# Logging in Python

Tabla de contenido



El registro (logging) es una herramienta muy útil en la caja de herramientas de un programador. Puede ayudarnos a desarrollar una mejor comprensión del flujo de un programa y descubrir escenarios en los que tal vez ni siquiera hayamos pensado durante el desarrollo.

Los registros proporcionan a los desarrolladores un par de ojos adicional que están constantemente mirando el flujo por el que atraviesa una aplicación. Pueden almacenar información, como qué usuario o IP accedió a la aplicación. Si se produce un error, pueden proporcionar más información que un seguimiento de la pila indicándole cuál era el estado del programa antes de que llegara a la línea de código donde ocurrió el error.

Al registrar datos útiles de los lugares correctos, no solo puede depurar errores fácilmente, sino que también podemos utilizar los datos para analizar el rendimiento de la aplicación para planificar el escalado o ver patrones de uso para planificar el marketing.

Python proporciona un sistema de registro como parte de su biblioteca estándar, por lo que se puede agregar registros rápidamente a nuestra aplicación. En este artículo, aprenderemos por qué usar este módulo es la mejor manera de agregar registros a una aplicación, así como cómo comenzar rápidamente, y obtendremos una introducción a algunas de las funciones avanzadas disponibles.

# El módulo de registro

El módulo de registro en Python es un módulo potente y listo para usar que está diseñado para satisfacer las necesidades tanto de los principiantes como de los equipos empresariales. Es utilizado por la mayoría de las bibliotecas de Python de terceros, por lo que se puede integrar los mensajes de registro con los de esas bibliotecas para producir un registro homogéneo para nuestra aplicación.

Agregar registro a un programa Python es tan fácil como esto:

```
import logging
```

Con el módulo de registro importado, podemos usar algo llamado "registrador" para registrar los mensajes que deseamos ver. De forma predeterminada, hay 5 niveles estándar que indican la gravedad de los eventos. Cada uno tiene un método correspondiente que se puede utilizar para registrar eventos en ese nivel de gravedad. Los niveles definidos, en orden de gravedad creciente, son los siguientes:

- DEPURAR
- INFO
- ADVERTENCIA
- ERROR
- CRÍTICO

El módulo de registro nos proporciona un registrador predeterminado que nos permite comenzar sin necesidad de realizar mucha configuración. Los métodos correspondientes para cada nivel se pueden llamar como se muestra en el siguiente ejemplo:

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
```

```
logging.critical('This is a critical message')
```

La salida del programa anterior se vería así:

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

La salida muestra el nivel de gravedad antes de cada mensaje junto con `root`, que es el nombre que el módulo de registro le da a su registrador predeterminado. (Los registradores se describen en detalle en secciones posteriores). Este formato, que muestra el nivel, el nombre y el mensaje separados por dos puntos ( `:`), es el formato de salida predeterminado que se puede configurar para incluir elementos como la marca de tiempo, el número de línea y otros detalles.

Observamos que los mensajes `debug()` e `info()` no se registraron. Esto se debe a que, de forma predeterminada, el módulo de registro registra los mensajes con un nivel de gravedad igual a `WARNING` o superior. Se puede cambiar eso configurando el módulo de registro para registrar eventos de todos los niveles si lo deseamos. También se puede definir nuestro propios niveles de gravedad cambiando las configuraciones, pero generalmente no se recomienda ya que puede causar confusión con los registros de algunas bibliotecas de terceros que podría estar utilizando.

## Configuraciones básicas

Puede utilizar el método para configurar el registro: `basicConfig(**kwargs)`

Algunos de los parámetros más utilizados `basicConfig()` son los siguientes:

- `level`: El registrador raíz se establecerá en el nivel de gravedad especificado.
- `filename`: Esto especifica el archivo.
- `filemode`: Si `filename`se da, el archivo se abre en este modo. El valor predeterminado es `a`, lo que significa agregar.
- `format`: Este es el formato del mensaje de registro.

Al usar el parámetro `level`, podemos establecer qué nivel de mensajes de registro deseamos registrar. Esto se puede hacer pasando una de las constantes disponibles en la clase, y esto permitiría que se registren todas las llamadas de registro en ese nivel o por encima de él. He aquí un ejemplo:

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('This will get logged')
```

```
DEBUG:root:This will get logged
```

Todos los eventos de nivel `DEBUG` o superior ahora se registrarán.

