logging — Report Status, Error, and Informational Messages

Purpose: Report status, error, and informational messages.

The `logging` module defines a standard API for reporting errors and status information from applications and libraries. The key benefit of having the logging API provided by a standard library module is that all Python modules can participate in logging, so an application's log can include messages from third-party modules.

Logging Components

The logging system is made up of four interacting types of objects. Each module or application that wants to log uses a `Logger` instance to add information to the logs. Invoking the logger creates a `LogRecord`, which is used to hold the information in memory until it is processed. A `Logger` may have a number of `Handler` objects configured to receive and process log records. The `Handler` uses a `Formatter` to turn the log records into output messages.

Logging in Applications vs. Libraries

Application developers and library authors can both use `logging`, but each audience has different considerations to keep in mind.

Application developers configure the `logging` module, directing the messages to appropriate output channels. It is possible to log messages with different verbosity levels or to different destinations. Handlers for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, or OS-specific logging mechanisms are all included, and it is possible to create custom log destination classes for special requirements not handled by any of the built-in classes.

Developers of libraries can also use `logging` and have even less work to do. Simply create a logger instance for each context, using an appropriate name, and then log messages using the standard levels. As long as a library uses the logging API with consistent naming and level selections, the application can be configured to show or hide messages from the library, as desired.

Logging to a File

Most applications are configured to log to a file. Use the `basicConfig()` function to set up the default handler so that debug messages are written to a file.

```
# logging_file_example.py

import logging

LOG_FILENAME = 'logging_example.out'
logging.basicConfig(
    filename=LOG_FILENAME,
    level=logging.DEBUG,
)

logging.debug('This message should go to the log file')

with open(LOG_FILENAME, 'rt') as f:
    body = f.read()

print('FILE:')
print(body)
```

After running the script, the log message is written to `logging_example.out`:

```
$ python3 logging_file_example.py

FILE:
DEBUG:root:This message should go to the log file
```

Rotating Log Files

Running the script repeatedly causes more messages to be appended to the file. To create a new file each time the program runs, pass a `filemode` argument to `basicConfig()` with a value of `'w'`. Rather than managing the creation of files this way, though, it is better to use a `RotatingFileHandler`, which creates new files automatically and preserves the old log file at the same time.

```
# logging_rotatingfile_example.py

import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
    LOG_FILENAME,
    maxBytes=20,
    backupCount=5,
)
my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)
for filename in sorted(logfiles):
    print(filename)
```

The result is six separate files, each with part of the log history for the application.

```
$ python3 logging_rotatingfile_example.py

logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

The most current file is always `logging_rotatingfile_example.out`, and each time it reaches the size limit it is renamed with the suffix `.1`. Each of the existing backup files is renamed to increment the suffix (`.1` becomes `.2`, etc.) and the `.5` file is erased.

Note

Obviously, this example sets the log length much too small as an extreme example. Set `maxBytes` to a more appropriate value in a real program.

Verbosity Levels

Another useful feature of the `logging` API is the ability to produce different messages at different *log levels*. This means code can be instrumented with debug messages, for example, and the log level can be set so that those debug messages are not written on a production system. the table below lists the logging levels defined by `logging`.

Logging Levels

Level	Value
CRITICAL	50
ERROR	40

WARNING	30
INFO	20
DEBUG	10
UNSET	0

The log message is only emitted if the handler and logger are configured to emit messages of that level or higher. For example, if a message is `CRITICAL`, and the logger is set to `ERROR`, the message is emitted (50 > 40). If a message is a `WARNING`, and the logger is set to produce only messages set to `ERROR`, the message is not emitted (30 < 40).

```
# logging_level_example.py

import logging
import sys

LEVELS = {
    'debug': logging.DEBUG,
    'info': logging.INFO,
    'warning': logging.WARNING,
    'error': logging.ERROR,
    'critical': logging.CRITICAL,
}

if len(sys.argv) > 1:
    level_name = sys.argv[1]
    level = LEVELS.get(level_name, logging.NOTSET)
    logging.basicConfig(level=level)

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 error message')
```

Run the script with an argument like ‘debug’ or ‘warning’ to see which messages show up at different levels:

```
$ python3 logging_level_example.py debug

DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message

$ python3 logging_level_example.py info

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

Naming Logger Instances

All of the previous log messages all have ‘root’ embedded in them because the code uses the root logger. An easy way to tell where a specific log message comes from is to use a separate logger object for each module. Log messages sent to a logger include the name of that logger. Here is an example of how to log from different modules so it is easy to trace the source of the message.

```
# logging_modules_example.py

import logging

logging.basicConfig(level=logging.WARNING)

logger1 = logging.getLogger('package1.module1')
logger2 = logging.getLogger('package2.module2')
```

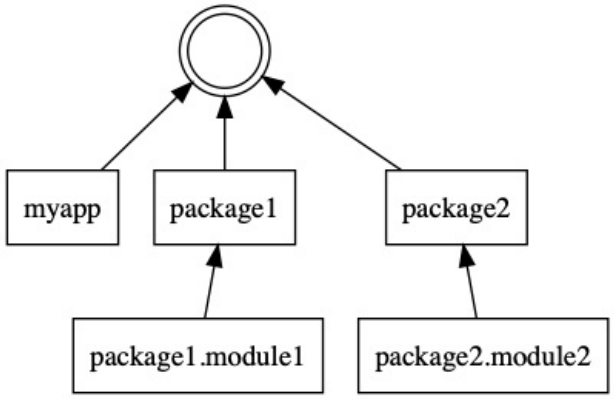
```
logger1.warning('This message comes from one module')
logger2.warning('This comes from another module')
```

