

Loguear Eventos I



Introducción

Bienvenidos al módulo 3 - Loguear Eventos I. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conocimientos teóricos y prácticos el logging de eventos utilizando Python.

Una vez finalizado este módulo serás capaz de:

- Interactuar con el módulo Logging y realizar un setup inicial
- Realizar la configuración del logging
- Realizar una captura de Stack Traces y manejar excepciones



Tema 1. Librería Logging

Objetivos

Conocer acerca del módulo logging, utilizar el logger que ofrece por defecto Python, realizar configuraciones básicas y ajustar el formato de salida.

Al finalizar este tema serás capaz de:

- Utilizar el módulo logging y utilizar el logger por defecto.
- Realizar configuraciones básicas del logger por defecto y ajustar su formato.
- Realizar una captura del stack trace y utilizar el logger por defecto.

Módulo logging

El módulo logging se encuentra listo para utilizar y está desarrollado para cumplir los requerimientos de desarrolladores principiantes, como así también de empresas. Este módulo es utilizado por la mayoría de librerías de terceros, por lo que nos permite integrar nuestros propios logs con los de esas librerías y tener un logging homogéneo en nuestra aplicación.

Cuando importamos el módulo logging, podemos utilizar algo llamado “logger” para registrar los mensajes que queramos ver. Por default hay 5 niveles estándar que indican la severidad del evento. Cada uno tiene su correspondiente método que puede ser utilizado para registrar eventos de acuerdo a la severidad.

Los niveles, ordenados por severidad, son los siguientes:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

El módulo logging nos provee un logger por defecto que nos permite comenzar realizando pocas configuraciones. Los métodos correspondientes a cada nivel se pueden invocar de la siguiente manera.

```
import logging

logging.debug('Este es un mensaje de debug')
logging.info('Este es un mensaje de información')
logging.warning('Este es un mensaje de alerta')
logging.error('Este es un mensaje de error')
logging.critical('Este es un mensaje crítico')
```

```
WARNING:root:Este es un mensaje de alerta
ERROR:root:Este es un mensaje de error
CRITICAL:root:Este es un mensaje crítico
```

La salida del programa nos muestra el nivel de severidad seguido de la palabra root, que es el nombre del logger que le da por default al módulo logging. Este formato que nos muestra el nivel, nombre y mensaje separado por el carácter : es el formato por defecto que puede ser configurado para incluir otra información como el timestamp o marca de tiempo.

En la salida del código anterior podemos notar que tanto los mensajes debug e info no aparecen en el output. Esto se debe a que por defecto el módulo logging registra los mensajes con nivel de severidad WARNING o superior. Esta configuración se puede editar también para registrar los mensajes de todos los niveles. Asimismo, también podemos definir nuestros propios niveles de severidad, aunque generalmente no se recomienda ya que puede generar confusión con registros de aplicaciones de terceros que podemos estar utilizando en nuestro programa.

Configuraciones básicas

Para realizar la configuración del módulo logging podemos utilizar el siguiente método **basicConfig(**kwargs)**. Algunos de los parámetros más utilizados de este método son los siguientes:

- **level:** Especifica el nivel de severidad raíz.
- **filename:** Especifica el nombre del archivo.

- **filemode:** Si se especifica un filename, ese archivo se abre. Por default, este método recibe el parámetro *a*, que significa append.
- **format:** Especifica el formato del mensaje de log.

Continuando el ejemplo anterior, si ahora especificamos el nivel de severidad como DEBUG, veremos que nos habilita el registro de logs a partir de ese nivel.

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug('Este es un mensaje de debug')
logging.info('Este es un mensaje de información')
logging.warning('Este es un mensaje de alerta')
logging.error('Este es un mensaje de error')
logging.critical('Este es un mensaje crítico')
```

DEBUG:root:Este es un mensaje de debug
 INFO:root:Este es un mensaje de información
 WARNING:root:Este es un mensaje de alerta
 ERROR:root:Este es un mensaje de error
 CRITICAL:root:Este es un mensaje crítico

En el caso de que deseemos registrar los logs en un archivo, en lugar de la consola, y además utilizando un formato del mensaje que sea diferente, podemos hacerlo de la siguiente manera.

```
import logging

logging.basicConfig(
    level = logging.DEBUG,
    filename = 'app.log',
    filemode = 'w',
    format = '%(name)s - %(levelname)s - %(message)s'
)

logging.debug('Este es un mensaje de debug')
logging.info('Este es un mensaje de información')
logging.warning('Este es un mensaje de alerta')
logging.error('Este es un mensaje de error')
logging.critical('Este es un mensaje crítico')
```

Así podemos observar que en vez de obtener los mensajes como output, se nos ha generado un archivo `app.log` con los logs correspondientes.

