HOWTO Hacer Registros (Logging)

Versión 3.10.13

Guido van Rossum
and the Python development team

noviembre 14, 2023

Python Software Foundation
Email: docs@python.org

Índice general

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

1 Tutorial Básico de *Logging*

Logging es un medio de rastrear los eventos que ocurren cuando se ejecuta algún software. El desarrollador del software agrega llamadas de registro a su código para indicar que ciertos eventos han ocurrido. Un evento se describe mediante un mensaje descriptivo que puede contener opcionalmente datos variables (es decir, datos que son potencialmente diferentes para cada ocurrencia del evento). Los eventos también tienen una importancia que el desarrollador atribuye al evento; la importancia también puede llamarse el *nivel* o la *severidad*.

1.1 Cuándo usar *logging*

Logging proporciona un conjunto de funciones convenientes para un uso sencillo de registro. Estas son `debug()`, `info()`, `warning()`, `error()` y `critical()`. Para determinar cuándo usar el registro, vea la tabla de abajo, que indica, para cada una de las tareas comunes, la mejor herramienta a usar para ello.

La tarea que quieres realizar	La mejor herramienta para la tarea
Mostrar salidas de la consola para el uso ordinario de un programa o guión (script) de línea de comandos	`print()`
Reportar eventos que ocurren durante el funcionamiento normal de un programa (por ejemplo, para la supervisión del estado o la investigación de fallos)	`logging.info()` (o `logging.debug()` para salidas de registro muy detalladas con fines de diagnóstico)
Emitir una advertencia con respecto a un evento de tiempo de ejecución en particular	`warnings.warn()` en el código de la biblioteca si el problema es evitable y la aplicación cliente debe ser modificada para eliminar la advertencia `logging.warning()` si no hay nada que la aplicación cliente pueda hacer sobre la situación, pero el evento debe ser anotado
Reportar un error con respecto a un evento particular al tiempo de ejecución	Lanza una excepción
Reporta la supresión de un error sin invocar una excepción (por ejemplo, el manejador de errores en un proceso de servidor de larga duración)	`logging.error()`, `logging.exception()` o `logging.critical()` según sea apropiado para el error específico y el dominio de la aplicación

Las funciones de registro se denominan según el nivel o la gravedad de los eventos que se utilizan para rastrear. A continuación se describen los niveles estándar y su aplicabilidad (en orden creciente de gravedad):

Nivel	Cuando es usado
`DEBUG`	Información detallada, típicamente de interés sólo durante el diagnóstico de problemas.
`INFO`	Confirmación de que las cosas están funcionando como se esperaba.
`WARNING`	Un indicio de que algo inesperado sucedió, o indicativo de algún problema en el futuro cercano (por ejemplo, «espacio de disco bajo»). El software sigue funcionando como se esperaba.
`ERROR`	Debido a un problema más grave, el software no ha sido capaz de realizar alguna función.
`CRITICAL`	Un grave error, que indica que el programa en sí mismo puede ser incapaz de seguir funcionando.

El nivel por defecto es `WARNING`, lo que significa que sólo los eventos de este nivel y superiores serán rastreados, a menos que el paquete de registro esté configurado para hacer lo contrario.

Los eventos que se rastrean pueden ser manejados en diferentes maneras. La forma más simple de manejar los eventos rastreados es imprimirlos en la consola o terminal. Otra forma común es escribirlos en un archivo de disco.

1.2 Un simple ejemplo

Un ejemplo muy simple es:

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

Si escribes estas líneas en un script y lo ejecutas, verás:

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

impreso en la consola. El mensaje `INFO` no aparece porque el nivel por defecto es `WARNING`. El mensaje impreso incluye la indicación del nivel y la descripción del evento proporcionado en la llamada de registro, es decir, «¡Cuidado!». No te preocupes por la parte de la 'root' por ahora: se explicará más adelante. La salida real puede ser formateada con bastante flexibilidad si lo necesita; las opciones de formato también se explicarán más adelante.

1.3 Logging a un archivo

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

```
import logging
logging.basicConfig(filename='example.log', encoding='utf-8', level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
logging.error('And non-ASCII stuff, too, like Øresund and Malmö')
```

Distinto en la versión 3.9: Se agregó el argumento *encoding*. En versiones anteriores de Python, o si no se especifica, la codificación utilizada es el valor predeterminado utilizado por `open()`. Aunque no se muestra en el ejemplo anterior, ahora también se puede pasar un argumento *errors*, que determina cómo se manejan los errores de codificación. Para conocer los valores disponibles y los predeterminados, consulte la documentación de `open()`.

Y ahora si abrimos el archivo y miramos lo que tenemos, deberíamos encontrar los mensajes de registro:

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
ERROR:root:And non-ASCII stuff, too, like Øresund and Malmö
```

Este ejemplo también muestra cómo se puede establecer el nivel de registro que actúa como umbral para el rastreo. En este caso, como establecimos el umbral en `DEBUG`, todos los mensajes fueron impresos.

Si quieres establecer el nivel de registro desde una opción de línea de comandos como:

```
--log=INFO
```

y tienes el valor del parámetro pasado por `--log` en alguna variable *loglevel*, puedes usar:

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

para obtener el valor que pasarás a `basicConfig()` mediante el argumento *level*. Puede que quieras comprobar un error por cualquier valor de entrada del usuario, quizás como en el siguiente ejemplo:

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
```

