Libro de recetas de Logging

Versión 3.9.21

Guido van Rossum
and the Python development team

diciembre 08, 2024

Python Software Foundation
Email: docs@python.org

Índice general

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

Esta página contiene un número de recetas sobre *logging*, que han sido útiles en el pasado.

1 Usar logging en múltiples módulos

Múltiples llamadas a `logging.getLogger('someLogger')` devuelven una referencia al mismo objeto logger. Esto es cierto no solo dentro del mismo módulo, sino también en todos los módulos siempre que estén ejecutándose en el mismo proceso del intérprete de Python. Es válido para las referencias al mismo objeto. Además, el código de la aplicación puede definir y configurar un logger primario en un módulo y crear (pero no configurar) un logger secundario en un módulo separado, y todas las llamadas al secundario pasarán al principal. A continuación un módulo 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()')
```

Y aquí un módulo auxiliar:

```
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"')
```

El resultado se ve así:

```
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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
2005-03-23 23:47:11,673 - spam_application - INFO -
   done with auxiliary_module.some_function()
```

2 Logging desde múltiples hilos

Realizar *logging* desde múltiples hilos (*threads*) no requiere ningún esfuerzo especial. El siguiente ejemplo muestra el logging desde el hilo principal (inicial) y otro hilo:

```
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()
```

Cuando se ejecuta, el script debe imprimir algo como lo siguiente:

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

Esto muestra la salida de logging intercalada como cabría esperar. Por supuesto, este enfoque funciona para más hilos de lo que se muestran aquí.

3 Múltiples gestores y formateadores

Los *loggers* son simples objetos Python. El método `addHandler()` no tiene una cuota mínima o máxima para la cantidad de gestores (*handlers*) que puede agregar. A veces será beneficioso para una aplicación registrar todos los mensajes de todas las prioridades en un archivo de texto mientras se registran simultáneamente los errores o más en la consola. Para configurar esto, simplemente configure los gestores apropiados. Las llamadas de logging en el código de la aplicación permanecerán sin cambios. Aquí hay una ligera modificación al ejemplo de configuración simple anterior basado en módulo:

```
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')
```

Tenga en cuenta que el código de la "aplicación" no se preocupa por los gestores múltiples. Todo lo que cambió fue la adición y configuración de un nuevo gestor llamado *fh*.

La capacidad de crear nuevos gestores con filtros de mayor o menor prioridad puede ser muy útil al escribir y probar una aplicación. En lugar de usar muchas declaraciones `print` para la depuración, use `logger.debug`: a diferencia de las declaraciones de impresión, que tendrá que eliminar o comentar más tarde, las declaraciones de *logger.debug* pueden permanecer intactas en el código fuente y permanecen inactivas hasta que las necesite nuevamente. En ese momento, el único cambio que debe realizar es modificar el nivel de prioridad del *logger* y/o gestor para depurar.

4 Logging en múltiples destinos

Supongamos que desea que la consola y un archivo tengan diferentes formatos de mensaje y salida de log para diferentes situaciones. Por ejemplo, desea registrar mensajes con un nivel DEBUG y superiores en un archivo y enviar mensajes con nivel INFO y superior a la consola. Además, suponga que desea grabar una marca de tiempo en el archivo y no imprimirlo en la consola. Puede lograr este comportamiento haciendo lo siguiente:

```
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='/temp/myapp.log',
```

(continué en la próxima página)

(proviene de la página anterior)

```
                    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.')
```

Cuando ejecute esto, en la consola verá

```
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.
```

y en el archivo verá algo como

```
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.
```

Como se puede ver, el mensaje DEBUG sólo se muestra en el archivo. Los otros mensajes se envían a los dos destinos.

Este ejemplo usa gestores de consola y archivos, pero puede usar cualquier número y combinación de los gestores que elija.

5 Ejemplo de servidor de configuración

Aquí hay un ejemplo de un módulo que usa el servidor de configuración logging:

```
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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()
```

Y aquí hay un script que toma un nombre de archivo y envía ese archivo al servidor, precedido adecuadamente con la longitud codificada en binario, como la nueva configuración de logging:

```
#!/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')
```

6 Tratar con gestores que bloquean

A veces tiene que hacer que sus gestores de logging hagan su trabajo sin bloquear el hilo desde el que está iniciando sesión. Esto es común en las aplicaciones web, aunque, por supuesto, también ocurre en otros escenarios.

Un responsable habitual que ejemplifica un comportamiento lento es la `SMTPHandler`: el envío de correos electrónicos puede llevar mucho tiempo, por varias razones fuera del control del desarrollador (por ejemplo, una infraestructura de red o correo de bajo rendimiento). Pero casi cualquier controlador basado en red puede bloquear: incluso una operación `SocketHandler` puede estar haciendo a bajo nivel una consulta DNS que es demasiado lenta (y esta consulta puede estar en el código de la biblioteca de socket, debajo de la capa de Python, y fuera de su control).

Una solución es utilizar un enfoque de dos partes. Para la primera parte, adjunte solo una `QueueHandler` a los loggers que se acceden desde subprocesos críticos de rendimiento. Simplemente escriben en su cola, que puede dimensionarse a una capacidad lo suficientemente grande o inicializarse sin límite superior a su tamaño. La escritura en la cola generalmente se aceptará rápidamente, aunque es probable que deba atrapar la excepción `queue.Full` como precaución en su código. Si usted es un desarrollador de bibliotecas que tiene subprocesos críticos de rendimiento en su código, asegúrese de documentar esto (junto con una sugerencia de adjuntar solo `QueueHandlers` a sus loggers) para el beneficio de otros desarrolladores que usarán su código.

La segunda parte de la solución es `QueueListener`, que fue designado como la contraparte de `QueueHandler`.

Un `QueueListener` es muy simple: ha pasado una cola y algunos gestores, y activa un hilo interno que escucha su cola para *LogRecords* enviados desde `QueueHandlers` (o cualquier otra fuente de `LogRecords`, para el caso). Los `LogRecords` se eliminan de la cola y se pasan a los gestores para su procesamiento.

La ventaja de tener una clase separada `QueueListener` es que puede usar la misma instancia para dar servicio a múltiples `QueueHandlers`. Esto es más amigable con los recursos que, por ejemplo, tener versiones enhebradas de las clases de gestores existentes, que consumirían un hilo por gestor sin ningún beneficio particular.

Un ejemplo del uso de estas dos clases a continuación (se omiten *imports*):

```
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()
```

que, cuando se ejecuta, producirá:

