**Python loggers**

**What is log file in programming?**

Log file is a file that contains information about usage pattern, tracking of an event that occur when software runs also provides information about number of requests received to the application. In short it stores the logging records.

Suppose there is no logging record, and the program is interrupted during its execution, we will be unable to find the actual cause of the problem. That’s why log file plays an important role in any programming language.

**What is logging in python?**

Logging is a [Python](https://www.javatpoint.com/python-tutorial) module in the standard library that provides the facility to work with the framework for releasing log messages from the [Python programs](https://www.javatpoint.com/python-programs).

Logging is important for software developing, debugging, and running

**What if we don’t have log file?**

If you don’t have any logging record and your program crashes, there are very few chances that you detect the cause of the problem. And if you detect the cause, it will consume a lot of time. With logging if something goes wrong, we can determine the cause of the problem.

**Why Printing is not a good option?**

Some developers use the concept of printing the statements to validate if the statements are executed correctly or some error has occurred. But printing is not a good idea. It may solve your issues for simple scripts but for complex scripts, the printing approach will fail

**Logging file contains?**

The file can contain the information on which part of the code is executed and what problems have been arisen.

## Levels of Log Message

|  |  |  |  |
| --- | --- | --- | --- |
| NO. | Levels | Numeric value | Explanation |
| 1 | Critical | 50 | Represents a very serious problem that needs high attention.  Indicating that the program itself may be unable to continue running |
| 2 | Error | 40 | Represents a serious error. due to a more serious problem, the software has not been able to perform some function. |
| 3 | Warning | 30 | Represents a warning message, some caution needed. It is alert to the programmer. These are used an indication that something unexpected happened, or is indicative of some problem in the near future |
| 4 | Info | 20 | These are used to confirm that things are working as expected. Represents a message with some important information |
| 5 | Debug | 10 | These are used to give Detailed information, typically of interest only when diagnosing problems. Or debugging information. |
| 6 | Notset | 0 | Represents that the level is not set. |

Logging module is packed with several features. It has several constants, classes, and methods. The items with all caps are constant, the capitalize items are classes and the items which start with lowercase letters are methods

Example 1 :

*import logging*

*logging.debug('The debug message is displaying')*

*logging.info ('The info message is displaying')*

*logging.warning('The warning message is displaying')*

*logging.error('The error message is displaying')*

*logging.critical('The critical message is displaying')*

each message is displayed along with the root, which is the logging module name given to its default logger. The message and the level name are separated by a colon (:) and print the messages in default output format.

**debug()** and **info()** message didn't display messages because, by default, the log module logs the messages with a severity level of **WARNING, ERROR and CRITICAL**.

**BasicConfig (\*\*kwargs)**

The basicConfig configures the root logger. It does basic configuration for the logging system by creating a stream handler (output on console)with a default formatter(default formatting)

To perform logging, first we required to create a file to store messages and we have to specify which level messages we have to store

It accepts some of the commonly used argument as follows.

* **level -** The specified severity level is set by the root level.
* **filename -** It specifies a file.
* **filemode -** It opens a file in a specific mode. The default mode of the opening file is a, which means we can append the content.
* **format -** The format defines the format of the log message.

Example:

*import logging*

*logging.basicConfig(level=logging.DEBUG)*

*logging.debug('The dubug message is logged')*

Example:

*import logging*

*logging.basicConfig(filename='msg.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')*

*logging.warning('This will get logged to a file')*

## Formatting the Output

A string passed in the program as a message to log can be modified according to our requirements. There are some basic elements in the given string and part of the **Logrecord**. Let's understand the following example.

*import logging*

*logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')*

*logging.warning('This is a Warning Message')*

*import logging*

*logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)*

*logging.info('Admin logged in')*

**Logging variable data**

we want to include the dynamic information from the application in the log. The logging methods are accepted a string as an argument, and it is good practice to format a string with the variable data and passed to a log method.