(continué en la próxima página)

(proviene de la página anterior)

```
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

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

Si ejecutas el script anterior varias veces, los mensajes de las ejecuciones sucesivas se añaden al archivo *example.log*. Si quieres que cada ejecución se inicie de nuevo, sin recordar los mensajes de ejecuciones anteriores, puedes especificar el argumento *filemode*, cambiando la llamada en el ejemplo anterior a:

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

La impresión será la misma que antes, pero el archivo de registro ya no se adjunta, por lo que los mensajes de las ejecuciones anteriores se pierden.

1.4 *Logging* de múltiples módulos

Si su programa consiste de múltiples módulos, aquí hay un ejemplo de cómo podría organizar el inicio de sesión en él:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

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

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

Si ejecutas *myapp.py*, deberías ver esto en *myapp.log*:

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

que es lo que esperabas ver. Puedes generalizar esto a múltiples módulos, usando el modelo en *mylib.py*. Ten en cuenta que para este simple patrón de uso, no sabrás, mirando en el archivo de registro, *donde* en tu aplicación vinieron tus mensajes, aparte de mirar la descripción del evento. Si quieres rastrear la ubicación de tus mensajes, tendrás que consultar la documentación más allá del nivel del tutorial – ver *Tutorial de registro avanzado*.

1.5 Registrar datos de variables

Para registrar los datos de variables, utilice una cadena de formato para el mensaje de descripción del evento y añada los datos variables como argumentos. Por ejemplo:

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

se mostrará:

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

Como puede ver, la fusión de datos variables en el mensaje de descripción del evento utiliza el antiguo estilo % de formato de cadena de caracteres. Esto es por compatibilidad con versiones anteriores: el paquete de registro es anterior a opciones de formato más nuevas como `str.format()` y `string.Template`. Estas nuevas opciones de formato *son* compatibles, pero explorarlas está fuera del alcance de este tutorial: consulte formatting-styles para obtener más información.

1.6 Cambiar el formato de los mensajes mostrados

Para cambiar el formato que se utiliza para visualizar los mensajes, es necesario especificar el formato que se desea utilizar:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

que se imprimirá:

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

Noten que la 'root' que aparecía en los ejemplos anteriores ha desaparecido. Para un conjunto completo de cosas que pueden aparecer en formato de cadenas, puede consultar la documentación de logrecord-attributes, pero para un uso sencillo, sólo necesita el *levelname* (gravedad), *message* (descripción del evento, incluyendo los datos variables) y tal vez mostrar cuándo ocurrió el evento. Esto se describe en la siguiente sección.

1.7 Visualización de la fecha/hora en los mensajes

Para mostrar la fecha y la hora de un evento, usted colocaría "%(asctime)s" en su cadena de formato:

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

que debería imprimir algo como esto:

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

El formato por defecto para la visualización de la fecha/hora (mostrado arriba) es como ISO8601 o **RFC 3339**. Si necesita más control sobre el formato de la fecha/hora, proporcione un argumento *datefmt* a `basicConfig`, como en este ejemplo:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p
↪')
logging.warning('is when this event was logged.')
```

que mostraría algo como esto:

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

El formato del argumento *datefmt* es el mismo que el soportado por `time.strftime()`.

1.8 Próximos pasos

Eso concluye el tutorial básico. Debería ser suficiente para ponerte en marcha con el registro. Hay mucho más que el paquete de registro ofrece, pero para obtener lo mejor de él, tendrá que invertir un poco más de su tiempo en la lectura de las siguientes secciones. Si estás listo para eso, toma un poco de tu bebida favorita y sigue adelante.

Si sus necesidades de registro son sencillas, utilice los ejemplos previos para incorporar el registro en sus propios scripts, y si tiene problemas o no entiende algo, por favor publique una pregunta en el grupo Usenet de comp.lang.python (disponible en https://groups.google.com/forum/#!forum/comp.lang.python) y debería recibir ayuda antes de que transcurra demasiado tiempo.

¿Todavía esta aquí? Puedes seguir leyendo las siguientes secciones, que proporcionan un tutorial un poco más avanzado y profundo que el básico de arriba. Después de eso, puedes echar un vistazo al logging-cookbook.

2 Tutorial de registro avanzado

La biblioteca de logging adopta un enfoque modular y ofrece varias categorías de componentes: registradores, gestores, filtros y formateadores.

- Los registradores exponen la interfaz que el código de la aplicación utiliza directamente.
- Los gestores envían los registros de log (creados por los registradores) al destino apropiado.
- Los filtros proporcionan una instalación de grano más fino para determinar qué registros de log se deben producir.
- Los formatos especifican la disposición de los archivos de log en el resultado final.

La información de los eventos de registro se pasa entre los registradores, gestores, filtros y formateadores en una instancia `LogRecord`.

El registro se realiza llamando a métodos en instancias de la clase `Logger` (de aquí en adelante llamada *loggers*). Cada instancia tiene un nombre, y se organizan conceptualmente en una jerarquía de espacios de nombres utilizando puntos (puntos) como separadores. Por ejemplo, un registrador llamado “scan” es el padre de los registradores “scan.text”, “scan.html” y “scan.pdf”. Los nombres de los registradores pueden ser cualquier cosa que se desee, e indican el área de una aplicación en la que se origina un mensaje registrado.