```
MainThread: Look out!
```

Distinto en la versión 3.5: Antes de Python 3.5, `QueueListener` siempre pasaba cada mensaje recibido de la cola a cada controlador con el que se inicializaba. (Esto se debió a que se asumió que el filtrado de nivel se realizó en el otro lado, donde se llena la cola). A partir de 3.5, este comportamiento se puede cambiar pasando un argumento de palabra clave `respect_handler_level=True` al constructor del oyente . Cuando se hace esto, el oyente compara el nivel de cada mensaje con el nivel del controlador y solo pasa un mensaje a un controlador si es apropiado hacerlo.

7 Enviar y recibir eventos logging a través de una red

Supongamos que desea enviar eventos de registro a través de una red y manejarlos en el extremo receptor. Una forma sencilla de hacer esto es adjuntar una instancia de `SocketHandler` al registrador raíz en el extremo de envío:

```
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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
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.')
```

En el extremo receptor, puede configurar un receptor usando el módulo `socketserver`. Aquí hay un ejemplo básico de trabajo:

```
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.
    """

    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.
```

(continué en la próxima página)

(proviene de la página anterior)

```
    """

    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()
```

Primero ejecuta el servidor, y luego el cliente. Del lado del cliente, nada se imprime en la consola; del lado del servidor, se debería ver algo como esto:

```
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.
```

Tenga en cuenta que existen algunos problemas de seguridad con pickle en algunos escenarios. Si estos le afectan, puede usar un esquema de serialización alternativo anulando el método `makePickle()` e implementando su alternativa allí, así como adaptar el script anterior para usar su serialización alternativa.

7.1 Running a logging socket listener in production

To run a logging listener in production, you may need to use a process-management tool such as Supervisor. Here is a Gist which provides the bare-bones files to run the above functionality using Supervisor: you will need to change the */path/to/* parts in the Gist to reflect the actual paths you want to use.

8 Agregar información contextual a su salida de logging

A veces, desea que la salida de logging contenga información contextual además de los parámetros pasados a la llamada del logging. Por ejemplo, en una aplicación en red, puede ser conveniente registrar información específica del cliente en el logging (por ejemplo, el nombre de usuario del cliente remoto o la dirección IP). Aunque puede usar el parámetro *extra* para lograr esto, no siempre es conveniente pasar la información de esta manera. Si bien puede resultar tentador crear instancias `Logger` por conexión, esta no es una buena idea porque estas instancias no se liberan de memoria vía el recolector de basura (*garbage collector*). Si bien esto no es un problema en la práctica, cuando el número de instancias de `Logger` depende del nivel de granularidad que desea usar para hacer el logging de una aplicación, podría ser difícil de administrar si el número de instancias `Logger` se vuelven efectivamente ilimitadas.

8.1 Uso de LoggerAdapters para impartir información contextual

Una manera fácil de pasar información contextual para que se genere junto con la información de eventos logging es usar la clase `LoggerAdapter`. Esta clase está diseñada para parecerse a `Logger`, de modo que pueda llamar `debug()`, `info()`, `warning()`, `error()`, `excepción()`, `critical()` y `log()`. Estos métodos tienen las mismas signaturas que sus contrapartes en `Logger`, por lo que puede usar los dos tipos de instancias indistintamente.

Cuando creas una instancia de `LoggerAdapter`, le pasas una instancia de `Logger` y un objeto similar a un dict que contiene tu información contextual. Cuando llamas a uno de los métodos de registro en una instancia de `LoggerAdapter`, delega la llamada a la instancia subyacente de `Logger` pasada a su constructor, y se arregla para pasar la información contextual en la llamada delegada . Aquí hay un fragmento del código 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)
```

El método `process()` de `LoggerAdapter` es donde la información contextual se agrega a la salida del logging. Se pasa el mensaje y los argumentos de palabra clave de la llamada logging, y retorna versiones (potencialmente) modificadas de estos para usar en la llamada al logging subyacente. La implementación predeterminada de este método deja el mensaje solo, pero inserta una clave "extra" en el argumento de palabra clave cuyo valor es el objeto tipo dict pasado al constructor. Por supuesto, si ha pasado un argumento de palabra clave "extra" en la llamada al adaptador, se sobrescribirá silenciosamente.

La ventaja de usar "extra" es que los valores en el objeto dict se combinan en la instancia `LogRecord` __dict__, lo que le permite usar cadenas personalizadas con sus instancias `Formatter` que conocen las claves del objeto dict. Si necesita un método diferente, por ejemplo, si desea anteponer o agregar la información contextual a la cadena del mensaje, solo necesita la subclase `LoggerAdapter` y anular `process()` para hacer lo que necesita. Aquí hay un ejemplo 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.
    """
```

(continué en la próxima página)

(proviene de la página anterior)

```
    def process(self, msg, kwargs):
        return '[%s] %s' % (self.extra['connid'], msg), kwargs
```

que puede usar así:

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

Luego, cualquier evento que registre en el adaptador tendrá el valor de `some_conn_id` antepuesto a los mensajes de logging.

Usar objetos distintos a los diccionarios para transmitir información contextual

No es necesario pasar un diccionario real a la `LoggerAdapter` - puedes pasar una instancia de una clase que implemente `__getitem__` y `__iter__` de modo que parezca un diccionario para el logging. Esto es útil si quieres generar valores dinámicamente (mientras que los valores en un diccionario son constantes).

8.2 Usar filtros para impartir información contextual

También puedes agregar información contextual a la salida del log utilizando un `Filter` definido por el usuario. Las instancias de `Filter` pueden modificar los `LogRecords` que se les pasan, incluido el agregado de atributos adicionales que luego se pueden generar utilizando cadena de caracteres con el formato adecuado, o si es necesario, un `Formatter` personalizado.

Por ejemplo, en una aplicación web, la solicitud que se está procesando (o al menos, las partes interesantes de la misma) se pueden almacenar en una variable *threadlocal* (`threading.local`) y luego se puede acceder a ella desde `Filter` para agregar información de la solicitud, - digamos, la dirección IP remota y el nombre de usuario-, al `LogRecord`, usando los nombres de atributo "ip" y "user" como en el ejemplo anterior de `LoggerAdapter`. En ese caso, se puede usar el mismo formato de cadena de caracteres para obtener un resultado similar al que se muestra arriba. Aquí hay un script de ejemplo:

```
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)
    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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
    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')
```

que cuando se ejecuta, produce algo como:

```
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
```

9 Logging a un sólo archivo desde múltiples procesos

Aunque logging es seguro para hilos, y el logging a un solo archivo desde múltiples hilos en un solo proceso *es* compatible, el logging en un solo archivo desde *múltiples procesos no* es compatible, porque no existe una forma estándar de serializar el acceso a un solo archivo en múltiples procesos en Python. Si necesita hacer esto último, una forma de abordarlo es hacer que todos los procesos se registren en una `SocketHandler`, y tener un proceso separado que implemente un servidor de socket que lee del socket y los loggings para archivar. (Si lo prefiere, puede dedicar un hilo en uno de los procesos existentes para realizar esta función.) *Esta sección* documenta este enfoque con más detalle e incluye un receptor socket que funciona que se puede utilizar como punto de partida para que se adapte a sus propias aplicaciones.

También puedes escribir tu propio gestor que use la clase `Lock` del módulo `multiprocessing` para serializar el acceso al archivo desde tus procesos. La existente `FileHandler` y las subclases no hacen uso de `multiprocessing` en la actualidad, aunque pueden hacerlo en el futuro. Tenga en cuenta que, en la actualidad, el módulo `multiprocessing` no proporciona la funcionalidad de bloqueo de trabajo en todas las plataformas (ver https://bugs.python.org/issue3770).

Alternativamente, puede usar una `Queue` y `QueueHandler` para enviar todos los logging a uno de los procesos en su aplicación multi-proceso. El siguiente script de ejemplo demuestra cómo puede hacer esto; en el ejemplo, un proceso de escucha independiente escucha los eventos enviados por otros procesos y los registra de acuerdo con su

propia configuración de logging. Aunque el ejemplo solo demuestra una forma de hacerlo (por ejemplo, es posible que desee utilizar un hilo de escucha en lugar de un proceso de escucha separado; la implementación sería análoga), permite configuraciones de logging completamente diferentes para el oyente y los otros procesos en su aplicación. Y se puede utilizar como base para el código que cumpla con sus propios requisitos específicos:

```
# 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')
    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]
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()
```

Una variante del script anterior mantiene el logging en el proceso principal, en un hilo separado:

```
import logging
import logging.config
import logging.handlers
```

(continué en la próxima página)

(proviene de la página anterior)

```
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',
```

(continué en la próxima página)

(proviene de la página anterior)

```
                '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()
```

Esta variante muestra cómo puede, por ejemplo, aplicar la configuración para logging particulares: el registrador `foo` tiene un gestor especial que almacena todos los eventos en el subsistema `foo` en un archivo `mplog-foo.log`. Esto será utilizado por la maquinaria de logging en el proceso principal (aunque los eventos logging se generen en los procesos de trabajo) para dirigir los mensajes a los destinos apropiados.

9.1 Usando concurrent.futures.ProcessPoolExecutor

Si desea utilizar `concurrent.futures.ProcessPoolExecutor` para iniciar sus procesos de trabajo, debe crear la cola de manera ligeramente diferente. En vez de

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

debería usar

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

y luego puede reemplazar la creación del trabajador de esto:

```
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()
```

a esto (recuerda el primer *import* `concurrent.futures`):

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

9.2 Deploying Web applications using Gunicorn and uWSGI

When deploying Web applications using Gunicorn or uWSGI (or similar), multiple worker processes are created to handle client requests. In such environments, avoid creating file-based handlers directly in your web application. Instead, use a `SocketHandler` to log from the web application to a listener in a separate process. This can be set up using a process management tool such as Supervisor - see *Running a logging socket listener in production* for more details.

10 Usando rotación de archivos

A veces, se desea dejar que un archivo de log crezca hasta cierto tamaño y luego abra un nuevo archivo e inicie sesión en él. Es posible que desee conservar una cierta cantidad de estos archivos, y cuando se hayan creado tantos archivos, rote los archivos para que la cantidad de archivos y el tamaño de los archivos permanezcan limitados. Para este patrón de uso, el paquete logging proporciona `RotatingFileHandler`:

```
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(
              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)
```

El resultado debe ser 6 archivos separados, cada uno con parte del historial de log de la aplicación:

```
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
```

El archivo más actual siempre es `logging_rotatingfile_example.out`, y cada vez que alcanza el límite de tamaño, se le cambia el nombre con el sufijo“.1“. Se cambia el nombre de cada uno de los archivos de respaldo existentes para incrementar el sufijo (`.1` se convierte en `.2`, etc.) y se borra el archivo `.6`.

Obviamente, este ejemplo establece la longitud del log demasiado pequeña como un ejemplo extremo. Se querrá establecer *maxBytes* en un valor apropiado.

11 Uso de estilos de formato alternativos

Cuando se agregó logging a la biblioteca estándar de Python, la única forma de formatear mensajes con contenido variable era usar el método de formateo "%". Desde entonces, Python ha ganado dos nuevos enfoques de formato: `string.Template` (agregado en Python 2.4) y `str.format()` (agregado en Python 2.6).

Logging (a partir de la versión 3.2) proporciona un soporte mejorado para estos dos estilos de formato adicionales. La clase `Formatter` ha sido mejorada para tomar un parámetro de palabra clave adicional llamado `style`. El valor predeterminado es `'%'`, pero otros valores posibles son `'{'` y `'$'`, que corresponden a los otros dos estilos de formato. La compatibilidad con versiones anteriores se mantiene de forma predeterminada (como era de esperar), pero al especificar explícitamente un parámetro de estilo, tiene la capacidad de especificar cadenas de formato que funcionan con `str.format()` o `string.Template`. Aquí hay una sesión de consola de ejemplo para mostrar las posibilidades:

```
>>> 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
>>> 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
>>>
```

Tenga en cuenta que el formato de logging para la salida final a los logs es completamente independiente de cómo se construye un mensaje de logging individual. Para eso todavía puede usar el formateo «%», como se muestra aquí:

```
>>> 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
>>>
```

Las llamadas de Logging (`logger.debug()`, `logger.info()`, etc.) solo toman parámetros posicionales para el mensaje de logging real en sí, los parámetros de palabras clave se usan solo para determinar opciones sobre cómo gestionar la llamada propiamente a Logging (por ejemplo, el parámetro de palabra clave `exc_info` para indicar que la información de rastreo debe registrarse, o el parámetro de palabra clave `extra` para indicar información contextual adicional que se agregará al log). Por lo tanto, no puede realizar llamadas de logging directamente usando la sintaxis `str.format()` o `string.Template`, porque internamente el paquete de logging usa formato % para fusionar la cadena de formato y los argumentos de las variables. No habría ningún cambio en esto mientras se conserva la compatibilidad con versiones anteriores, ya que todas las llamadas de logging que están en el código existente usarán cadenas de formato %.

Sin embargo, existe una forma en la que puede usar el formato {} - y $ - para construir sus mensajes de log individuales. Recuerde que para un mensaje puede usar un objeto arbitrario como una cadena de caracteres de formato de mensaje, y que el paquete logging llamará a `str()` en ese objeto para obtener la cadena de caracteres de formato real. Considere las siguientes dos clases:

```
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)
```

Cualquiera de estos puede usarse en lugar de una cadena de formato, para permitir que se use el formato {} - o $ - para construir la parte del «mensaje» real que aparece en la salida del log en lugar de «%(message)s» o «{message}» o «$message». Es un poco difícil de manejar usar los nombres de las clases siempre que quieras registrar algo, pero es bastante aceptable si usas un alias como __ (doble subrayado — no confundir con _, el subrayado simple usado como sinónimo/alias para `gettext.gettext()` o sus hermanos).

Las clases anteriores no están incluidas en Python, aunque son bastante fáciles de copiar y pegar en su propio código. Se pueden usar de la siguiente manera (asumiendo que están declaradas en un módulo llamado `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
>>>
```

Si bien los ejemplos anteriores usan `print()` para mostrar cómo funciona el formateo, por supuesto usaría `logger.debug()` o similar para realmente registrar usando este enfoque.

Una cosa a tener en cuenta es que no paga una penalización de rendimiento significativa con este enfoque: el formateo real no ocurre cuando realiza la llamada a logging, sino cuando (y si) el mensaje registrado está a punto de ser enviado a un log por un gestor. Entonces, lo único un poco inusual que podría confundirte es que los paréntesis rodean la cadena de formato y los argumentos, no solo la cadena de formato. Eso es porque la notación __ es solo azúcar sintáctico para una llamada de constructor a una de las clases XXXMessage.

Si lo prefiere, puede usar `LoggerAdapter` para lograr un efecto similar al anterior, como en el siguiente ejemplo:

```
import logging

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

    def __str__(self):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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()
```

El script anterior debería registrar el mensaje `Hello, world!` Cuando se ejecuta con Python 3.2 o posterior.

12 Personalización de `LogRecord`

Cada evento logging está representado por una instancia `LogRecord`. Cuando se registra un evento y no se filtra por el nivel de un registrador, se crea `LogRecord`, se llena con información sobre el evento y luego se pasa a los gestores de ese registrador (y sus antepasados, hasta (e incluyéndolo) el registrador donde se deshabilita una mayor propagación en la jerarquía). Antes de Python 3.2, solo había dos lugares donde se realizaba esta creación:

- `Logger.makeRecord()`, que se llama en el proceso normal de logging de un evento. Esto invoca `LogRecord` directamente para crear una instancia.
- `makeLogRecord()`, que se llama con un diccionario que contiene atributos que se agregarán al *LogRecord*. Esto se suele invocar cuando se ha recibido un diccionario adecuado a través de la red (por ejemplo, en forma de *pickle* a través de `SocketHandler`, o en formato JSON a través de `HTTPHandler`).

Por lo general, esto significa que si necesita hacer algo especial con `LogRecord`, debe hacer una de las siguientes cosas.

- Cree su propia subclase `Logger`, que anula `Logger.makeRecord()`, y configúrelo usando `setLoggerClass()` antes de que se creen instancias de los registradores que le interesan.
- Agrega un `Filter` a un registrador o gestor, que realiza la manipulación especial necesaria que necesita cuando se llama a su método `filter()`.

El primer enfoque sería un poco difícil de manejar en el escenario en el que (digamos) varias bibliotecas diferentes quisieran hacer cosas diferentes. Cada uno intentaría establecer su propia subclase `Logger`, y el que hiciera esto último ganaría.

El segundo enfoque funciona razonablemente bien en muchos casos, pero no le permite, por ejemplo, usar una subclase especializada de `LogRecord`. Los desarrolladores de bibliotecas pueden establecer un filtro adecuado en sus registradores, pero tendrían que recordar hacerlo cada vez que introduzcan un nuevo registrador (lo que harían simplemente agregando nuevos paquetes o módulos y haciendo

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

a nivel de módulo). Probablemente sean demasiadas cosas en las que pensar. Los desarrolladores también podrían agregar el filtro a `NullHandler` adjunto a su registrador de nivel superior, pero esto no se invocaría si un desarrollador de aplicaciones adjuntara un controlador a un registrador de biblioteca de nivel inferior — así que la salida de ese gestor no reflejaría las intenciones del desarrollador de la biblioteca.

En Python 3.2 y posteriores, la creación de `LogRecord` se realiza a través de una fábrica, que puede especificar. La fábrica es simplemente un invocable que puede configurar con `setLogRecordFactory()`, e interrogar con `getLogRecordFactory()`. La fábrica se invoca con la misma firma que el constructor `LogRecord`, ya que `LogRecord` es la configuración predeterminada de la fábrica.

Este enfoque permite que una fábrica personalizada controle todos los aspectos de la creación de *LogRecord*. Por ejemplo, podría devolver una subclase, o simplemente agregar algunos atributos adicionales al registro una vez creado, usando un patrón similar a este:

```
old_factory = logging.getLogRecordFactory()

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

logging.setLogRecordFactory(record_factory)
```

Este patrón permite que diferentes bibliotecas encadenen fábricas juntas, y siempre que no sobrescriban los atributos de las demás o sobrescriban involuntariamente los atributos proporcionados como estándar, no debería haber sorpresas. Sin embargo, debe tenerse en cuenta que cada eslabón de la cadena agrega una sobrecarga de tiempo de ejecución a todas las operaciones de logging, y la técnica solo debe usarse cuando el uso de `Filter` no proporciona el resultado deseado.

13 Subclasificación QueueHandler - un ejemplo de ZeroMQ

Puede usar una subclase `QueueHandler` para enviar mensajes a otros tipos de colas, por ejemplo, un socket de "publicación" ZeroMQ. En el siguiente ejemplo, el socket se crea por separado y se pasa al gestor (como su "cola"):

```
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)
```

Por supuesto, hay otras formas de organizar esto, por ejemplo, pasando los datos que necesita el gestor para crear el socket:

```
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()
```

14 Subclasificación QueueListener - un ejemplo de ZeroMQ

También puede subclasificar `QueueListener` para obtener mensajes de otros tipos de colas, por ejemplo, un socket de "suscripción" de ZeroMQ. Aquí tienes un ejemplo:

```
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)
```

Ver también:

Módulo `logging` Referencia de API para el módulo logging.

Módulo `logging.config` API de configuración para el módulo logging.