But instead of that, we can also use a format string for the message and appending the variable data as an argument.

*import logging*

*name = 'Peter Decosta'*

*logging.error('%s raised an error', name)*

**OR**

*import logging*

*name = 'Antonio Mino'*

*logging.error(f'{name} raised an error')*

**Python logging getLogger**

Loggers are never instantiated directly, but always through the module-level function logging.getLogger(name)

The getLogger returns a logger with the specified name. If no name is specified, it returns the root logger.

If the **getLogger()** method is called multiple times with the same name, it will return the reference of the same logger object.

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.

*Example:*

*import logging*

*logger = logging.getLogger('first\_logger')*

*logger.warning('This is a warning message')*

*Explanation :* We have created the own logger name **first\_logger**, but unlike the root logger, the **first\_logger** is not part of the output format. To display it, pass it into the configuration function. Then the output will look like as follows.

Output= > WARNING:first\_logger:This is a warning message

*Example*

***#creating logging object***

*import logging*

*#Create and configure logger using the basicConfig() function*

*logging.basicConfig(filename="newfile.log",*

*format='%(asctime)s %(message)s',*

*filemode='w')*

*#Creating an object of the logging*

*main=logging.getLogger(‘*main’)

*main.setLevel(logging.DEBUG)*

*#A logger named main is created; we set the logging level to DEBUG.*

*main.warning("It is a Warning. Please make changes")*

*main.error("You are trying to divide by zero")*

*main.critical("Internet is down")*

**#with file handler**

## *#with file handler*

## *import logging*

## *import sys*

## *main = logging.getLogger('main')*

## *main.setLevel(logging.DEBUG)*

## *handler = logging.FileHandler('my.log')*

## *#A file handler is ceated. The messages will be written to the my.log file.*

## *format = logging.Formatter('%(asctime)s %(name)s %(levelname)s: %(message)s')*

## *#A formatter is created. It includes the time, the logger name,*

## *#the logging level, and the message in to log. The formatter is set to the handler with setFormatter.*

## *handler.setFormatter(format)*

## *main.addHandler(handler)*

## *main.info('info message')*

## *main.critical('critical message')*

## *main.debug('debug message')*

## *main.warning('warning message')*

## *main.error('error message')*

**There are several logger objects offered by the module itself.**

1. **Logger.info(msg) :** This will log a message with level INFO on this logger.
2. **Logger.warning(msg) :** This will log a message with a level WARNING on this logger.
3. **Logger.error(msg) :** This will log a message with level ERROR on this logger.
4. **Logger.critical(msg) :** This will log a message with level CRITICAL on this logger.
5. **Logger.log(lvl,msg) :** This will Logs a message with integer level lvl on this logger.
6. **Logger.exception(msg) :** This will log a message with level ERROR on this logger.
7. **Logger.setLevel(lvl) :** This function sets the threshold of this logger to lvl. This means that all the messages below this level will be ignored.
8. **Logger.addFilter(filt) :** This adds a specific filter filt into this logger.
9. **Logger.removeFilter(filt) :** This removes a specific filter filt into this logger.
10. **Logger.filter(record) :** This method applies the logger’s filter to the record provided and returns True if the record is to be processed. Else, it will return False.
11. **Logger.addHandler(hdlr) :** This adds a specific handler hdlr to this logger.
12. **Logger.removeHandler(hdlr) :** This removes a specific handler hdlr into this logger.
13. **Logger.hasHandlers() :** This checks if the logger has any handler configured or not.

## Classes and Functions

We have seen so far, the default logger called **root**. The logging module is used it whenever its functions are called such as **logging.debug(),** logging.error(), etc. We can also define own logger by creating an object of the **Logger** class. Here, we are defining the commonly used classes and functions.

Below are the classes and functions defined in the logging module.

