

## Python Logging

### Logging

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.

### Use logging

Logging provides a set of convenience functions for simple logging usage. These are `debug()`, `info()`, `warning()`, `error()` and `critical()`. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	<code>print()</code>
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	<code>logging.info()</code> (or <code>logging.debug()</code> for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	<code>warnings.warn()</code> in library code if the issue is avoidable and the client application should be modified to eliminate the warning <code>logging.warning()</code> if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	<code>logging.error()</code> , <code>logging.exception()</code> or <code>logging.critical()</code> as appropriate for the specific error and application domain

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it's used
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	Confirmation that things are working as expected.
WARNING	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
ERROR	Due to a more serious problem, the software has not been able to perform some function.
CRITICAL	A serious error, indicating that the program itself may be unable to continue running.

The default level is WARNING, which means that only events of this level and above will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. Another common way is to write them to a disk file.

### **Advanced Logging**

The logging library takes a modular approach and offers several categories of components: loggers, handlers, filters, and formatters.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

**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.

**Handler** objects are responsible for dispatching the appropriate log messages (based on the log messages' severity) to the handler's specified destination. Logger objects can add zero or more handler objects to themselves with an `addHandler()` method. As an example scenario, an application may want to send all log messages to a log file, all log messages of error or higher to stdout, and all messages of critical to an email address. This scenario requires three individual handlers where each handler is responsible for sending messages of a specific severity to a specific location.

**Formatter** objects configure the final order, structure, and contents of the log message. Unlike the base `logging.Handler` class, application code may instantiate formatter classes, although you could likely subclass the formatter if your application needs special behavior. The constructor takes three optional arguments – a message format string, a date format string and a style indicator.

The following example configures a very simple logger, a console handler, and a simple formatter using Python code:

```
import logging
```

```
# create logger
```

```
logger = logging.getLogger('simple_example')
```

```
logger.setLevel(logging.DEBUG)
```

```
# create console handler and set level to debug
```

```
ch = logging.StreamHandler()
```

```
ch.setLevel(logging.DEBUG)
```

```
# create formatter
```

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -  
%(message)s')
```

```
# add formatter to ch
```

```
ch.setFormatter(formatter)
```

```
# add ch to logger
```

```
logger.addHandler(ch)
```

```
# 'application' code
```

```
logger.debug('debug message')
```

```
logger.info('info message')
```

```
logger.warning('warn message')
```

```
logger.error('error message')
```


```
logger.critical('critical message')
```

Running this module from the command line produces the following output:

\$ python filename.py

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

## My Output :

 user1@vml:~

```
[user1@vml ~]$ cat word.py
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

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

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
[user1@vml ~]$
[user1@vml ~]$ python word.py
2022-03-24 08:29:29,461 - simple_example - DEBUG - debug message
2022-03-24 08:29:29,461 - simple_example - INFO - info message
2022-03-24 08:29:29,461 - simple_example - WARNING - warn message
2022-03-24 08:29:29,461 - simple_example - ERROR - error message
2022-03-24 08:29:29,461 - simple_example - CRITICAL - critical message
[user1@vml ~]$
```