
        logging.INFO: 'blue',
        logging.WARNING: 'orange',
        logging.ERROR: 'red',
        logging.CRITICAL: 'purple',
    }

    def __init__(self, app):
        super().__init__()
        self.app = app
        self.textedit = te = QtWidgets.QPlainTextEdit(self)
        # Set whatever the default monospace font is for the platform
        f = QtGui.QFont('monospace')
        f.setStyleHint(f.Monospace)
        te.setFont(f)
        te.setReadOnly(True)
        PB = QtWidgets.QPushButton
        self.work_button = PB('Start background work', self)
        self.log_button = PB('Log a message at a random level', self)
        self.clear_button = PB('Clear log window', self)
        self.handler = h = QtHandler(self.update_status)
        # Remember to use qThreadName rather than threadName in the format string.
        fs = '%(asctime)s %(qThreadName)-12s %(levelname)-8s %(message)s'
        formatter = logging.Formatter(fs)
        h.setFormatter(formatter)
        logger.addHandler(h)
        # Set up to terminate the QThread when we exit
        app.aboutToQuit.connect(self.force_quit)

        # Lay out all the widgets
        layout = QtWidgets.QVBoxLayout(self)
        layout.addWidget(te)
        layout.addWidget(self.work_button)
        layout.addWidget(self.log_button)
        layout.addWidget(self.clear_button)
        self.setFixedSize(900, 400)

        # Connect the non-worker slots and signals
        self.log_button.clicked.connect(self.manual_update)
        self.clear_button.clicked.connect(self.clear_display)

        # Start a new worker thread and connect the slots for the worker

```



```

self.start_thread()
self.work_button.clicked.connect(self.worker.start)
# Once started, the button should be disabled
self.work_button.clicked.connect(lambda : self.work_button.
→setEnabled(False))

def start_thread(self):
    self.worker = Worker()
    self.worker_thread = QtCore.QThread()
    self.worker.setObjectName('Worker')
    self.worker_thread.setObjectName('WorkerThread') # for qThreadName
    self.worker.moveToThread(self.worker_thread)
    # This will start an event loop in the worker thread
    self.worker_thread.start()

def kill_thread(self):
    # Just tell the worker to stop, then tell it to quit and wait for that
    # to happen
    self.worker_thread.requestInterruption()
    if self.worker_thread.isRunning():
        self.worker_thread.quit()
        self.worker_thread.wait()
    else:
        print('worker has already exited.')

def force_quit(self):
    # For use when the window is closed
    if self.worker_thread.isRunning():
        self.kill_thread()

# The functions below update the UI and run in the main thread because
# that's where the slots are set up

@Slot(str, logging.LogRecord)
def update_status(self, status, record):
    color = self.COLORS.get(record.levelno, 'black')
    s = '<pre><font color="%s">%s</font></pre>' % (color, status)
    self.textedit.appendHtml(s)

@Slot()
def manual_update(self):
    # This function uses the formatted message passed in, but also uses
    # information from the record to format the message in an appropriate
    # color according to its severity (level).
    level = random.choice(LEVELS)
    extra = {'qThreadName': ctname()}
    logger.log(level, 'Manually logged!', extra=extra)

@Slot()
def clear_display(self):
    self.textedit.clear()

def main():
    QtCore.QThread.currentThread().setObjectName('MainThread')
    logging.getLogger().setLevel(logging.DEBUG)
    app = QtWidgets.QApplication(sys.argv)
    example = Window(app)
    example.show()
    sys.exit(app.exec_())

```

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

34 Logging to syslog with RFC5424 support

Although **RFC 5424** dates from 2009, most syslog servers are configured by default to use the older **RFC 3164**, which hails from 2001. When logging was added to Python in 2003, it supported the earlier (and only existing) protocol at the time. Since RFC5424 came out, as there has not been widespread deployment of it in syslog servers, the SysLogHandler functionality has not been updated.

RFC 5424 contains some useful features such as support for structured data, and if you need to be able to log to a syslog server with support for it, you can do so with a subclassed handler which looks something like this :