Una buena convención que se puede utilizar para nombrar a los registradores es utilizar un registrador a nivel de módulo, en cada módulo que utilice el registro, llamado de la siguiente manera:

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

Esto significa que los nombres de los registradores rastrean la jerarquía de paquetes/módulos, y es intuitivamente obvio donde se registran los eventos sólo a partir del nombre del registrador.

La raíz de la jerarquía de los registradores se llama root *logger*. Ese es el registrador usado por las funciones `debug()`, `info()`, `warning()`, `error()` y `critical()`, que sólo llaman al mismo método del registrador raíz. Las funciones y los métodos tienen las mismas firmas. El nombre del root *logger* se imprime como ‘root’ en la salida registrada.

Por supuesto, es posible registrar mensajes a diferentes destinos. El paquete incluye soporte para escribir mensajes de registro en archivos, ubicaciones HTTP GET/POST, correo electrónico a través de SMTP, sockets genéricos, colas o mecanismos de registro específicos del sistema operativo como syslog o el registro de eventos de Windows NT. Los destinos son servidos por clases *handler*. Puedes crear tu propia clase de destino de registro si tienes requisitos especiales que no se cumplen con ninguna de las clases de gestor incorporadas.

Por defecto, no se establece ningún destino para los mensajes de registro. Puede especificar un destino (como consola o archivo) usando `basicConfig()` como en los ejemplos del tutorial. Si llama a las funciones `debug()`, `info()`, `warning()`, `error()` y `critical()`, ellas comprobarán si no hay ningún destino establecido; y si no hay ninguno establecido, establecerán un destino de la consola (`sys.stderr`) y un formato por defecto para el mensaje mostrado antes de delegar en el registrador root para hacer la salida real del mensaje.

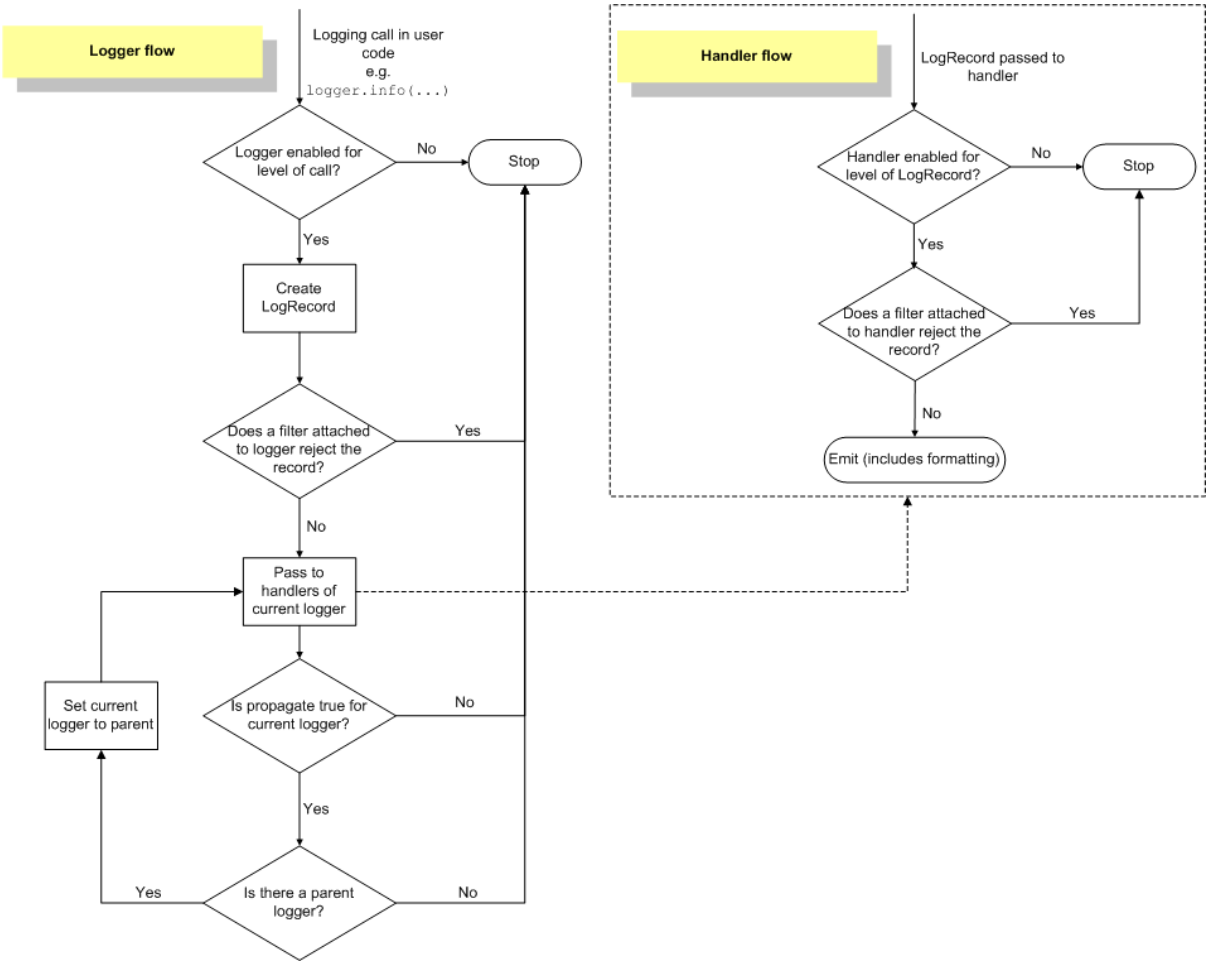
El formato por defecto establecido por `basicConfig()` para los mensajes es:

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

Puedes cambiar esto pasando una cadena de formato a `basicConfig()` con el argumento de la palabra clave *format*. Para todas las opciones relativas a cómo se construye una cadena de formato, ver formatter-objects.