```

app.log: Bloc de notas
Archivo  Editar  Ver

root - DEBUG - Este es un mensaje de debug
root - INFO - Este es un mensaje de información
root - WARNING - Este es un mensaje de alerta
root - ERROR - Este es un mensaje de error
root - CRITICAL - Este es un mensaje crítico

```

En el siguiente [link](#) se puede consultar la documentación oficial con todos los parámetros que soporta el método `basicConfig`.

Es importante destacar que una vez que llamamos al método **`basicConfig()`** para configurar el logger raíz, solo funcionará en caso de que no haya sido configurado anteriormente. Esto significa que podemos llamar a este método solo una vez, y en caso de necesitar editar la configuración, deberemos reiniciar el kernel.

Formato de salida

Además de definir el texto que se enviará según la severidad, existen algunos elementos básicos que ya vienen incorporados dentro del módulo `logging` y que pueden incorporarse para mejorar la comprensión del log.

```

import logging

logging.basicConfig(
    level = logging.DEBUG,
    datefmt = '%d-%b-%y %H:%M:%S',
    format = '%(asctime)s - %(levelname)s - %(levelname)s - %(message)s'
)

logging.debug('Este es un mensaje de debug')
logging.info('Este es un mensaje de información')
logging.warning('Este es un mensaje de alerta')
logging.error('Este es un mensaje de error')
logging.critical('Este es un mensaje crítico')

07-Jun-22 13:32:45 - DEBUG -10 - Este es un mensaje de debug
07-Jun-22 13:32:45 - INFO -20 - Este es un mensaje de información
07-Jun-22 13:32:45 - WARNING -30 - Este es un mensaje de alerta
07-Jun-22 13:32:45 - ERROR -40 - Este es un mensaje de error
07-Jun-22 13:32:45 - CRITICAL -50 - Este es un mensaje crítico

```

En el siguiente [link](#) podemos consultar todos los modificadores.

Attribute name	Format	Description
args	You shouldn't need to format this yourself.	The tuple of arguments merged into <code>msg</code> to produce <code>message</code> , or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
asctime	<code>%(asctime)s</code>	Human-readable time when the <code>LogRecord</code> was created. By default this is of the form '2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
created	<code>%(created)f</code>	Time when the <code>LogRecord</code> was created (as returned by <code>time.time()</code>).
exc_info	You shouldn't need to format this yourself.	Exception tuple (à la <code>sys.exc_info</code>) or, if no exception has occurred, <code>None</code> .
filename	<code>%(filename)s</code>	Filename portion of <code>pathname</code> .
funcName	<code>%(funcName)s</code>	Name of function containing the logging call.
levelname	<code>%(levelname)s</code>	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').
levelno	<code>%(levelno)s</code>	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
lineno	<code>%(lineno)d</code>	Source line number where the logging call was issued (if available).
message	<code>%(message)s</code>	The logged message, computed as <code>msg % args</code> . This is set when <code>Formatter.format()</code> is invoked.
module	<code>%(module)s</code>	Module (name portion of <code>filename</code>).
msecs	<code>%(msecs)d</code>	Millisecond portion of the time when the <code>LogRecord</code> was created.
msg	You shouldn't need to format this yourself.	The format string passed in the original logging call. Merged with <code>args</code> to produce <code>message</code> , or an arbitrary object (see Using arbitrary objects as messages).
name	<code>%(name)s</code>	Name of the logger used to log the call.
pathname	<code>%(pathname)s</code>	Full pathname of the source file where the logging call was issued (if available).
process	<code>%(process)d</code>	Process ID (if available).
processName	<code>%(processName)s</code>	Process name (if available).
relativeCreated	<code>%(relativeCreated)d</code>	Time in milliseconds when the <code>LogRecord</code> was created, relative to the time the logging module was loaded.
stack_info	You shouldn't need to format this yourself.	Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
thread	<code>%(thread)d</code>	Thread ID (if available).
threadName	<code>%(threadName)s</code>	Thread name (if available).

Capturar los Stack Traces

El módulo logging también nos permite capturar las trazas completas en una aplicación. Además podemos agregar información sobre la excepción, agregando el parámetro **exc_info = True**.

