Logging

Antonio Espín Herranz

Contenidos

- Módulo logging: Objetos y métodos
- Niveles de logging
- Handlers
- Formateadores
- Filtros
- LogRecord

Módulo logging

- El módulo logging de la librería estándar de python nos permite registrar eventos dentro del sotfware.

- La ventaja de utilizar registros es que se pueden desactivar, volcar a consola, a un fichero.

- Se pueden establecer niveles de gravedad para informar sobre el error o simplemente utilizar como mensajes de depuración.

Métodos

- Los principales métodos son:
  - debug(),
  - info(),
  - warning(),
  - error() y
  - critical().

- Que indican el nivel de gravedad del mensaje.

Niveles de Logging

- Hay 5 niveles de logging:
  - **DEBUG**
    - Información detallada, típicamente cuando se diagnostican problemas.
  - **INFO**
    - Para confirmar que las cosas funcionan como se espera.
  - **WARNING**
    - Algo inesperado sucedió aunque el programa continúa ejecutándose como se esperaba.
  - **ERROR**
    - Ha habido un problema mas grave y el programa no ha podido realizar alguna opción.
  - **CRITICAL**
    - El error es grave y el programa no puede continuar ejecutándose.

Niveles de Logging

- Cada nivel de logging está relacionado con un método del módulo:

- import logging
- logging.info(‘mensaje de info’)
- logging.debug(‘mensaje de debug’)

- Solo se muestran a partir del nivel warning hacia arriba.
- El nivel por defecto es warning.

Volcado a Ficheros

- Los mensajes de Log a parte de mostrarse por consola se puede volcar a un fichero.

- Al establecer la configuración se indica un fichero y el nivel logging.

```
import logging
logging.basicConfig(filename='example.log',level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

Volcado a Ficheros

- El comportamiento por defecto es que el fichero se abre en modo **append**, va añadiendo los mensajes de log al final.

- Este comportamiento se puede cambiar añadiendo el parámetro **filemode = ‘w’** y en cada nueva ejecución el fichero empieza de nuevo.

Formatear mensajes

- import logging
- logging.warning(‘El contenido es: **%s**’, ‘prueba’)

- Se puede mostrar el nivel del mensaje, en cada uno de los mensajes (si no indicamos fichero se vuelca a consola).
  - **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')

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

Visualizar fecha / hora

- Viene con un formato preestablecido y se puede cambiar:

- En la cadena formato se indica ‘% (asctime)’
  - **import logging**
  - logging.basicConfig(**format**='*%(**asctime**)s %(message)s*')
  - logging.warning('is when this event was logged.')

- Por defecto, se muestra con el siguiente formato:

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

Visualizar fecha / hora

- El parámetro **datefmt** en **basicConfig** permite cambiar el formato de la fecha y hora:
  - **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.')

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

Handlers

- Los handler son los objetos responsables de **enviar los mensajes al destino** especificado en el manejador.
- Puede ser a consola, a un fichero, por correo … hay un manejador para cada destino.
- Los manejadores se registran en el **logger** mediante un método **addHandler**()
- Principalmente, se utilizan **StreamHandler** y **FileHandler**.

Tipos de Handler

- **StreamHandler**:
  - Envía las secuencias de registro a **sys.stdout**, **sys.stderr** o cualquier objeto que admita el método **write**() y **flush**()
  - Constructor:
    - logging.StreamHandler(stream=None)
      - Devuelve una nueva instancia. Se puede indicar un flujo, y si es así lo utilizará para volcar la salida.
    - emit(record)
      - Si se ha indicado un formateador, se utilizará este método para formatear la salida.
    - flush()
      - Para vaciar el buffer, y volcar el registro al flujo
    - setStream(stream)
      - Para configurar el flujo que utilizará el handler.

Tipos de Handler

- **FileHandler**:
  - Esta clase hereda de StreamHandler y envía el resultado a un archivo.
    - Normalmente se utiliza esta clase.
  - logging.FileHandler(nombre_archivo, modo='a', encoding=None, delay=False)
    - Constructor devuelve una nueva instancia.
  - close()
    - Cierra el fichero.
  - emit(record)
    - Para formatear la salida.

Handler

- Dentro del **Logger** los métodos de gestión son:
  - setLevel()
  - setFormatter()
  - addFilter(), removeFilter():
    - Para configurar filtros.

Formateadores

- Estos objetos son los encargados de formatear la salida de los mensajes.

- El constructor recibe 3 parámetros:
  - logging.Formatter.**__init__**(fmt=None, datefmt=None, style=%)
  - Si no se indica una cadena de fecha, se indica:
    - %Y-%m-%d %H:%M:%S
  - El parámetro **style**, utiliza %(<dictionary_keys>)s para indicar el estilo de la sustitución.
  - Por ejemplo:
    - '%(asctime)s - %(levelname)s - %(message)s'
    - La fecha, el nivel y el mensaje.

Configurando el registro

- Hay 3 formas de configurar el registro:
  - Se pueden crear registradores (logger), handlers y formatter.
  - Crear un fichero de configuración y la función fileConfig()
  - Creando un diccionario con la información de configuración y pasándolo a la función dictConfig().

Ejemplo 1ª forma