2.1 Flujo de Registro

En el siguiente diagrama se ilustra el flujo de información de los eventos de registro en los registradores y gestores.



2.2 Registradores

Los objetos de `Logger` tienen un trabajo triple. Primero, exponen varios métodos al código de la aplicación para que las aplicaciones puedan registrar mensajes en tiempo de ejecución. Segundo, los objetos *logger* determinan sobre qué mensajes de registro actuar en base de la severidad (la facilidad de filtrado por defecto) o los objetos de filtro. Tercero, los objetos registradores pasan los mensajes de registro relevantes a todos los manejadores de registro interesados.

Los métodos más utilizados en los objetos de registro se dividen en dos categorías: configuración y envío de mensajes.

Estos son los métodos de configuración más comunes:

- `Logger.setLevel()` especifica el mensaje de registro de menor gravedad que un registrador manejará, donde debug es el nivel de gravedad incorporado más bajo y *critical* es el de mayor gravedad incorporado. Por ejemplo, si el nivel de severidad es INFO, el registrador sólo manejará los mensajes INFO, WARNING, ERROR y CRITICAL e ignorará los mensajes DEBUG.
- `Logger.addHandler()` y `Logger.removeHandler()` agregan y quitan los objetos *handler* del objeto *logger*. Los manejadores (*handlers*) se tratan con más detalle en *Gestores*.
- `Logger.addFilter()` y `Logger.removeFilter()` agregan y quitan los objetos de filtro del objeto logger. Los filtros se tratan con más detalle en filter.

No es necesario que siempre llames a estos métodos en cada registrador que crees. Vea los dos últimos párrafos de esta sección.

Con el objeto *logger* configurado, los siguientes métodos crean mensajes de log:

- `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()`, y `Logger.critical()` todos crean registros de registro con un mensaje y un nivel que corresponde a sus respectivos nombres de método. El mensaje es en realidad una cadena de formato, que puede contener la sintaxis estándar de sustitución de cadenas de `%s`, `%d`, `%f`, y así sucesivamente. El resto de sus argumentos es una lista de objetos que se corresponden con los campos de sustitución del mensaje. Con respecto a `**kwargs`, los métodos de registro sólo se preocupan por una palabra clave de `exc_info` y la usan para determinar si registran información de excepción.
- `Logger.exception()` crea un mensaje de registro similar a `Logger.error()`. La diferencia es que `Logger.exception()` vuelca un rastro de pila junto con él. Llama a este método sólo desde un manejador de excepciones.
- `Logger.log()` toma un nivel de log como argumento explícito. Esto es un poco más verboso para el registro de mensajes que usar los métodos de conveniencia de nivel de registro listados arriba, pero así es como se registra en niveles de registro personalizados.

`getLogger()` retorna una referencia a una instancia de *logger* con el nombre especificado si se proporciona, o `root` si no. Los nombres son estructuras jerárquicas separadas por períodos. Múltiples llamadas a `getLogger()` con el mismo nombre retornarán una referencia al mismo objeto *logger*. Los *loggers* que están más abajo en la lista jerárquica son hijos de los *loggers* que están más arriba en la lista. Por ejemplo, dado un *logger* con un nombre de `foo`, los *loggers* con nombres de `foo.bar`, `foo.bar.baz`, y `foo.bam` son todos descendientes de `foo`.

Los registradores tienen un concepto de *nivel efectivo*. Si un nivel no se establece explícitamente en un registrador, el nivel de su clase padre se utiliza en su lugar como su nivel efectivo. Si el padre no tiene un nivel explícito establecido, *su* padre es examinado, y así sucesivamente - se buscan todos los ancestros hasta que se encuentra un nivel explícitamente establecido. El registrador raíz siempre tiene un conjunto de niveles explícito (`Advertencia` por defecto). Cuando se decide si se procesa un evento, el nivel efectivo del registrador se utiliza para determinar si el evento se pasa a los manejadores del registrador.

Los *loggers* inferiores propagan mensajes hasta los gestores asociados con sus *loggers* ancestros. Debido a esto, no es necesario definir y configurar los manejadores para todos los registradores que utiliza una aplicación. Basta con configurar los manejadores para un registrador de nivel superior y crear registradores hijos según sea necesario. (Sin embargo, puedes desactivar la propagación estableciendo el atributo *propagate* de un *logger* en `False`.)

2.3 Gestores

Los objetos `Handler` son responsables de enviar los mensajes de registro apropiados (basados en la severidad de los mensajes de registro) al destino especificado por el handler. `Logger` los objetos pueden añadir cero o más objetos *handler* a sí mismos con un método `addHandler()`. Como escenario de ejemplo, una aplicación puede querer enviar todos los mensajes de registro a un archivo de registro, todos los mensajes de registro de error o superiores a stdout, y todos los mensajes de crítico a una dirección de correo electrónico. Este escenario requiere tres manejadores individuales donde cada manejador es responsable de enviar mensajes de una severidad específica a una ubicación específica.