Módulo `logging.handlers` Gestores útiles incluidos con el módulo logging.

Un tutorial básico de logging

Un tutorial de logging más avanzado

15 Una configuración de ejemplo basada en diccionario

A continuación se muestra un ejemplo de un diccionario de configuración de logging, tomado de la documentación del proyecto Django. Este diccionario se pasa a `dictConfig()` para poner en efecto la configuración:

```
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'
```

(continué en la próxima página)

(proviene de la página anterior)

```
        },
        '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']
        }
    }
}
```

Para obtener más información sobre esta configuración, puede ver la sección correspondiente de la documentación de Django.

16 Usar un rotador y un nombre para personalizar el procesamiento de rotación de log

Un ejemplo de cómo puede definir un nombre y un rotador se da en el siguiente fragmento, que muestra la compresión basada en zlib del archivo de log:

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

def rotator(source, dest):
    with open(source, "rb") as sf:
        data = sf.read()
        compressed = zlib.compress(data, 9)
        with open(dest, "wb") as df:
            df.write(compressed)
    os.remove(source)

rh = logging.handlers.RotatingFileHandler(...)
rh.rotator = rotator
rh.namer = namer
```

Estos no son archivos .gz «verdaderos», ya que son datos comprimidos sin ningún «contenedor» como el que encontraría en un archivo gzip real. Este fragmento es solo para fines ilustrativos.

17 Un ejemplo de multiprocesamiento más elaborado

El siguiente ejemplo de trabajo muestra cómo logging se puede usar con multiprocesamiento usando archivos de configuración. Las configuraciones son bastante simples, pero sirven para ilustrar cómo se podrían implementar las más complejas en un escenario real de multiprocesamiento.

En el ejemplo, el proceso principal genera un proceso de escucha y algunos procesos de trabajo. Cada uno de los procesos principales, el oyente y los trabajadores tienen tres configuraciones separadas (todos los trabajadores comparten la misma configuración). Podemos ver el registro en el proceso principal, cómo los trabajadores se registran en un QueueHandler y cómo el oyente implementa un QueueListener y una configuración de registro más compleja, y organiza el envío de eventos recibidos a través de la cola a los controladores especificados en la configuración. Tenga en cuenta que estas configuraciones son puramente ilustrativas, pero debería poder adaptar este ejemplo a su propio escenario.

Aquí está el script, el docstrings y los comentarios, esperemos, expliquen cómo funciona:

```
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
        # parent process, but should have been disabled following the
        # dictConfig call.
        # On Windows, since fork isn't used, the setup logger won't
```

(continué en la próxima página)

(proviene de la página anterior)

```
        # 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
```

(continué en la próxima página)

(proviene de la página anterior)

```
    # 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'
            }
```

(continué en la próxima página)

(proviene de la página anterior)

```
        },
        '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()
```

18 Insertar BOM en mensajes enviados a SysLogHandler

RFC 5424 requiere que se envíe un mensaje Unicode a un demonio syslog como un conjunto de bytes que tienen la siguiente estructura: un componente opcional ASCII puro, seguido de una marca de orden de bytes UTF-8 (BOM), seguida de Codificado en Unicode usando UTF-8. (Ver sección relevante de la especificación **RFC 5424#section-6**.)

En Python 3.1, se agregó código a `SysLogHandler` para insertar BOM en el mensaje, pero desafortunadamente, se implementó incorrectamente, BOM aparece al principio del mensaje y, por lo tanto, no permite ningún componente ASCII puro para que aparezca antes.

Como este comportamiento no funciona, el código de inserción BOM incorrecto se elimina de Python 3.2.4 y versiones posteriores. Sin embargo, no se está reemplazando, y si desea producir mensajes compatibles con **RFC 5424** que incluyan BOM, una secuencia opcional de ASCII puro antes y Unicode arbitrario después, codificados usando UTF-8; entonces necesita hacer lo siguiente:

1. Adjunte una instancia de `Formatter` a su instancia `SysLogHandler`, con una cadena de caracteres de formato como:

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

El punto de código Unicode U+ FEFF, cuando se codifica usando UTF-8, se codificará como una BOM UTF-8, la cadena de bytes `b'\xef\xbb\xbf'`.

2. Reemplace la sección ASCII con los marcadores de posición que desee, pero asegúrese de que los datos que aparecen allí después de la sustitución sean siempre ASCII (de esa manera, permanecerán sin cambios después de la codificación UTF-8).
3. Reemplace la sección Unicode con los marcadores de posición que desee; si los datos que aparecen allí después de la sustitución contienen caracteres fuera del rango ASCII, está bien: se codificarán usando UTF-8.

El mensaje formateado *se* codificará utilizando la codificación UTF-8 por `SysLogHandler`. Si sigue las reglas anteriores, debería poder producir mensajes compatibles con **RFC 5424**. Si no lo hace, es posible que el logging no se queje, pero sus mensajes no serán compatibles con RFC 5424 y su demonio syslog puede quejarse.

19 Implementar logging estructurado

Aunque la mayoría de los mensajes de registro están destinados a ser leídos por humanos y, por lo tanto, no se pueden analizar fácilmente mediante una máquina, puede haber circunstancias en las que desee generar mensajes en un formato estructurado que *sea* capaz de ser analizado por un programa (sin necesidad de expresiones regulares complejas para analizar el mensaje de registro). Esto es sencillo de lograr utilizando el paquete de registro. Hay varias formas de lograr esto, pero el siguiente es un enfoque simple que usa JSON para serializar el evento de una manera analizable por máquina:

```
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 se ejecuta el script anterior, se imprime:

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

Tenga en cuenta que el orden de los elementos puede ser diferente según la versión de Python utilizada.

Si necesita un procesamiento más especializado, puede utilizar un codificador JSON personalizado, como en el siguiente ejemplo completo:

```
from __future__ import unicode_literals

import json
import logging

# This next bit is to ensure the script runs unchanged on 2.x and 3.x
try:
    unicode
except NameError:
    unicode = str
```

(continué en la próxima página)

(proviene de la página anterior)

```
class Encoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, unicode):
            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()
```

Cuando se ejecuta el script anterior, se imprime:

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

Tenga en cuenta que el orden de los elementos puede ser diferente según la versión de Python utilizada.

