How to enable and use logging module in Python

Author : Ramanathan Muthaiah

Categories : Development, Python

Tagged as : log filesyslog

Question: I am writing a Python program, and I would like to use Python's built-in `logging` facility to debug the program. How can I enable and use the `logging` module in Python?

Logging is an essential debugging feature for any programming and scripting language, which goes beyond simple `print` statements. For example, logging allows you to track in which module/function/linenumber logging messages are generated. You can also differentiate logging based on severity, and can direct logging messages to stdout/stderr, a file, a network socket, etc.

In Python, the `logging` module is part of the standard package and no special installation is required. This tutorial will walk you through the basic steps in enabling and configuring `logging` in Python.

To use the `logging` module, first import the module in your code.

```
import logging
```

There are varied levels of severity and verbosity supported by `logging` module. The following table shows different levels of logging, in the order of increasing severity (or decreasing verbosity).

Type	Level	When it is used
DEBUG	10	Show detailed information that can help with program diagnosis and troubleshooting.
INFO	20	Everything is running as expected without any problem.
WARNING	30	The program continues running, but something unexpected happened, which may lead to some problem down the road.
ERROR	40	The program fails to perform a certain function due to a bug.
CRITICAL	50	The program encounters a serious error and may stop running.

Unless you set logging to a particular level, the logging level is automatically set to WARNING by default. Once the logging level is set, any events whose severity is higher than the set level will be tracked and

printed by the `logging` module.

For example, the following Python code will print the warning statement, but not the info statement, since the default logging level is WARNING, and INFO has lower severity than WARNING.

```
  import logging  logging.warning('warning statement')  logging.info('info stat
ement')
```

Output of the code:

```
  WARNING:root:warning statement
```

On the other hand, if you change the logging level to INFO, both statements will be printed.

```
  import logging  logging.basicConfig(level=logging.INFO)  # set logging level
to INFO  logging.warning('warning statement')  logging.info('info statement')
```

Output of the code:

```
  WARNING:root:warning statement  INFO:root:info statement
```

As you can see, you can change the verbosity of logging simply by changing the logging level, but without changing individual logging statements.

How to Configure Logging Message Format

Beyond changing the logging level, the `logging` module allows you to customize logging message format. To do so, you will have to configure several logging parameters in the module.

Below is the sample code snippet that shows how to configure logging parameters.

```
  logLevel = logging.DEBUG  logFormat = "{{ %(asctime)s == %(levelname)-8s == M
odule:%(module)s Function:%(funcName)s Line:%(lineno)d }} %(message)s"  logging
.basicConfig(level=logLevel, format=logFormat, datefmt='%m/%d/%Y %I:%M:%S %p')
```

Let's examine the parameters passed to `basicConfig` method of `logging` module.

- **level**: this parameter specifies a desirable logging level (DEBUG in this example).
- **format**: this parameter defines the format in which a logging message is printed. A couple of keywords can be used here. For example, %(asctime) for current date/time, %(levelname) for name of the logging level, %(module) for name of the current module, %(funcName) for name of the current function, %(lineno) for code line number where the message is emitted, and %(message) for actual logging message.
- **datefmt**: this parameter defines the format of date and time for %(asctime). For example, %m (month), %d (date), %Y (Year), %I (24-hour format), %M (minutes), %S (seconds), %p (AM or PM).

Here is a complete code that puts together individual parts that we have discussed.

```
  #!/usr/bin/env python  import logging  def callDummy():      logging.info("IN
FO message spewed out from the callDummy module")      logging.debug("DEBUG mes
sage spewed out from the callDummy module")      return ' '  def main():      l
ogLevel = logging.DEBUG      logFormat = "{{ %(asctime)s == %(levelname)-8s ==
Module:%(module)s Function:%(funcName)s Line:%(lineno)d }} %(message)s"      lo
gging.basicConfig(level=logLevel, format=logFormat, datefmt='%m/%d/%Y %I:%M:%S
%p')      logging.info("INFO message spewed out from the main module")      log
ging.debug("DEBUG message spewed out from the main module")      callDummy()
   return ' '  if __name__ == "__main__":      main()
```

Output of this code is shown below.

```
$ python test_logging.py
{{ 06/03/2016 09:24:18 PM == INFO     == Module:test_logging Function:main Line:11 }} INFO message spewed out from the main
module
{{ 06/03/2016 09:24:18 PM == DEBUG    == Module:test_logging Function:main Line:12 }} DEBUG message spewed out from the main
 module
{{ 06/03/2016 09:24:18 PM == INFO     == Module:test_logging Function:callDummy Line:4 }} INFO message spewed out from the c
allDummy module
{{ 06/03/2016 09:24:18 PM == DEBUG    == Module:test_logging Function:callDummy Line:5 }} DEBUG message spewed out from the
callDummy module
$
```

The `logging` module provides much more enhanced capabilities than illustrated here. For example, you can provision for log messages to be saved to a rotated logfile or sent to a remote syslog server over the network. You can also extend logging across multiple modules within a Python package. For more comprehensive information on Python logging, refer to the official reference.