```
import datetime
import logging.handlers
import re
import socket
import time

class SysLogHandler5424(logging.handlers.SysLogHandler):

    tz_offset = re.compile(r'([+-]\d{2}) (\d{2})$')
    escaped = re.compile(r'([\\"\\])')

    def __init__(self, *args, **kwargs):
        self.msgid = kwargs.pop('msgid', None)
        self.appname = kwargs.pop('appname', None)
        super().__init__(*args, **kwargs)

    def format(self, record):
        version = 1
        asctime = datetime.datetime.fromtimestamp(record.created).isoformat()
        m = self.tz_offset.match(time.strftime('%z'))
        has_offset = False
        if m and time.timezone:
            hrs, mins = m.groups()
            if int(hrs) or int(mins):
                has_offset = True
        if not has_offset:
            asctime += 'Z'
        else:
            asctime += f'{hrs}:{mins}'
        try:
            hostname = socket.gethostname()
        except Exception:
            hostname = '-'
        appname = self.appname or '-'
        procid = record.process
        msgid = '-'
        msg = super().format(record)
        sdata = '-'
        if hasattr(record, 'structured_data'):
            sd = record.structured_data
            # This should be a dict where the keys are SD-ID and the value is a
            # dict mapping PARAM-NAME to PARAM-VALUE (refer to the RFC for what
            ↪ these
            # mean)
            # There's no error checking here - it's purely for illustration, and
            ↪ you
```

```

# can adapt this code for use in production environments
parts = []

def replacer(m):
    g = m.groups()
    return '\\\\' + g[0]

for sdid, dv in sd.items():
    part = f'[{sdid}]'
    for k, v in dv.items():
        s = str(v)
        s = self.escaped.sub(replacer, s)
        part += f' {k}="{s}"'
    part += ']'
    parts.append(part)
sdata = ''.join(parts)
return f'{version} {asctime} {hostname} {appname} {procid} {msgid} {sdata}
↪{msg}'

```

You'll need to be familiar with RFC 5424 to fully understand the above code, and it may be that you have slightly different needs (e.g. for how you pass structural data to the log). Nevertheless, the above should be adaptable to your specific needs. With the above handler, you'd pass structured data using something like this :

```

sd = {
    'foo@12345': {'bar': 'baz', 'baz': 'bozz', 'fizz': r'buzz'},
    'foo@54321': {'rab': 'baz', 'zab': 'bozz', 'zzif': r'buzz'}
}
extra = {'structured_data': sd}
i = 1
logger.debug('Message %d', i, extra=extra)

```

35 How to treat a logger like an output stream

Sometimes, you need to interface to a third-party API which expects a file-like object to write to, but you want to direct the API's output to a logger. You can do this using a class which wraps a logger with a file-like API. Here's a short script illustrating such a class :

```

import logging

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

    def write(self, message):
        if message != '\n': # avoid printing bare newlines, if you like
            self.logger.log(self.level, message)

    def flush(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation
        pass

    def close(self):
        # doesn't actually do anything, but might be expected of a file-like
        # object - so optional depending on your situation. You might want
        # to set a flag so that later calls to write raise an exception
        pass

```

(suite sur la page suivante)

```
def main():
    logging.basicConfig(level=logging.DEBUG)
    logger = logging.getLogger('demo')
    info_fp = LoggerWriter(logger, logging.INFO)
    debug_fp = LoggerWriter(logger, logging.DEBUG)
    print('An INFO message', file=info_fp)
    print('A DEBUG message', file=debug_fp)

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

When this script is run, it prints

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

You could also use `LoggerWriter` to redirect `sys.stdout` and `sys.stderr` by doing something like this :

```
import sys

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

You should do this *after* configuring logging for your needs. In the above example, the `basicConfig()` call does this (using the `sys.stderr` value *before* it is overwritten by a `LoggerWriter` instance). Then, you'd get this kind of result :

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

Of course, these above examples show output according to the format used by `basicConfig()`, but you can use a different formatter when you configure logging.

Note that with the above scheme, you are somewhat at the mercy of buffering and the sequence of write calls which you are intercepting. For example, with the definition of `LoggerWriter` above, if you have the snippet

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

then running the script results in

```
WARNING:demo:Traceback (most recent call last):

WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 53, in
↳<module>

WARNING:demo:
WARNING:demo:main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/test.py", line 49, in main