20 Personalización de gestores con `dictConfig()`

Hay ocasiones en las que desea personalizar los gestores de logging de formas particulares, y si usa `dictConfig()` puede hacerlo sin subclases. Como ejemplo, considere que es posible que desee establecer la propiedad de un archivo de log. En POSIX, esto se hace fácilmente usando `shutil.chown()`, pero los gestores de archivos en stdlib no ofrecen soporte integrado. Puede personalizar la creación de gestores usando una función simple como:

```
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)
```

Luego puede especificar, en una configuración de logging pasada a `dictConfig()`, que se cree un gestor de logging llamando a esta función:

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

(continué en la próxima página)

(proviene de la página anterior)

```
    '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',
    },
}
```

En este ejemplo, se establece la propiedad utilizando el usuario y el grupo `pulse`, solo con fines ilustrativos. Poniéndolo junto en un script de trabajo, `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',
```

(continué en la próxima página)

(proviene de la página anterior)

```
    },
}

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

Para ejecutar esto, probablemente se necesite ejecutarlo como `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
```

Tenga en cuenta que este ejemplo usa Python 3.3 porque ahí es donde `shutil.chown()` aparece. Este enfoque debería funcionar con cualquier versión de Python que admita `dictConfig()`, es decir, Python 2.7, 3.2 o posterior. Con las versiones anteriores a 3.3, necesitaría implementar el cambio de propiedad real usando, por ejemplo, `os.chown()`.

En la práctica, la función de creación de gestores puede estar en un módulo de utilidad en algún lugar de su proyecto. En lugar de la línea en la configuración:

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

podrías usar, por ejemplo,:

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

donde `project.util` se puede reemplazar con el nombre real del paquete donde reside la función. En el script de trabajo anterior, el uso de `'ext://__main__.owned_file_handler'` debería funcionar. Aquí, el invocable real se resuelve mediante `dictConfig()` de la especificación `ext://`.

Por fortuna, este ejemplo también indica el camino hacia cómo podría implementar otros tipos de cambio de archivo, por ejemplo, configurando de la misma manera bits de permisos POSIX específicos, usando `os.chmod()`.

Por supuesto, el enfoque también podría extenderse a tipos de gestores distintos a `FileHandler` - por ejemplo, uno de los gestores de archivos rotativos, o un tipo diferente por completo.

21 Usar estilos de formato particulares en toda su aplicación

En Python 3.2, `Formatter` obtuvo un parámetro de palabra clave `estilo` que, aunque por defecto era `%` para compatibilidad con versiones anteriores, permitía la especificación de `{` o `$` para permitir los enfoques de formato admitidos por `str.format()` y `string.Template`. Tenga en cuenta que esto rige el formato de los mensajes de logging para la salida final a los logging y es completamente ortogonal a cómo se construye un mensaje de logging individual.

Las llamadas de logging (`debug()`, `info()`, etc.) solo toman parámetros posicionales para el mensaje logging real en sí, con parámetros de palabras clave que se utilizan solo para determinar las opciones sobre cómo manejar la llamada de logging (por ejemplo, el parámetro de palabra clave `exc_info` para indicar que la información de rastreo debe registrarse, o el parámetro de palabra clave `extra` para indicar información contextual adicional que se agregará al log). Por lo tanto, no puede realizar llamadas de logging directamente usando la sintaxis `str.format()` o `string.Template`, porque internamente el paquete logging usa formato % para fusionar la cadena de formato y los argumentos de las variables. No se cambiaría esto mientras se conserve la compatibilidad con versiones anteriores, ya que todas las llamadas de logging que están en el código existente utilizarán cadenas de caracteres formato %.

Ha habido sugerencias para asociar estilos de formato con *loggers* específicos, pero ese enfoque también tiene problemas de compatibilidad con versiones anteriores porque cualquier código existente podría estar usando un nombre de *logger* dado y usando formato %.

Para que logging funcione de manera interoperable entre cualquier biblioteca de terceros y su código, las decisiones sobre el formato deben tomarse a nivel de la llamada de logging individual. Esto abre un par de formas en las que se pueden acomodar estilos de formato alternativos.

21.1 Uso de fábricas de LogRecord

En Python 3.2, junto con los cambios de `Formatter` mencionados anteriormente, el paquete logging ganó la capacidad de permitir a los usuarios establecer sus propias subclases `LogRecord`, usando la función `setLogRecordFactory()`. Puede usar esto para configurar su propia subclase de `LogRecord`, que hace lo correcto al anular el método `getMessage()`. La implementación de la clase base de este método es donde ocurre el formato `msg % args` y donde puede sustituir su formato alternativo; sin embargo, debe tener cuidado de admitir todos los estilos de formato y permitir formato % como predeterminado, para garantizar la interoperabilidad con otro código. También se debe tener cuidado de llamar a `str(self.msg)`, tal como lo hace la implementación base.

Consulte la documentación de referencia en `setLogRecordFactory()` y `LogRecord` para obtener más información.

21.2 Usar objetos de mensaje personalizados

Existe otra forma, quizás más sencilla, de usar el formato {} - y $ - para construir sus mensajes de log individuales. Al iniciar sesión, recuerde que puede usar cualquier objeto como una cadena de caracteres de formato de mensaje (arbitrary-object-messages) que al iniciar sesión puede usar un objeto arbitrario como una cadena de formato de mensaje, y que el paquete de logging llamará `str()` en ese objeto para obtener el cadena de formato real. Considere las siguientes dos clases:

```
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)
```

Cualquiera de estos puede usarse en lugar de una cadena de formato, para permitir que se use el formato {} - o $ - para construir la parte del «mensaje» real que aparece en la salida del log formateado en lugar de “%(message)s” or “{message}” or “$message”. Si le resulta un poco difícil de manejar usar los nombres de las clases cada vez que desea registrar algo, puede hacerlo más tolerable si usa un alias como `M` o _ para el mensaje (o quizás __, si está utilizando“_“ para localización).

A continuación se dan ejemplos de este enfoque. En primer lugar, formatear con `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
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> print(__('Message with coordinates: ({point.x:.2f}, {point.y:.2f})', point=p))
Message with coordinates: (0.50, 0.50)
```

En segundo lugar, formatear con `string.Template`:

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

Una cosa a tener en cuenta es que no paga ninguna penalización significativa del rendimiento con este enfoque: el formato real no se produce cuando se realiza la llamada logging, sino cuando (y si) el mensaje registrado está realmente a punto de ser salida a un log por un gestor. Así que lo único un poco inusual con lo que podría tropezar es que los paréntesis van alrededor de la cadena de caracteres de formato y los argumentos, no sólo la cadena de formato. Esto se debe a que la notación __ es solo azúcar sintáctico para una llamada de constructor a una de las clases `XXXMessage` mostradas anteriormente.