La biblioteca estándar incluye bastantes tipos de *handler* (ver *Gestores útiles*); los tutoriales usan principalmente `StreamHandler` y `FileHandler` en sus ejemplos.

Hay muy pocos métodos en un manejador para que los desarrolladores de aplicaciones se preocupen. Los únicos métodos de manejador que parecen relevantes para los desarrolladores de aplicaciones que utilizan los objetos de manejador incorporados (es decir, que no crean manejadores personalizados) son los siguientes métodos de configuración:

- El método `setLevel()`, al igual que en los objetos de *logger*, especifica la menor gravedad que será enviada al destino apropiado. ¿Por qué hay dos métodos `setLevel()`? El nivel establecido en el registrador determina qué gravedad de los mensajes pasará a sus manejadores. El nivel establecido en cada manejador determina qué mensajes enviará ese manejador.
- `setFormatter()` selecciona un objeto *Formatter* para que este *handler* lo use.
- `addFilter()` y `removeFilter()` respectivamente configuran y desconfiguran los objetos del filtro en los handlers.

El código de la aplicación no debe instanciar directamente y usar instancias de `Handler`. En su lugar, la clase `Handler` es una clase base que define la interfaz que todos los *handlers* deben tener y establece algún comportamiento por defecto que las clases hijas pueden usar (o anular).

2.4 Formateadores

Los objetos de formato configuran el orden final, la estructura y el contenido del mensaje de registro. A diferencia de la clase base `logging.Handler`, el código de la aplicación puede instanciar clases de formateo, aunque probablemente podría subclasificar el formateo si su aplicación necesita un comportamiento especial. El constructor toma tres argumentos opcionales – una cadena de formato de mensaje, una cadena de formato de fecha y un indicador de estilo.

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

Si no hay una cadena de formato de mensaje, el valor predeterminado es utilizar el mensaje en bruto. Si no hay una cadena de formato de fecha, el formato de fecha por defecto es:

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

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

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

Distinto en la versión 3.2: Añadió el parámetro `style`.

La siguiente cadena de formato de mensaje registrará la hora en un formato legible para los humanos, la gravedad del mensaje y el contenido del mensaje, en ese orden:

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

Los formateadores utilizan una función configurable por el usuario para convertir la hora de creación de un registro en una tupla. Por defecto, se utiliza `time.localtime()`; para cambiar esto para una instancia de formateador particular, establezca el atributo `converter` de la instancia a una función con la misma firma que `time.localtime()` o `time.gmtime()`. Para cambiarlo para todos los formateadores, por ejemplo si quieres que todas las horas de registro se muestren en GMT, establece el atributo `converter` en la clase *Formatter* (a `time.gmtime` para mostrar GMT).

2.5 Configuración del registro

Los programadores pueden configurar el registro en tres maneras:

1. Creando registradores, manejadores y formateadores explícitamente usando código Python que llama a los métodos de configuración listados arriba.
2. Creando un archivo de configuración de registro y leyéndolo usando la función `fileConfig()`.
3. Creando un diccionario de información de configuración y pasándolo a la función `dictConfig()`.

Para la documentación de referencia sobre las dos últimas opciones, vea logging-config-api. El siguiente ejemplo configura un *logger* muy simple, un manejador de consola, y un formateador simple usando código Python:

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

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

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

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

Ejecutar este módulo desde la línea de comandos produce la siguiente salida:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

El siguiente módulo de Python crea un registrador, manejador y formateador casi idéntico a los del ejemplo anterior, con la única diferencia de los nombres de los objetos:

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

(continué en la próxima página)

(proviene de la página anterior)

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

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

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

Aquí está el archivo logging.conf:

```
[loggers]
keys=root,simpleExample

[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

La salida es casi idéntica a la del ejemplo basado en un archivo no configurado:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

Se puede ver que el enfoque del archivo de configuración tiene algunas ventajas sobre el enfoque del código Python, principalmente la separación de la configuración y el código y la capacidad de los no codificadores de modificar fácilmente las propiedades de registro.

Advertencia: La función `fileConfig`() toma un parámetro por defecto, `disable_existing_loggers`, que por defecto es `True` por razones de compatibilidad retroactiva. Esto puede ser o no lo que usted quiera, ya que causará que cualquier registrador no existente antes de la llamada `fileConfig()` sea desactivado a menos que ellos (o un ancestro) sean nombrados explícitamente en la

configuración. Por favor, consulte la documentación de referencia para más información, y especifique `False` para este parámetro si lo desea.

El diccionario pasado a `dictConfig()` también puede especificar un valor booleano con la tecla `disable_existing_loggers`, que si no se especifica explícitamente en el diccionario también se interpreta por defecto como `True`. Esto lleva al comportamiento de deshabilitación de los registradores descrito anteriormente, que puede no ser lo que usted desea - en cuyo caso, proporcione a la clave explícitamente un valor de `False`.

Obsérvese que los nombres de clase a los que se hace referencia en los archivos de configuración deben ser relativos al módulo de registro, o bien valores absolutos que puedan resolverse mediante mecanismos de importación normales. Por lo tanto, puedes usar `WatchedFileHandler` (relativo al módulo de registro) o `mypackage.mymodule.MyHandler` (para una clase definida en el paquete `mypackage` y el módulo `mymodule`, donde `mypackage` está disponible en la ruta de importación de Python).

En Python 3.2, se ha introducido un nuevo medio para configurar el registro, utilizando diccionarios para guardar la información de configuración. Esto proporciona un superconjunto de la funcionalidad del enfoque basado en archivos de configuración descrito anteriormente, y es el método de configuración recomendado para nuevas aplicaciones y despliegues. Dado que se utiliza un diccionario de Python para guardar información de configuración, y dado que se puede rellenar ese diccionario utilizando diferentes medios, se dispone de más opciones de configuración. Por ejemplo, puede utilizar un archivo de configuración en formato JSON o, si tiene acceso a la funcionalidad de procesamiento YAML, un archivo en formato YAML, para rellenar el diccionario de configuración. O, por supuesto, puedes construir el diccionario en código Python, recibirlo en forma encurtida sobre un zócalo, o usar cualquier enfoque que tenga sentido para tu aplicación.

Aquí hay un ejemplo de la misma configuración que arriba, en formato YAML para el nuevo enfoque basado en el diccionario:

```yaml
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

Para más información sobre el registro usando un diccionario, ver logging-config-api.

2.6 ¿Qué pasa si no se proporciona ninguna configuración

Si no se proporciona una configuración de registro, es posible que se produzca una situación en la que sea necesario dar salida a un suceso de registro, pero no se puede encontrar a ningún gestor para dar salida al suceso. El comportamiento del paquete de registro en estas circunstancias depende de la versión de Python.

Para las versiones de Python anteriores a la 3.2, el comportamiento es el siguiente:

- Si *logging.raiseExceptions* es `Falso` (modo de producción), el evento es abandonado silenciosamente.
- Si *logging.raiseExceptions* es `True` (modo de desarrollo), se imprime una vez un mensaje "*No handlers could be found for logger X.Y.Z*" ("No se pudo encontrar ningún manejador (*handler*) para el *logger* X.Y.Z").

En Python 3.2 y posteriores, el comportamiento es el siguiente:

- El evento es emitido usando un 'handler de último recurso', almacenado en `logging.lastResort`. Este manejador interno no está asociado con ningún logger, y actúa como un `StreamHandler`` que escribe el mensaje de descripción del evento con el valor actual de `sys.stderr` (respetando así cualquier redireccionamiento que pueda estar en vigor). No se hace ningún tipo de formateo en el mensaje, sólo se imprime el mensaje de descripción del evento. El nivel del manejador se establece en `WARNING`, por lo que todos los eventos de esta y mayores severidades serán emitidos.

Para obtener el comportamiento anterior a la 3.2, `logging.lastResort` se puede configurar como `None`.

2.7 Configurando Logging para una biblioteca

Cuando se desarrolla una biblioteca que utiliza el registro, se debe tener cuidado de documentar la forma en que la biblioteca utiliza el registro, por ejemplo, los nombres de los registradores utilizados. También hay que tener en cuenta su configuración de registro. Si la aplicación que lo utiliza no usa el registro, y el código de la biblioteca hace llamadas de registro, entonces (como se describe en la sección anterior) los eventos de gravedad `WARNING` y mayores se imprimirán en `sys.stderr`. Esto se considera el mejor comportamiento por defecto.

Si por alguna razón usted *no* quiere que estos mensajes se impriman en ausencia de cualquier configuración de registro, puede adjuntar un manejador de no hacer nada al registrador de nivel superior de su biblioteca. Esto evita que el mensaje se imprima, ya que siempre se encontrará un manejador para los eventos de la biblioteca: simplemente no produce ninguna salida. Si el usuario de la biblioteca configura el registro para el uso de la aplicación, presumiblemente esa configuración añadirá algunos manejadores, y si los niveles están configurados adecuadamente, entonces las llamadas de registro realizadas en el código de la biblioteca enviarán una salida a esos manejadores, como es normal.

Un manejador de no hacer nada está incluido en el paquete de registro: `NullHandler` (desde Python 3.1). Una instancia de este manejador podría ser añadida al *logger* de nivel superior del espacio de nombres de registro usado por la biblioteca (*si* quieres evitar que los eventos de registro de tu biblioteca se envíen a `sys.stderr` en ausencia de la configuración de registro). Si todo el registro de una biblioteca *foo* se hace usando registradores con nombres que coincidan con "foo.x", "foo.x.y", etc. entonces el código:

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

debería tener el efecto deseado. Si una organización produce varias bibliotecas, el nombre del registrador especificado puede ser 'orgname.foo' en lugar de sólo 'foo'.

Nota: Se recomienda encarecidamente que *no añada ningún otro manejador que no sea* `NullHandler` *a los loggers de su biblioteca*. Esto se debe a que la configuración de los *handlers* es prerrogativa del desarrollador de aplicaciones que utiliza su biblioteca. El desarrollador de la aplicación conoce su público objetivo y qué manejadores son los más apropiados para su aplicación: si añades manejadores 'bajo el capó', podrías interferir en su capacidad de realizar pruebas unitarias y entregar registros que se ajusten a sus necesidades.

---

3 Niveles de registro

Los valores numéricos de los niveles de registro se indican en el siguiente cuadro. Éstos son de interés principalmente si se desea definir los propios niveles y se necesita que tengan valores específicos en relación con los niveles predefinidos. Si se define un nivel con el mismo valor numérico, éste sobrescribe el valor predefinido; el nombre predefinido se pierde.