```
import logging

# crear logger
logger = logging.getLogger('simple_example')  # simple_example: es el nombre → %(name)s
logger.setLevel(logging.DEBUG)

# Crear un manejador de consola y establecer el nivel en DEBUG
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# Crear formateador:
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# Se añade el formateador al handler:
ch.setFormatter(formatter)

# Añadir el handler
logger.addHandler(ch)

# pruebas
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Salida

- 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

Ejemplo 2ª forma

- Con un fichero de configuración:
  - Se simplifica el código de python.
  - Al separar la configuración es más cómodo hacer los cambios:

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

# create logger
logger = logging.getLogger('simpleExample')
# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

El fichero de configuración

[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
datefmt=

Salida

- 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

Utilizando logging en múltiples módulos (1 de 2)

```
import logging
import auxiliary_module

# Crear un logger: 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)

# Crear un manejador de file:
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)

# Crear otro manejador para la consola:
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)

# Crear un formateador y se añade a los handlers:
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
    %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
```

Utilizando logging en múltiples módulos (1 de 2)

```
# Añadir los handler al logger:
logger.addHandler(fh)
logger.addHandler(ch)

# Emitir mensajes y llamar al otro módulo:
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()')
```

El otro módulo

```
import logging

# Crear el 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"')
```

Salida

- 2018-12-03 19:08:37,390 - **spam_application** - INFO - creating an instance of auxiliary_module.Auxiliary
- 2018-12-03 19:08:37,390 - **spam_application.auxiliary.Auxiliary** - INFO - creating an instance of Auxiliary
- 2018-12-03 19:08:37,390 - **spam_application** - INFO - created an instance of auxiliary_module.Auxiliary
- 2018-12-03 19:08:37,390 - **spam_application** - INFO - calling auxiliary_module.Auxiliary.do_something
- 2018-12-03 19:08:37,390 - **spam_application.auxiliary.Auxiliary** - INFO - doing something
- 2018-12-03 19:08:37,391 - **spam_application.auxiliary.Auxiliary** - INFO - done doing something
- 2018-12-03 19:08:37,391 - **spam_application** - INFO - finished auxiliary_module.Auxiliary.do_something
- 2018-12-03 19:08:37,391 - **spam_application** - INFO - calling auxiliary_module.some_function()
- 2018-12-03 19:08:37,391 - **spam_application.auxiliary** - INFO - received a call to "some_function"
- 2018-12-03 19:08:37,391 - **spam_application** - INFO - done with auxiliary_module.some_function()

Filtros

- Los **filtros** se pueden **utilizar** en los **handlers** y **loggers** para un filtrado más sofisticado que el que proporcionan los niveles (info, debug, etc.)

- Los filtros actúan sobre los registros que se van a emitir, un registro deberá cumplir todos los filtros para que este se emita.

Filtros

- El módulo logging dispone de la clase Filter y el método filter(record) que devuelve 1 / 0, o True / False para indicar si el registro se emite o no.

- Esta clase se puede utilizar como base para construir los filtros.

- Una vez creada la clase se añade al Logger o a un Handler.

Filtros

- La clase hereda de **logging.Filter**:

```
class NoParsingFilter(logging.Filter):
    def filter(self, record):
        # El método devuelve True / False
        return not record.getMessage().startswith('parsing')

logger.addFilter(NoParsingFilter())
```

Enlaces

- Documentación Python:
  - https://docs.python.org/3/library/logging.handlers.html?highlight=web%20services#module-logging.handlers
- Rotación de logs (para que no crezcan infinitamente):
  - http://codigo-python.blogspot.com/2017/12/logs-con-rotado-en-python-como-modulo.html
- Preguntas sobre filtros:
  - https://stackoverflow.com/questions/879732/logging-with-filters