* **Logger -** The logger object is used to call the functions directly.
* **LogRecord -** It creates automatically log record file which consists the information related to all event of being logged such as the logger's name, the function, the line number, the message, and more.
* **Handler -** The handlers are used to dispatch the **LogRecord** to the output endpoint. The **FileHandler, StreamHandler, HTTPHandler, SMTTPHandler** are the subclasses of a **Handler**. These subclasses send the logging outputs to corresponding destinations,
* **Formatters -** The formatters are used to define the structure of the output. It is used the string formatting methods to specify the format of the log messages.

**Work with handlers**

Handlers come into the picture when you want to configure your own loggers and send the logs to multiple places when they are generated. Handlers send the log messages to configured destinations like the standard output stream or a file or over HTTP or to your email via SMTP.

A logger that you create can have more than one handler, which means you can set it up to be saved to a log file and also send it over email.

Like loggers, you can also set the severity level in handlers. This is useful if you want to set multiple handlers for the same logger but want different severity levels for each of them. For example, you may want logs with level WARNING and above to be logged to the console, but everything with level ERROR and above should also be saved to a file. Here’s a program that does that:

**Example:**

*#working with handlers*

*# logging\_example.py*

*import logging*

*# Create a custom logger*

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

*# Create handlers*

*c\_handler = logging.StreamHandler()*

*f\_handler = logging.FileHandler('file.log')*

*c\_handler.setLevel(logging.WARNING)*

*f\_handler.setLevel(logging.ERROR)*

*# Create formatters and add it to handlers*

*c\_format = logging.Formatter('%(name)s - %(levelname)s - %(message)s')*

*f\_format = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')*

*c\_handler.setFormatter(c\_format)*

*f\_handler.setFormatter(f\_format)*

*# Add handlers to the logger*

*logger.addHandler(c\_handler)*

*logger.addHandler(f\_handler)*

*logger.warning('This is a warning')*

*logger.error('This is an error')*

*Explanation:*

Here, logger.warning() is creating a LogRecord that holds all the information of the event and passing it to all the Handlers that it has: c\_handler and f\_handler.

c\_handler is a StreamHandler with level WARNING and takes the info from the LogRecord to generate an output in the format specified and prints it to the console. f\_handler is a FileHandler with level ERROR, and it ignores this LogRecord as its level is WARNING.

When logger.error() is called, c\_handler behaves exactly as before, and f\_handler gets a LogRecord at the level of ERROR, so it proceeds to generate an output just like c\_handler, but instead of printing it to console, it writes it to the specified file in this format:

The name of the logger corresponding to the \_\_name\_\_ variable is logged as \_\_main\_\_, which is the name Python assigns to the module where execution starts. If this file is imported by some other module, then the \_\_name\_\_ variable would correspond to its name

**Capturing Stack traces**

The logging module also allows you to capture the full stack traces in an application. [Exception information](https://realpython.com/python-exceptions/) can be captured if the exc\_info parameter is passed as True, and the logging functions are called like this:

Example:

*import logging*

*a = 5*

*b = 0*

*try:*

*c = a / b*

*except Exception as e:*

*logging.error("Exception occurred", exc\_info=True)*

If exc\_info is not set to True, the output of the above program would not tell us anything about the exception, which, in a real-world scenario, might not be as simple as a ZeroDivisionError. Imagine trying to debug an error in a complicated codebase with a log that shows only this:

**Logging.exception()-**

Here’s a quick tip: if you’re logging from an exception handler, use the logging.exception() method, which logs a message with level ERROR and adds exception information to the message. To put it more simply, calling logging.exception() is like calling logging.error(exc\_info=True). But since this method always dumps exception information, it should only be called from an exception handler. Take a look at this example:

#logging.exception

*import logging*

*a = 5*

*b = 0*

*try:*

*c = a / b*

*except Exception as e:*

*logging.exception("Exception occurred")*

Using logging.exception() would show a log at the level of ERROR. If you don’t want that, you can call any of the other logging methods from debug() to critical() and pass the exc\_info parameter as True.