WARNING:demo:
WARNING:demo:1 / 0
WARNING:demo:ZeroDivisionError
WARNING:demo::
WARNING:demo:division by zero
```

As you can see, this output isn't ideal. That's because the underlying code which writes to `sys.stderr` makes multiple writes, each of which results in a separate logged line (for example, the last three lines above). To get around

this problem, you need to buffer things and only output log lines when newlines are seen. Let's use a slightly better implementation of `LoggerWriter` :

```
class BufferingLoggerWriter(LoggerWriter):
    def __init__(self, logger, level):
        super().__init__(logger, level)
        self.buffer = ''

    def write(self, message):
        if '\n' not in message:
            self.buffer += message
        else:
            parts = message.split('\n')
            if self.buffer:
                s = self.buffer + parts.pop(0)
                self.logger.log(self.level, s)
            self.buffer = parts.pop()
            for part in parts:
                self.logger.log(self.level, part)
```

This just buffers up stuff until a newline is seen, and then logs complete lines. With this approach, you get better output :

```
WARNING:demo:Traceback (most recent call last):
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 55, in
↳<module>
WARNING:demo:    main()
WARNING:demo:  File "/home/runner/cookbook-loggerwriter/main.py", line 52, in main
WARNING:demo:    1/0
WARNING:demo:ZeroDivisionError: division by zero
```

36 Patterns to avoid

Although the preceding sections have described ways of doing things you might need to do or deal with, it is worth mentioning some usage patterns which are *unhelpful*, and which should therefore be avoided in most cases. The following sections are in no particular order.

36.1 Opening the same log file multiple times

On Windows, you will generally not be able to open the same file multiple times as this will lead to a "file is in use by another process" error. However, on POSIX platforms you'll not get any errors if you open the same file multiple times. This could be done accidentally, for example by :

- Adding a file handler more than once which references the same file (e.g. by a copy/paste/forget-to-change error).
- Opening two files that look different, as they have different names, but are the same because one is a symbolic link to the other.
- Forking a process, following which both parent and child have a reference to the same file. This might be through use of the `multiprocessing` module, for example.

Opening a file multiple times might *appear* to work most of the time, but can lead to a number of problems in practice :

- Logging output can be garbled because multiple threads or processes try to write to the same file. Although logging guards against concurrent use of the same handler instance by multiple threads, there is no such protection if concurrent writes are attempted by two different threads using two different handler instances which happen to point to the same file.
- An attempt to delete a file (e.g. during file rotation) silently fails, because there is another reference pointing to it. This can lead to confusion and wasted debugging time - log entries end up in unexpected places, or are

lost altogether. Or a file that was supposed to be moved remains in place, and grows in size unexpectedly despite size-based rotation being supposedly in place.

Use the techniques outlined in *Journalisation vers un fichier unique à partir de plusieurs processus* to circumvent such issues.

36.2 Using loggers as attributes in a class or passing them as parameters

While there might be unusual cases where you'll need to do this, in general there is no point because loggers are singletons. Code can always access a given logger instance by name using `logging.getLogger(name)`, so passing instances around and holding them as instance attributes is pointless. Note that in other languages such as Java and C#, loggers are often static class attributes. However, this pattern doesn't make sense in Python, where the module (and not the class) is the unit of software decomposition.

36.3 Adding handlers other than `NullHandler` to a logger in a library

Configuring logging by adding handlers, formatters and filters is the responsibility of the application developer, not the library developer. If you are maintaining a library, ensure that you don't add handlers to any of your loggers other than a `NullHandler` instance.

36.4 Creating a lot of loggers

Loggers are singletons that are never freed during a script execution, and so creating lots of loggers will use up memory which can't then be freed. Rather than create a logger per e.g. file processed or network connection made, use the *existing mechanisms* for passing contextual information into your logs and restrict the loggers created to those describing areas within your application (generally modules, but occasionally slightly more fine-grained than that).

37 Autres ressources

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.

Tutoriel de découverte

Tutoriel avancé

Index

R

RFC

RFC 3164, [58](#)

RFC 5424, [37](#), [38](#), [58](#)

RFC 5424#section-6, [37](#)