```
import logging

logging.basicConfig(
    level = logging.DEBUG,
    datefmt = '%d-%b-%y %H:%M:%S',
    format = '%(asctime)s - %(levelname)s - %(levelname)s - %(message)s'
)

a = 5
b = 0
try:
    a/b
except Exception as e:
    logging.error('Encontramos una excepción', exc_info = True)
```

07-Jun-22 13:58:25 - ERROR -40 - Encontramos una excepción
 Traceback (most recent call last):
 File "<ipython-input-2-b722278ba4b9>", line 10, in <module>
 a/b
 ZeroDivisionError: division by zero

En caso de que el parámetro exc_info = False, el output del programa no nos dirá nada acerca de la excepción.

Cuando trabajamos con logging en el manejo de excepciones podemos utilizar el método logging.exception () que genera un log con un mensaje con nivel de ERROR y agrega la información de la excepción al mensaje.

```
logging.basicConfig(
    level = logging.DEBUG,
    datefmt = '%d-%b-%y %H:%M:%S',
    format = '%(asctime)s - %(levelname)s - %(levelname)s - %(message)s'
)

a = 5
b = 0
try:
    a/b
except Exception as e:
    logging.exception('Encontramos una excepción')
```

07-Jun-22 15:17:40 - ERROR -40 - Encontramos una excepción
 Traceback (most recent call last):
 File "<ipython-input-8-b48900f67a7c>", line 10, in <module>
 a/b
 ZeroDivisionError: division by zero



Tema 2. Creando nuestro Logging

Objetivo

Comprender cómo crear un objeto logger y configurarlo de acuerdo a nuestras necesidades para ser importado en nuestros proyectos.

Al finalizar este tema serás capaz de:

- Crear un logger personalizado utilizando handlers y formatos.

Funciones y Clases

Hasta ahora estuvimos trabajando con el logger por defecto llamado root, el cual es utilizado por el módulo logging y que nos permite utilizar sus métodos como debug(), info(), etc.

No obstante podemos definir nuestro propio logger creando un objeto utilizando la clase Logger. Esto último resulta muy útil cuando la aplicación tiene múltiples módulos.

Las clases más utilizadas 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 Loggers crean automáticamente este tipo de objeto que tiene toda la información relacionada con el evento que se registra, como el nombre del registrador, la función, el número de línea, el mensaje, etc.
- **Handler:** El handler envía los objetos LogRecord al destino de salida requerido, como por ejemplo 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:** Es donde especificamos el formato de salida, especificando un formato de cadena que enumera los atributos que debe contener la salida.

De estos, principalmente nos ocupamos de los objetos de la clase `Logger`, que se instancian mediante **`logging.getLogger(nombre)`**. Múltiples llamadas a `getLogger()` con el mismo nombre devolverán una referencia al mismo objeto `Logger`, lo que nos evita pasar los objetos de registro a cada parte donde se necesita.

```
import logging

logger = logging.getLogger('ejemplo')
logger.warning('Esta es una alerta')

Esta es una alerta
```

Este código nos genera un logger personalizado llamado `ejemplo`, pero a diferencia del logger `root`, no nos muestra en su salida, el tipo de alerta y nombre del logger, como vimos anteriormente.

```
WARNING:root:Este es un mensaje de alerta
```

Anteriormente, cuando trabajamos con el logger `root`, realizamos esta personalización utilizando el método `basicConfig()`. En el caso de un logger personalizado, realizaremos esa configuración utilizando `Handler` y `Formatter`.

Utilización de Handlers y formatos

Los handlers entran en acción cuando queremos configurar nuestros propios loggers y enviar mensajes a diferentes lugares apenas se generen. Así podemos enviarlos a destinos preconfigurados como el output de nuestro programa, a un archivo o a un correo electrónico.

Nuestro logger puede tener más de un handler, lo cual significa que podemos configurarlo para que guarde un log en un archivo y también para que envíe ese log a un correo electrónico.

Por otra parte, podemos configurar el nivel de severidad de los handlers. Esto es muy útil si queremos ajustar múltiples handlers para un mismo logger, pero queremos diferentes niveles de severidad.

Un caso de ejemplo sería visualizar los logs de severidad WARNING y superiores en el output del programa y además los de severidad ERROR guardarlos en un archivo aparte.

Logging Handlers

En la [documentación](#) oficial podemos consultar sobre las clases Handler y sus métodos.

- Logging handlers
- StreamHandler
 - FileHandler
 - NullHandler
 - WatchedFileHandler
 - BaseRotatingHandler
 - RotatingFileHandler
 - TimedRotatingFileHandler
 - SocketHandler
 - DatagramHandler
 - SysLogHandler
 - NTEventLogHandler
 - SMTPHandler
 - MemoryHandler
 - HTTPHandler
 - QueueHandler
 - QueueListener

Las principales clases a tener en cuenta son las siguientes.

Handler	Descripción
StreamHandler	Envía la salida del logging utilizando la consola de python
FileHandler	Envía la salida del logging a un archivo
RotatingFileHandler	Permite la rotación de archivos,

	creando nuevos archivos cada vez que se cumplan determinadas condiciones.
TimedRotatingFileHandler	Admite la rotación de archivos de registro de disco en ciertos intervalos de tiempo.

Ejercicio práctico

Crear un logging personalizado con las siguientes características:

Handler 1:

- Nivel de severidad: Warning
- Formato:
 - modificadores: name - levelname - message
- formato de salida: imprimir por consola

● Handler 2:

- Nivel de severidad: Error
- Formato:
 - modificadores: asctime - levelname - message
- Formato de salida: imprimir por consola y guardar en un archivo txt.

Resolución