22 Configurar filtros con `dictConfig()`

Puedes configurar filtros usando `dictConfig()`, aunque a primera vista es posible que no sea obvio cómo hacerlo (de ahí esta receta). Dado que `Filter` es la única clase de filtro incluida en la biblioteca estándar, y es poco probable que satisfaga muchos requisitos (solo está allí como clase base), normalmente necesitarás definir tu propia subclase `Filter` con un método `filter()` sobreescrito. Para hacer esto, especifique la clave `()` en el diccionario de configuración para el filtro, especificando un invocable que se usará para crear el filtro (una clase es la más obvia, pero puede proporcionar cualquier invocable que devuelva una instancia `Filter`). Aquí hay un ejemplo completo:

```
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': {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

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

Este ejemplo muestra cómo puede pasar datos de configuración al invocable que construye la instancia, en forma de parámetros de palabras clave. Cuando se ejecuta, se imprimirá el script anterior:

```
changed: hello
```

que muestra que el filtro está funcionando según lo configurado.

Un par de puntos adicionales a tener en cuenta:

- Si no puede hacer referencia al invocable directamente en la configuración (por ejemplo, si vive en un módulo diferente y no puede importarlo directamente donde está el diccionario de configuración), puede usar el formulario `ext://...` como se describe en logging-config-dict-externalobj. Por ejemplo, podría haber usado el texto `'ext://__main__.MyFilter'` en lugar de `MyFilter` en el ejemplo anterior.
- Además de los filtros, esta técnica también se puede utilizar para configurar gestores y formateadores personalizados. Consultar logging-config-dict-userdef para obtener más información sobre cómo logging admite el uso de objetos definidos por el usuario en su configuración, y ver arriba la otra receta *Personalización de gestores con dictConfig()*.

23 Formato de excepción personalizado

Puede haber ocasiones en las que desee personalizar un formato de excepción; por el bien del argumento, digamos que desea exactamente una línea por evento registrado, incluso cuando la información de la excepción está presente. Puede hacer esto con una clase de formateador personalizada, como se muestra en el siguiente ejemplo:

```
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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
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()
```

Cuando se ejecuta, esto produce un archivo con exactamente dos líneas:

```
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'|
```

Si bien el tratamiento anterior es simplista, señala el camino hacia cómo se puede formatear la información de excepción a su gusto. El módulo `traceback` puede resultar útil para necesidades más especializadas.

24 Mensajes de logging hablantes

Puede haber situaciones en las que sea deseable que los mensajes de logging se presenten en un formato audible en lugar de visible. Esto es fácil de hacer si tiene la funcionalidad de texto a voz (*TTS* por sus siglas en inglés) disponible en su sistema, incluso si no tiene un *binding* Python. La mayoría de los sistemas TTS tienen un programa de línea de comandos que puede ejecutar, y esto puede invocarse desde un gestor usando `subprocess`. Aquí se asume que los programas de línea de comando TTS no esperarán interactuar con los usuarios o tardarán mucho en completarse, y que la frecuencia de los mensajes registrados no será tan alta como para inundar al usuario con mensajes, y que es aceptable que los mensajes se reproducen uno a la vez en lugar de todos al mismo tiempo. La implementación de ejemplo a continuación espera a que se pronuncie un mensaje antes de que se procese el siguiente, y esto puede hacer que otros gestores se mantengan esperando. Aquí hay un breve ejemplo que muestra el enfoque, que asume que el paquete TTS `espeak` está disponible:

```
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')
```

(continué en la próxima página)

(proviene de la página anterior)

```
    logging.debug('Goodbye')

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

Cuando se ejecute, este script debería decir «Hola» y luego «Adiós» con voz femenina.

El enfoque anterior puede, por supuesto, adaptarse a otros sistemas TTS e incluso a otros sistemas que pueden procesar mensajes a través de programas externos ejecutados desde una línea de comando.

25 Almacenamiento en búfer de mensajes de logging y su salida condicional

Puede haber situaciones en las que desee registrar mensajes en un área temporal y solo mostrarlos si se produce una determinada condición. Por ejemplo, es posible que desee comenzar a registrar eventos de depuración en una función, y si la función se completa sin errores, no desea saturar el log con la información de depuración recopilada; pero si hay un error, desea toda la información de depuración información así como el error.

Aquí hay un ejemplo que muestra cómo puede hacer esto usando un decorador para sus funciones donde desea que el logging se comporte de esta manera. Hace uso de `logging.handlers.MemoryHandler`, que permite el almacenamiento en búfer de eventos registrados hasta que se produzca alguna condición, momento en el que los eventos almacenados en búfer se `flushed` y se pasan a otro gestor (el gestor `target`) para su procesamiento. De forma predeterminada, el `MemoryHandler` se vacía cuando su búfer se llena o se ve un evento cuyo nivel es mayor o igual a un umbral especificado. Puede usar esta receta con una subclase más especializada de `MemoryHandler` si desea un comportamiento de descarga personalizado.

El script de ejemplo tiene una función simple, `foo`, que recorre todos los niveles de logging, escribiendo en `sys.stderr` para decir en qué nivel está a punto de *loguear* y luego registrar un mensaje en ese nivel. Puede pasar un parámetro a `foo` que, si es verdadero, se registrará en los niveles ERROR y CRITICAL; de lo contrario, solo registrará en los niveles DEBUG, INFO y WARNING.

El script simplemente dispone decorar `foo` con un decorador que hará el logging condicional que se requiere. El decorador toma un registrador como parámetro y adjunta un gestor de memoria durante la duración de la llamada a la función decorada. El decorador se puede parametrizar adicionalmente utilizando un gestor *target*, un nivel en el que debe producirse el vaciado y una capacidad para el búfer (número de registros almacenados en búfer). Estos están predeterminados a `StreamHandler` que escribe en“sys.stderr“, `logging.ERROR` y `100` respectivamente.

Aquí está el 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):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        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)

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)
```

Cuando se ejecuta este script, se debe observar el siguiente resultado:

```
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
```

(continué en la próxima página)

(proviene de la página anterior)

```
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
```

Como puede ver, la salida real de logging solo ocurre cuando se registra un evento cuya gravedad es ERROR o mayor, pero en ese caso, también se registra cualquier evento anterior con una gravedad menor.

Por supuesto, puede utilizar las formas de decoración convencionales:

```
@log_if_errors(logger)
def foo(fail=False):
    ...
```

26 Formateo de horas usando UTC (GMT) a través de la configuración

A veces desea formatear las horas usando UTC, lo que se puede hacer usando una clase como *UTCFormatter*, como se muestra a continuación:

```
import logging
import time

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

y luego puede usar el `UTCFormatter` en su código en lugar de `Formatter`. Si desea hacer eso a través de la configuración, puede usar la API `dictConfig()` con un enfoque ilustrado por el siguiente ejemplo completo:

```
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',
```

(continué en la próxima página)

(proviene de la página anterior)

```
        },
        '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())
```

Cuando se ejecuta este script, debería imprimir algo como:

```
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
```

mostrando cómo se formatea la hora como hora local y UTC, una para cada gestor.