De manera similar, para iniciar sesión en un archivo en lugar de en la consola, se puede usar `filename` y `filemode`, y se puede decidir el formato del mensaje usando `format`. El siguiente ejemplo muestra el uso de los tres:

```
import logging

logging.basicConfig(filename='app.log', filemode='w', format='%(name)s
- %(levelname)s - %(message)s')
logging.warning('This will get logged to a file')
```

```
root - ERROR - This will get logged to a file
```

El mensaje se verá así, pero se escribirá en un archivo con nombre `app.log` en lugar de en la consola. El modo de archivo está configurado en `w` , lo que significa que el archivo de registro se abre en "modo de escritura" cada vez que se llama a `basicConfig()` cada ejecución del programa reescribirá el archivo. La configuración predeterminada para el modo de archivo es `a`, que se agrega.

Podemos personalizar aún más el registrador raíz utilizando más parámetros para `basicConfig()` que se pueden encontrar aquí .

Cabe señalar que las llamadas `basicConfig()` para configurar el registrador raíz solo funcionan si el registrador raíz no se ha configurado antes. **Básicamente, esta función solo se puede llamar una vez.**

`debug()`, `info()`, `warning()`, `error()`, Y `critical()`también llaman a `basicConfig()` sin argumentos automáticamente si no se ha llamado antes. Esto significa que después de la primera vez que se llama a una de las funciones anteriores, ya no se puede configurar el registrador raíz porque se habría llamado a la `basicConfig()` función internamente.

La configuración predeterminada en `basicConfig()` es configurar el registrador para que escriba en la consola en el siguiente formato:

```
ERROR:root:This is an error message
```

## Formatear la salida

Si bien podemos pasar cualquier variable que se pueda representar como una cadena de nuestro programa como un mensaje a sus registros, hay algunos elementos básicos que ya forman parte del `LogRecord` y se pueden agregar fácilmente al formato de salida. Si deseamos registrar el ID del proceso junto con el nivel y el mensaje, se puede hacer algo como esto:

```
import logging

logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')
logging.warning('This is a Warning')
```

```
18472-WARNING-This is a Warning
```

`Format` puede tomar una cadena con atributos `LogRecord` en cualquier disposición que deseemos. La lista completa de atributos disponibles se puede encontrar aquí .

Aquí hay otro ejemplo en el que agregamos la información de fecha y hora:

```
import logging
```

```
logging.basicConfig(format='%(asctime)s - %(message)s',
level=logging.INFO)
logging.info('Admin logged in')
```

```
2018-07-11 20:12:06,288 - Admin logged in
```

`%(asctime)s` agrega la hora de creación del `LogRecord`. El formato se puede cambiar usando el atributo `datefmt` que usa el mismo lenguaje de formato que las funciones de formato en el módulo de fecha y hora, como por ejemplo `time.strftime()`:

```
import logging

logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%d-
%b-%y %H:%M:%S')
logging.warning('Admin logged out')
```

```
12-Jul-18 20:53:19 - Admin logged out
```

Puedes encontrar la guía aquí .

## Registro de datos variables

En la mayoría de los casos, querremos incluir información dinámica de nuestra aplicación en los registros. Hemos visto que los métodos de registro toman una cadena como argumento y puede parecer natural formatear una cadena con datos variables en una línea separada y pasarla al método de registro. Pero esto en realidad se puede hacer directamente usando una cadena de formato para el mensaje y agregando los datos variables como argumentos. He aquí un ejemplo:

```
import logging

name = 'John'

logging.error('%s raised an error', name)
```

```
ERROR:root:John raised an error
```

Los argumentos pasados al método se incluirían como datos variables en el mensaje.

Si bien puede usar cualquier estilo de formato, el método .format() es una forma increíble de formatear cadenas, ya que pueden ayudar a mantener el formato corto y fácil de leer:

```
import logging

name = 'John'

logging.error('{} raised an error'.format(name))
```

```
ERROR:root:John raised an error
```

## Captura de rastros de pila

El módulo de registro también nos permite capturar los seguimientos de la pila completa en una aplicación. La información de excepción se puede capturar si el parámetro `exc_info` se pasa como `True`, y las funciones de registro se llaman así:

```
import logging

a = 5
b = 0

try:
  c = a / b
except Exception as e:
  logging.error("Exception occurred", exc_info=True)
```

```
ERROR:root:Exception occurred
Traceback (most recent call last):
  File "exceptions.py", line 6, in <module>
    c = a / b
ZeroDivisionError: division by zero
[Finished in 0.2s]
```