Nivel	Valor numérico
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

Los niveles también pueden asociarse con los registradores, siendo establecidos por el desarrollador o mediante la carga de una configuración de registro guardada. Cuando se llama a un método de registro en un registrador, éste compara su propio nivel con el nivel asociado a la llamada del método. Si el nivel del registrador es superior al de la llamada al método, no se genera ningún mensaje de registro. Este es el mecanismo básico que controla la verbosidad de la salida del registro.

Los mensajes de registro se codifican como instancias de la clase `logging.LogRecord`. Cuando un *logger* decide registrar realmente un evento, se crea una instancia `LogRecord` a partir del mensaje de registro.

Los mensajes de registro están sujetos a un mecanismo de envío mediante el uso de *handlers*, que son instancias de subclases de la clase `Handler`. Los gestores son responsables de asegurar que un mensaje registrado (en forma de `LogRecord`) termine en una ubicación particular (o conjunto de ubicaciones) que sea útil para el público al que va dirigido ese mensaje (como usuarios finales, personal de asistencia técnica, administradores de sistemas, desarrolladores). Los manejadores pasan instancias `LogRecord` destinadas a destinos particulares. Cada *logger* puede tener cero, uno o más manejadores asociados a él (a través del método `addHandler()` de `Logger`). Además de los *handlers* directamente asociados a un *logger*, todos los manejadores (*handlers*) asociados a todos los ancestros del *logger* son llamados a enviar el mensaje (a menos que el flag *propagate* de un *logger* se establezca en un valor falso, en cuyo caso el paso a los *handlers* ancestrales se detiene).

Al igual que para los *logger*, los gestores pueden tener niveles asociados a ellos. El nivel de un gestor actúa como un filtro de la misma manera que el nivel de un *logger*. Si un manejador (*handler*) decide realmente enviar un evento, el método `emit()` se utiliza para enviar el mensaje a su destino. La mayoría de las subclases definidas por el usuario de `Handler` necesitarán anular este `emit()`.

3.1 Niveles personalizados

Definir sus propios niveles es posible, pero no debería ser necesario, ya que los niveles existentes se han elegido sobre la base de la experiencia práctica. Sin embargo, si usted está convencido de que necesita niveles personalizados, debe tener mucho cuidado al hacer esto, y es posiblemente *una muy mala idea definir niveles personalizados si está desarrollando una biblioteca*. Esto se debe a que si los autores de múltiples bibliotecas definen sus propios niveles personalizados, existe la posibilidad de que el resultado del registro de tales bibliotecas múltiples utilizadas conjuntamente sea difícil de controlar y/o interpretar para el desarrollador usuario, porque un valor numérico dado podría significar cosas diferentes para diferentes bibliotecas.

4 Gestores útiles

Además de la base `Handler` class, se proporcionan muchas subclases útiles:

1. `StreamHandler` instancias envían mensajes a los *streams* (objetos como de tipo archivo).
2. `FileHandler` instancias enviar mensajes a los archivos del disco.
3. `BaseRotatingHandler` es la clase base para los manejadores (*handlers*) que rotan los archivos de registro en un punto determinado. No está pensada para ser instanciada directamente. En su lugar, utilice `RotatingFileHandler` o `TimedRotatingFileHandler`.
4. Las instancias de `RotatingFileHandler` envían mensajes a los archivos de disco, con soporte para el tamaño máximo de los archivos de registro y la rotación de los mismos.
5. Las instancias de `TimedRotatingFileHandler` envían mensajes a los archivos de disco, rotando el archivo de registro a ciertos intervalos de tiempo.
6. Las instancias de `SocketHandler` envían mensajes a los sockets TCP/IP. Desde la versión 3.4, los sockets de dominio Unix también están soportados.
7. Instancias de `DatagramHandler` envían mensajes a los sockets UDP. Desde la versión 3.4, los sockets de dominio Unix también están soportados.
8. Las instancias de `SMTPHandler` envían mensajes a una dirección de correo electrónico designada.
9. Las instancias de `SysLogHandler` envían mensajes a un demonio del syslog de Unix, posiblemente en una máquina remota.
10. Las instancias de `NTEventLogHandler` envían mensajes a un registro de eventos de Windows NT/2000/XP.
11. Las instancias de `MemoryHandler` envían mensajes a un buffer en la memoria, que es limpiado cuando se cumplen ciertos criterios.
12. Las instancias de `HTTPHandler` envían mensajes a un servidor HTTP usando la semántica de «GET» o «POST».
13. Las instancias de `WatchedFileHandler` ven el archivo al que están accediendo. Si el archivo cambia, se cierra y se vuelve a abrir usando el nombre del archivo. Este manejador sólo es útil en sistemas tipo Unix; Windows no soporta el mecanismo subyacente utilizado.
14. Las instancias de `QueueHandler` envían mensajes a una cola, como los implementados en los módulos `queue` or `multiprocessing`.
15. `NullHandler` instancias no hacen nada con los mensajes de error. Son utilizadas por los desarrolladores de bibliotecas que quieren utilizar el registro, pero quieren evitar el mensaje «No se pudo encontrar ningún controlador para el registrador XXX», que puede mostrarse si el usuario de la biblioteca no ha configurado el registro. Vea *Configurando Logging para una biblioteca* para más información.

Nuevo en la versión 3.1: La clase `NullHandler`.

Nuevo en la versión 3.2: La `QueueHandler` (La clase de gestores de Cola).