27 Usar un administrador de contexto para logging selectivo

Hay ocasiones en las que sería útil cambiar temporalmente la configuración de logging y revertir esto después de hacer algo. Para ello, un administrador de contexto es la forma más obvia de guardar y restaurar el contexto de logging. Aquí hay un ejemplo simple de un administrador de contexto de este tipo, que le permite cambiar opcionalmente el nivel de logging y agregar un gestor de logging exclusivamente en el alcance del administrador de contexto:

```
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
```

Si especifica un valor de nivel, el nivel del registrador se establece en ese valor en el alcance del bloque *with* cubierto por el administrador de contexto. Si especifica un gestor, se agrega al registrador al entrar al bloque y se elimina al salir del bloque. También puede pedirle al administrador que cierre el gestor por usted al salir del bloque si ya no lo necesita.

Para ilustrar cómo funciona, podemos agregar el siguiente bloque de código al anterior:

```
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.')
```

Inicialmente configuramos el nivel del registrador en `INFO`, por lo que aparece el mensaje #1 y el mensaje #2 no. Luego cambiamos el nivel a `DEBUG` temporalmente en el siguiente bloque `with`, y aparece el mensaje #3. Una vez que se sale del bloque, el nivel del registrador se restaura a `INFO` y, por lo tanto, el mensaje #4 no aparece. En el siguiente bloque `with`, configuramos el nivel en `DEBUG` nuevamente, pero también agregamos un gestor que escribe en `sys.stdout`. Por lo tanto, el mensaje #5 aparece dos veces en la consola (una vez a través de `stderr` y una vez a través de `stdout`). Después de la finalización de la declaración `with`, se vuelve al estado anterior, por lo que aparece el mensaje #6 (como el mensaje #1) mientras que el mensaje #7 no (como el mensaje #2).

Si ejecutamos el script resultante, el resultado es el siguiente:

```
$ 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.
```

Si lo ejecutamos de nuevo, pero dirigimos `stderr` a `/dev/null`, vemos lo siguiente, que es el único mensaje escrito en `stdout`:

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

Una vez más, pero canalizando `stdout` a `/dev/null`, obtenemos:

```
$ 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.
```

En este caso, el mensaje #5 impreso en `stdout` no aparece, como se esperaba.

Por supuesto, el enfoque descrito aquí puede generalizarse, por ejemplo, para adjuntar filtros de logging temporalmente. Tenga en cuenta que el código anterior funciona tanto en Python 2 como en Python 3.

28 Una plantilla de inicio de aplicación CLI

Aquí hay un ejemplo que muestra cómo puede:

- Utilizar un nivel de logging basado en argumentos de la línea de comandos
- Enviar a varios subcomandos en archivos separados, todos registrando en el mismo nivel de forma coherente
- Utilizar una configuración mínima y sencilla

Supongamos que tenemos una aplicación de línea de comandos cuyo trabajo es detener, iniciar o reiniciar algunos servicios. Esto podría organizarse con fines ilustrativos como un archivo `app.py` que es el script principal de la aplicación, con comandos individuales implementados en `start.py`, `stop.py` y `restart.py`. Supongamos además que queremos controlar la verbosidad de la aplicación a través de un argumento de línea de comandos, por defecto en `logging.INFO`. Aquí hay una forma en que se podría escribir `app.py`:

```
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())
```

Y los comandos `start`, `stop` y `reiniciar` se pueden implementar en módulos separados, como para iniciar:

```
# start.py
import logging

logger = logging.getLogger(__name__)
```

(continué en la próxima página)

(proviene de la página anterior)

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

y así para detener:

```
# 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)
```

y de manera similar para reiniciar:

```
# 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)
```

Si ejecutamos esta aplicación con el nivel de log predeterminado, obtenemos un resultado como este:

```
$ 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.
```

La primera palabra es el nivel de logging y la segunda palabra es el nombre del módulo o paquete del lugar donde se registró el evento.

Si cambiamos el nivel de logging, podemos cambiar la información enviada al log. Por ejemplo, si queremos más información:

```
$ 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.
```

Y si queremos menos:

```
$ 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
```

En este caso, los comandos no imprimen nada en la consola, ya que no registran nada en el nivel de `WARNING` o superior.

29 Una GUI de Qt para logging

Una pregunta que surge de vez en cuando es sobre cómo *loggear* en una aplicación GUI. El *framework Qt* <*https://www.qt.io/*> _ es un *framework* popular multiplataforma con bindings de Python que usan PySide2 <https://pypi.org/project/PySide2/>‘_ o librerías PyQt5 .

El siguiente ejemplo muestra cómo iniciar sesión en una GUI de Qt. Esto introduce una clase simple `QtHandler` que toma un invocable, que debería ser un slot en el hilo principal que realiza actualizaciones de la GUI. También se crea un hilo de trabajo para mostrar cómo puede iniciar sesión en la GUI tanto desde la propia interfaz de usuario (a través de un botón para el logging manual) como desde un hilo de trabajo que trabaja en segundo plano (aquí, simplemente registrando mensajes en niveles aleatorios con aleatorio breves retrasos intermedios).

El hilo *worker* se implementa usando la clase `QThread` de Qt en lugar del módulo `threading`, ya que hay circunstancias en las que uno tiene que usar `QThread`, que ofrece una mejor integración con otros componentes `Qt`.

El código debería funcionar con versiones recientes de `PySide2` o `PyQt5`. Debería poder adaptar el enfoque a versiones anteriores de Qt. Consulte los comentarios en el fragmento de código para obtener información más detallada.

```
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__)
```

(continué en la próxima página)

(proviene de la página anterior)

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

(continué en la próxima página)

(proviene de la página anterior)

```
        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().isInterruptionRequested():
            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('nosuchfont')
        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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
        # 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)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    app = QtWidgets.QApplication(sys.argv)
    example = Window(app)
    example.show()
    sys.exit(app.exec_())

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

30 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.

30.1 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.

Use the techniques outlined in *Logging a un sólo archivo desde múltiples procesos* to circumvent such issues.

30.2 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.

30.3 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.

30.4 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).

Índice

R