Si `exc_info` no se establece en `True`, la salida del programa anterior no nos dirá nada sobre la excepción, que, en un escenario del mundo real, podría no ser tan simple como `ZeroDivisionError`. Imaginemos intentar depurar un error en una base de código complicada con un registro que solo muestra esto:

```
ERROR:root:Exception occurred
```

Aquí hay un consejo rápido: si estamos iniciando sesión desde un controlador de excepciones, usamos el método `logging.exception()` que registra un mensaje con nivel `ERROR` y agrega información de excepción al mensaje. Para decirlo de manera más simple, llamar `logging.exception()` es como llamar `logging.error(exc_info=True)`. Pero dado que este método siempre descarga información de excepciones, solo debe llamarse desde un controlador de excepciones. Échale un vistazo a este ejemplo:

```
import logging

a = 5
b = 0
try:
  c = a / b
except Exception as e:
  logging.exception("Exception occurred")
```

```
ERROR:root:Exception occurred
Traceback (most recent call last):
  File "exceptions.py", line 6, in <module>
    c = a / b
ZeroDivisionError: division by zero
[Finished in 0.2s]
```

El uso `logging.exception()` mostraría un registro al nivel de `ERROR`. Si no lo deseamos, se puede llamar a cualquiera de los otros métodos de registro de `debug()` a `critical()` y pasar el parámetro `exc_info` como `True`.

## Clases y funciones

Hasta ahora, hemos visto el registrador predeterminado denominado `root`, que se utiliza por el módulo de registro cada vez que sus funciones son llamadas directamente como esto: `logging.debug()`. Se puede (y debemos) definir nuestro propio registrador creando un objeto de la clase `Logger` especialmente si la aplicación tiene varios módulos. Echemos un vistazo a algunas de las clases y funciones del módulo.

Las clases más utilizadas definidas en el módulo de registro son las siguientes:

- **`Logger`:** Esta es la clase cuyos objetos se usarán en el código de la aplicación directamente para llamar a las funciones.
- **`LogRecord`:** Los registradores crean automáticamente onjetos `LogRecord` que tienen toda la información relacionada con el evento que se registra, como el nombre del registrador, la función, el número de línea, el mensaje y más.
- **`Handler`:** Los controladores envían el `LogRecord` al destino de salida requerido, como la consola o un archivo. `Handler` es una base para subclases como `StreamHandler`, `FileHandler`, `SMTPHandler`, `HTTPHandler`, y más. Estas subclases envían las salidas de registro a los destinos correspondientes, como `sys.stdout` o un archivo de disco.
- **`Formatter`:** Aquí es donde se especifica el formato de la salida especificando un formato de cadena que enumera los atributos que la salida debe contener.

De estos, principalmente nos ocupamos de los objetos de la clase `Logger` que se instancian utilizando la función de nivel de módulo `logging.getLogger(name)`. Varias llamadas a `getLogger()` con el mismo `name` devolverán una referencia al mismo objeto `Logger` lo que nos ahorra pasar los objetos del registrador a cada parte donde sea necesario. He aquí un ejemplo:

```
import logging
```

```
logger = logging.getLogger('example_logger')
logger.warning('This is a warning')
```

```
This is a warning
```

Esto crea un registrador personalizado con nombre `example_logger`, pero a diferencia del registrador raíz, el nombre de un registrador personalizado no forma parte del formato de salida predeterminado y debe agregarse a la configuración. Configurarlo en un formato para mostrar el nombre del registrador daría un resultado como este:

```
WARNING:example_logger:This is a warning
```

Nuevamente, a diferencia del registrador raíz, no se puede configurar un registrador personalizado con `basicConfig()`. Tienes que configurarlo usando Manejadores y Formateadores:

## Uso de controladores

Los controladores entran en escena cuando deseamos configurar nuestros propios registradores y enviar los registros a varios lugares cuando se generan. Los controladores envían los mensajes de registro a destinos configurados como el flujo de salida estándar o un archivo o mediante HTTP o su correo electrónico a través de SMTP.

Un registrador que se cree puede tener más de un controlador, lo que significa que podemos configurarlo para que se guarde en un archivo de registro y también enviarlo por correo electrónico.