Las clases `NullHandler`, `StreamHandler` y `FileHandler` están definidas en el paquete de registro del núcleo. Los otros manejadores se definen en un sub-módulo, `logging.handlers`. (También hay otro submódulo, `logging.config`, para la funcionalidad de configuración)

Los mensajes registrados se formatean para su presentación a través de instancias de la clase `Formatter`. Se inicializan con una cadena de formato adecuada para su uso con el operador % y un diccionario.

Para dar formato a varios mensajes en un lote, se pueden utilizar instancias de `BufferingFormatter`. Además de la cadena de formato (que se aplica a cada mensaje del lote), hay una provisión para cadenas de formato de cabecera y de tráiler.

Cuando el filtrado basado en el nivel de *logger* o el nivel de manejador (*handler*) no es suficiente, se pueden añadir instancias de `Filter` tanto a `Logger` como a `Handler` instancias (a través de su método `addFilter()`). Antes

de decidir procesar un mensaje más adelante, tanto los *loggers* como los manejadores (*handlers*) consultan todos sus filtros para obtener permiso. Si algún filtro retorna un valor falso, el mensaje no se procesa más.

La funcionalidad básica `Filtro` permite filtrar por un nombre de registro específico. Si se utiliza esta función, los mensajes enviados al registrador nombrado y a sus hijos se permiten a través del filtro, y todos los demás se eliminan.

5 Excepciones lanzadas durante logging

El paquete de tala está diseñado para tragarse las excepciones que se producen durante la tala en la producción. Esto es así para que los errores que ocurren durante el manejo de los eventos de registro - como la mala configuración del registro, errores de red u otros errores similares - no causen que la aplicación que utiliza el registro termine prematuramente.

Las excepciones de `SystemExit` (Salida del sistema) y `KeyboardInterrupt` (Interrupción del teclado) nunca se tragan. Otras excepciones que ocurren durante el método `emit()` de una subclase `Handler` se pasan a su método `handleError()`.

La implementación por defecto de `handleError()` en `Handler` comprueba si una variable de nivel de módulo, `raiseExceptions`, está establecida. Si se establece, se imprime una traza en `sys.stderr`. Si no se establece, se traga la excepción.

Nota: El valor por defecto de `raiseExceptions` (lanzar excepciones) es `True`. Esto se debe a que durante el desarrollo, normalmente quieres ser notificado de cualquier excepción que ocurra. Se aconseja que establezca `raiseExceptions` a `False` para el uso en producción.

---

6 Usando objetos arbitrarios como mensajes

En las secciones y ejemplos anteriores, se ha supuesto que el mensaje pasado al registrar el suceso es una cadena. Sin embargo, esta no es la única posibilidad. Se puede pasar un objeto arbitrario como mensaje, y su método `__str__()` será llamado cuando el sistema de registro necesite convertirlo en una representación de cadena. De hecho, si quieres, puedes evitar computar una representación de cadena por completo - por ejemplo, el método `SocketHandler` emite un evento al *pickling* y enviarlo por el cable.

7 Optimización

El formato de los argumentos del mensaje se aplaza hasta que no se pueda evitar. Sin embargo, el cálculo de los argumentos pasados al método de registro también puede ser costoso, y puede que quieras evitar hacerlo si el registrador simplemente tirará tu evento. Para decidir qué hacer, puedes llamar al método `isEnabledFor()` que toma un argumento de nivel y retorna true si el evento sería creado por el *Logger* para ese nivel de llamada. Puedes escribir código como este:

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

de modo que si el umbral del registrador se establece por encima de `DEBUG`, las llamadas a `expensive_func1()` y `expensive_func2()` nunca se hacen.

Nota: En algunos casos, `isEnabledFor()` puede ser en sí mismo más caro de lo que te gustaría (por ejemplo, para los *loggers* profundamente anidados donde un nivel explícito sólo se establece en lo alto de la jerarquía de *loggers*). En estos casos (o si quieres evitar llamar a un método en bucles estrechos), puedes guardar en caché el resultado de una llamada a `isEnabledFor()` en una variable local o de instancia, y usarla en lugar de llamar al método cada

vez. Tal valor en caché sólo necesitaría ser recalculado cuando la configuración de registro cambie dinámicamente mientras la aplicación se está ejecutando (lo cual no es tan común).

Hay otras optimizaciones que pueden hacerse para aplicaciones específicas que necesitan un control más preciso sobre la información de registro que se recoge. Aquí hay una lista de cosas que puede hacer para evitar el procesamiento durante el registro que no necesita:

Lo que no quieres colectar	Cómo evitar coleccionarlo
Información sobre dónde se hicieron las llamadas.	Establezca `logging._srcfile` en `None`. Esto evita llamar a `sys._getframe()`, que puede ayudar a acelerar su código en entornos como PyPy (que no puede acelerar el código que usa `sys._getframe()`).
Información sobre código con hilos.	Establece `logging.logThreads` en `False`.
ID del proceso actual (`os.getpid()`)	Establece `logging.logProcesses` en `False`.
Nombre del proceso actual cuando se usa `multiprocessing` para administrar múltiples procesos.	Establece `logging.logMultiprocessing` en `False`.

Observe también que el módulo de registro del núcleo sólo incluye los gestores básicos. Si no importas `logging.handlers` y `logging.config`, no ocuparán ninguna memoria.

Ver también:

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

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

Módulo `logging.handlers` Gestores útiles incluidos en el módulo de registro.

Un libro de recetas

Índice

No alfabético

R