**巨蟒伐木手册**

官方手册摘录编辑

Logging is a means of tracking events that happen when some software runs. The software’s developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

a means of 一种手段

日志是软件运行时追踪事件的一种方法。开发者在其代码中添加logging调用以指示确定发生的事件。事件由解释消息描述，解释消息中可包含变量数据。

**简单示例**

最贱例：

**import** logging

logging.warning('Watch out!') # will print a message to the console

logging.info('I told you so') # will not print anything

日志写入文件

**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')

This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to DEBUG, all of the messages were printed.

The call to basicConfig() should come before any calls to debug(), info() etc. 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.

As it is 实际上，事实上

Intended as 作为

One-off 一次性

实际上作为一次性的简单配置功能，

If you run the above script several times, the messages from successive runs are appended to the file example.log. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the filemode argument, by changing the call in the above example to:

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

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

**Logging variable data**

**Using particular formatting styles throughout your application**

In Python 3.2, the Formatter gained a style keyword parameter which, while defaulting to % for backward compatibility, allowed the specification of { or $ to support the formatting approaches supported by str.format() and string.Template. Note that this governs the formatting of logging messages for final output to logs, and is completely orthogonal to how an individual logging message is constructed.

Logging calls (debug(), info() etc.) only take positional parameters for the actual logging message itself, with keyword parameters used only for determining options for how to handle the logging call (e.g. the exc\_info keyword parameter to indicate that traceback information should be logged, or the extra keyword parameter to indicate additional contextual information to be added to the log). So you cannot directly make logging calls using str.format() or string.Template syntax, because internally the logging package uses %-formatting to merge the format string and the variable arguments. There would no changing this while preserving backward compatibility, since all logging calls which are out there in existing code will be using %-format strings.

There have been suggestions to associate format styles with specific loggers, but that approach also runs into backward compatibility problems because any existing code could be using a given logger name and using %-formatting.

For logging to work interoperably between any third-party libraries and your code, decisions about formatting need to be made at the level of the individual logging call. This opens up a couple of ways in which alternative formatting styles can be accommodated.

**Changing the format of displayed messages**

For a full set of things that can appear in format strings, you can refer to the documentation for LogRecord attributes

**Displaying the date/time in messages**

**Advanced Logging Tutorial**

Logger 书记员

Log event information is passed between loggers, handlers, filters and formatters in a LogRecord instance.

Logging is performed by calling methods on instances of the Logger class (hereafter called loggers). Each instance has a name, and they are conceptually arranged in a namespace hierarchy using dots (periods) as separators. For example, a logger named ‘scan’ is the parent of loggers ‘scan.text’, ‘scan.html’ and ‘scan.pdf’. Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

Logging是由调用Logger类实例的方法来实现的。每个实例都有名称，其遵从由点分隔的命名空间结构概念。Logger名称可随意，其指示日志消息在应用程序中的哪个区间内产生。

A good convention to use when naming loggers is to use a module-level logger, in each module which uses logging, named as follows:

当命名logger 在模块中使用时的一个好习惯，如下所示：

logger = logging.getLogger(\_\_name\_\_)

This means that logger names track the package/module hierarchy, and it’s intuitively obvious where events are logged just from the logger name.

这意味着书记员命名遵循包结构，并且该结构关系仅由日志消息的书记员名称即明显可见。

The root of the hierarchy of loggers is called the root logger. That’s the logger used by the functions debug(), info(), warning(), error() and critical(), which just call the same-named method of the root logger. The functions and the methods have the same signatures. The root logger’s name is printed as ‘root’ in the logged output.

#

logging.**getLogger**(name=None)

Return a logger with the specified name or, if name is None, return a logger which is the root logger of the hierarchy. If specified, the name is typically a dot-separated hierarchical name like ‘a’, ‘a.b’ or ‘a.b.c.d’. Choice of these names is entirely up to the developer who is using logging.

All calls to this function with a given name return the same logger instance. This means that logger instances never need to be passed between different parts of an application.

It is, of course, possible to log messages to different destinations. Support is included in the package for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, queues, or OS-specific logging mechanisms such as syslog or the Windows NT event log. Destinations are served by handler classes. 特殊需求自行解决。

By default, no destination is set for any logging messages. You can specify a destination (such as console or file) by using basicConfig() as in the tutorial examples. If you call the functions debug(), info(), warning(), error() and critical(), they will check to see if no destination is set; and if one is not set, they will set a destination of the console (sys.stderr) and a default format for the displayed message before delegating to the root logger to do the actual message output.

The default format set by basicConfig() for messages is:

severity:logger name:message

You can change this by passing a format string to basicConfig() with the format keyword argument. For all options regarding how a format string is constructed, see Formatter Objects.

**Loggers 书记员**

Logger objects have a threefold job. First, they expose several methods to application code so that applications can log messages at runtime. Second, logger objects determine which log messages to act upon based upon severity (the default filtering facility) or filter objects. Third, logger objects pass along relevant log messages to all interested log handlers.

书记员有三项工作。首先，告知应用程序自己能干点啥。其次，根据重要级别或filter objects确定记录哪些消息。其三，把消息给需要的信差。

The most widely used methods on logger objects fall into two categories: configuration and message sending.

书记员的诸多方法中应用最广泛的有两类：配置与消息发送。

These are the most common configuration methods:

Logger.setLevel() specifies the lowest-severity log message a logger will handle, where debug is the lowest built-in severity level and critical is the highest built-in severity. For example, if the severity level is INFO, the logger will handle only INFO, WARNING, ERROR, and CRITICAL messages and will ignore DEBUG messages.

Logger.addHandler() and Logger.removeHandler() add and remove handler objects from the logger object. ~~Handlers are covered in more detail in Handlers.~~

Logger.addFilter() and Logger.removeFilter() add and remove filter objects from the logger object. ~~Filters are covered in more detail in Filter Objects.~~

You don’t need to always call these methods on every logger you create. See the last two paragraphs in this section.

With the logger object configured, the following methods create log messages:

Logger.debug(), Logger.info(), Logger.warning(), Logger.error(), and Logger.critical() all create log records with a message and a level that corresponds to their respective method names. The message is actually a format string, which may contain the standard string substitution syntax of %s, %d, %f, and so on. The rest of their arguments is a list of objects that correspond with the substitution fields in the message. With regard to \*\*kwargs, the logging methods care only about a keyword of exc\_info and use it to determine whether to log exception information.

Logger.exception() creates a log message similar to Logger.error(). The difference is that Logger.exception() dumps a stack trace along with it. Call this method only from an exception handler.

Logger.log() takes a log level as an explicit argument. This is a little more verbose for logging messages than using the log level convenience methods listed above, but this is how to log at custom log levels.

getLogger() returns a reference to a logger instance with the specified name if it is provided, or root if not. The names are period-separated hierarchical structures. Multiple calls to getLogger() with the same name will return a reference to the same logger object. Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of foo, loggers with names of foo.bar, foo.bar.baz, and foo.bam are all descendants of foo.

Loggers have a concept of effective level. If a level is not explicitly set on a logger, the level of its parent is used instead as its effective level. If the parent has no explicit level set, its parent is examined, and so on - all ancestors are searched until an explicitly set level is found. The root logger always has an explicit level set (WARNING by default). When deciding whether to process an event, the effective level of the logger is used to determine whether the event is passed to the logger’s handlers.

Child loggers propagate messages up to the handlers associated with their ancestor loggers. Because of this, it is unnecessary to define and configure handlers for all the loggers an application uses. It is sufficient to configure handlers for a top-level logger and create child loggers as needed. (You can, however, turn off propagation by setting the propagate attribute of a logger to False.)