```
#customLogger.py
import logging

# Creamos nuestro logger personalizado
logger = logging.getLogger(__name__)

#Ajustamos el nivel de severidad
logger.setLevel(logging.DEBUG)

# Creamos los handlers
c_handler = logging.StreamHandler()
f_handler = logging.FileHandler('app.log')

#Ajustamos el nivel de severidad de los handlers
c_handler.setLevel(logging.WARNING)
f_handler.setLevel(logging.ERROR)

# Ajustamos el formato y lo agregamos a cada handler
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)

# Agregamos los handlers al logger
logger.addHandler(c_handler)
logger.addHandler(f_handler)

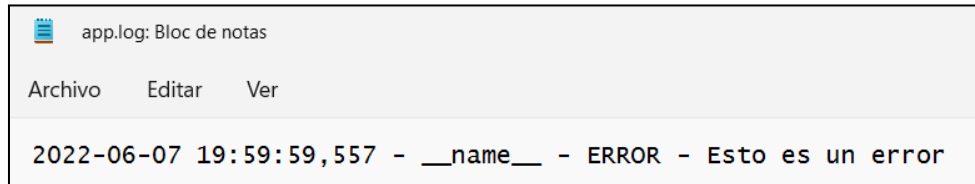
#Probamos el logger personalizado
logger.warning('Esto es un warning')
logger.error('Esto es un error')

__main__ - WARNING - Esto es un warning
__main__ - ERROR - Esto es un error
```

Acá podemos ver como **logger.warning()** está creando un log que contiene toda la información del evento y la está pasando a los dos handlers que tiene: **c_handler** y **f_handler**.

El handler **c_handler** es del tipo `StreamHandler` con un nivel de severidad `WARNING` que toma la información del log para generar un output en el formato especificado en la variable **c_format** y lo muestra en la consola. Por otro lado el handler **f_handler** es del tipo `FileHandler` con un nivel de severidad `ERROR` que ignora si el tipo de log es de severidad `WARNING`.

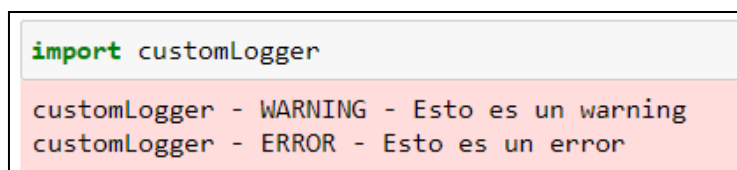
Cuando ejecutamos `logger.error()`, el **f_handler** toma la información del log de severidad **ERROR** y la escribe en un archivo, teniendo en cuenta el formato establecido anteriormente.



```
app.log: Bloc de notas
Archivo  Editar  Ver
2022-06-07 19:59:59,557 - __name__ - ERROR - Esto es un error
```

Finalmente vemos que el nombre del logger que corresponde a la variable `__name__` se registra como `__main__`, el cual corresponde al nombre que asigna al módulo cuando comienza la ejecución.

En el caso de que tengamos nuestro logger guardado en un archivo separado (ej. `customLogger.py`) y luego lo importamos en nuestro programa principal, vamos a ver que reemplazará esa variable `__name__` por el nombre del archivo.



```
import customLogger
customLogger - WARNING - Esto es un warning
customLogger - ERROR - Esto es un error
```

Cierre

A lo largo de este módulo revisamos el módulo logging que viene por defecto en Python.

Aprendimos cómo configurarlo en su versión por defecto y a ajustar el formato de salida.

Posteriormente aprendimos a crear nuestro propio logging aplicando configuraciones de formatos y handlers según nuestras necesidades.

Referencias

Instalación de Logging para Python | Documentación de Python - 3.10.7. Recuperado de:
<https://docs.python.org/es/3/library/logging.html#logging.info>

Logging in Python | Real Python. Recuperado de:
<https://realpython.com/python-logging/#using-handlers>

tiiiiiit by 