Al igual que los registradores, también se puede establecer el nivel de gravedad en los controladores. Esto es útil si deseamos configurar varios controladores para el mismo registrador, pero deseamos diferentes niveles de gravedad para cada uno de ellos. Por ejemplo, es posible que se desee que los registros con nivel `WARNING` y superiores se registren en la consola, pero todo con nivel `ERROR` y superior también debe guardarse en un archivo. Aquí hay un programa que hace eso:

```
# logging_example.py
```

```
import logging

# Create a custom logger
logger = logging.getLogger(__name__)

# Create handlers
c_handler = logging.StreamHandler()
f_handler = logging.FileHandler('file.log')
c_handler.setLevel(logging.WARNING)
f_handler.setLevel(logging.ERROR)

# Create formatters and add it to handlers
c_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
f_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')
c_handler.setFormatter(c_format)
f_handler.setFormatter(f_format)

# Add handlers to the logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

logger.warning('This is a warning')
logger.error('This is an error')
```

```
__main__ - WARNING - This is a warning
__main__ - ERROR - This is an error
```

Aquí, `logger.warning()` está creando un `LogRecord` que contiene toda la información del evento y pasándola a todos los Handlers que tiene: `c_handler` y `f_handler`.

`c_handler` es un `StreamHandler` nivel con `WARNING` y toma la información del `LogRecord` para generar una salida en el formato especificado y la imprime en la consola. `f_handler` es un `FileHandler` nivel con `ERROR` y lo ignora `LogRecord` como su nivel `WARNING`.

Cuando `logger.error()`se llama, `c_handler` se comporta exactamente como antes y `f_handler` obtiene un valor `LogRecord` al nivel de `ERROR`, por lo

que procede a generar una salida como `c_handler`, pero en lugar de imprimirla en la consola, la escribe en el archivo especificado en este formato:

```
2018-08-03 16:12:21,723 - __main__ - ERROR – This is an error
```

El nombre del registrador correspondiente a la variable `__name__` se registra como `__main__`, que es el nombre que Python asigna al módulo donde comienza la ejecución. Si este archivo es importado por algún otro módulo, entonces la `__name__`variable correspondería a su nombre *logging_example* . Así es como se vería:

```
# run.py

import logging_example
```

```
logging_example - WARNING - This is a warning
logging_example - ERROR - This is an error
```

## Otros métodos de configuración

Podemos configurar el registro como se muestra arriba usando las funciones de módulo y clase o creando un archivo de configuración o un diccionario y cargándolo, usando `fileConfig()` o `dictConfig()` respectivamente. Estos son útiles en caso de que se desee cambiar su configuración de registro en una aplicación en ejecución.

Aquí hay un ejemplo de configuración de archivo:

```
[loggers]
keys=root,sampleLogger

[handlers]
keys=consoleHandler

[formatters]
keys=sampleFormatter

[logger_root]
level=DEBUG
```

```
handlers=consoleHandler

[logger_sampleLogger]
level=DEBUG
handlers=consoleHandler
qualname=sampleLogger
propagate=0

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

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

En el archivo anterior, hay dos registradores, un controlador y un formateador. Una vez definidos sus nombres, se configuran agregando las palabras registrador, manejador y formateador antes de sus nombres separados por un guión bajo.

Para cargar este archivo de configuración, debe usar `fileConfig()`:

```
import logging
import logging.config

logging.config.fileConfig(fname='file.conf',
disable_existing_loggers=False)

# Get the logger specified in the file
logger = logging.getLogger(__name__)

logger.debug('This is a debug message')
```

```
2018-07-13 13:57:45,467 - __main__ - DEBUG - This is a debug message
```

La ruta del archivo de configuración se pasa como parámetro al método `fileConfig()` y el parámetro `disable_existing_loggers` se usa para mantener o deshabilitar los registradores que están presentes cuando se llama a la función. Por defecto, `True` si no se menciona.

Esta es la misma configuración en un formato YAML para el enfoque de 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:
  sampleLogger:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

Aquí hay un ejemplo que muestra cómo cargar la configuración desde un archivo `yaml`:

```python
import logging
import logging.config
import yaml

with open('config.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)

logger = logging.getLogger(__name__)
```

```
logger.debug('This is a debug message')
```

```
2018-07-13 14:05:03,766 - __main__ - DEBUG - This is a debug message
```

## Mantenga la calma y lea los registros

El módulo de registro se considera muy flexible. Su diseño es muy práctico y debería adaptarse a nuestro caso de uso desde el primer momento. Se pueden agregar registros básicos a un proyecto pequeño, o se puede ir tan lejos como para crear sus propios niveles de registros personalizados, clases de controladores y más si estamos trabajando en un proyecto grande.

Si no se ha estado utilizando el inicio de sesión en nuestras aplicaciones, ahora es un buen momento para comenzar. Cuando se hace correctamente, el registro seguramente eliminará mucha fricción de nuestro proceso de desarrollo y nos ayudará a encontrar oportunidades para llevar nuestra aplicación al siguiente nivel.