The output shows the different module names for each output line.

```
$ python3 logging_modules_example.py

WARNING:package1.module1:This message comes from one module
WARNING:package2.module2:This comes from another module
```

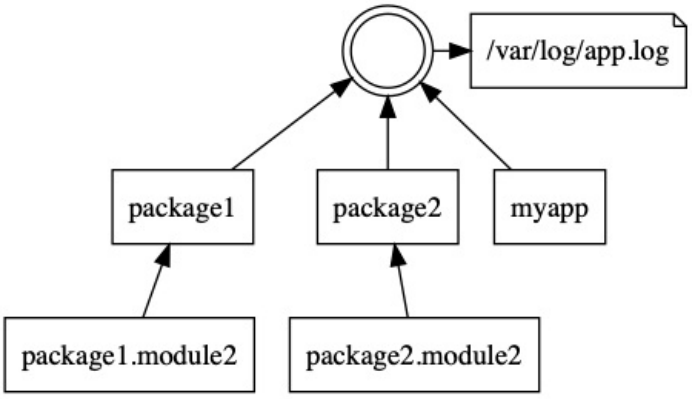
The Logging Tree

The Logger instances are configured in a tree structure, based on their names, as illustrated in the figure. Typically each application or library defines a base name, with loggers for individual modules set as children. The root logger has no name.



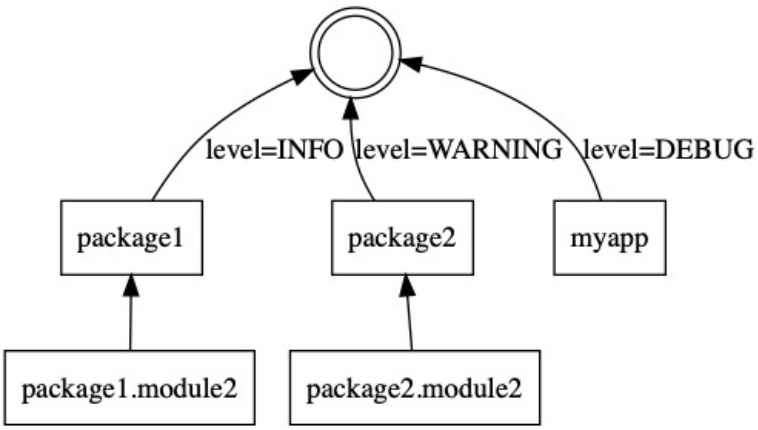
Example Logger Tree

The tree structure is useful for configuring logging because it means each logger does not need its own set of handlers. If a logger does not have any handlers, the message is handed to its parent for processing. This means that for most applications it is only necessary to configure handlers on the root logger, and all log information will be collected and sent to the same place, as shown in the figure.



One Logging Handler

The tree structure also allows different verbosity levels, handlers, and formatters to be set for different parts of the application or library to control which messages are logged and where they go, as in the figure.



Integration with the warnings Module

The logging module integrates with warnings through captureWarnings(), which configures warnings to send messages through the logging system instead of outputting them directly.

```
# logging_capture_warnings.py

import logging
import warnings

logging.basicConfig(
    level=logging.INFO,
)

warnings.warn('This warning is not sent to the logs')

logging.captureWarnings(True)

warnings.warn('This warning is sent to the logs')
```

The warning message is sent to a logger named py.warnings using the WARNING level.

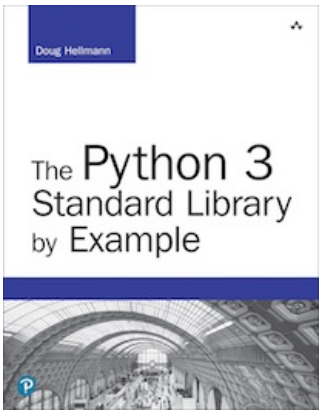
```
$ python3 logging_capture_warnings.py

logging_capture_warnings.py:13: UserWarning: This warning is not
 sent to the logs
  warnings.warn('This warning is not sent to the logs')
WARNING:py.warnings:logging_capture_warnings.py:17: UserWarning:
 This warning is sent to the logs
  warnings.warn('This warning is sent to the logs')
```

See also

- Standard library documentation for logging – The documentation for logging is extensive, and includes tutorials and reference material that goes beyond the exmaples presented here.
- Python 2 to 3 porting notes for logging
- warnings – Non-fatal alerts.
- logging_tree – Third-party package by Brandon Rhodes for showing the logger tree for an application.
- Logging Cookbook – Part of the standard library documentation, with examples of using logging for different tasks.

This page was last updated 2016-12-28.

Navigation
configparser — Work with Configuration Files
fileinput — Command-Line Filter Framework



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site
Module Index
Index

Other Writing
Blog
The Python Standard Library By Example