Logging in Python

by Abhinav Ajitsaria Sep 12, 2018 53 Comments intermediate tools

Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Logging in Python**

Logging is a very useful tool in a programmer's toolbox. It can help you develop a better understanding of the flow of a program and discover scenarios that you might not even have thought of while developing.

Logs provide developers with an extra set of eyes that are constantly looking at the flow that an application is going through. They can store information, like which user or IP accessed the application. If an error occurs, then they can provide more insights than a stack trace by telling you what the state of the program was before it arrived at the line of code where the error occurred.

Help

By logging useful data from the right places, you can not only debug errors easily but also use the data to analyze the

By logging useful data from the right places, you can not only debug errors easily but also use the data to analyze the performance of the application to plan for scaling or look at usage patterns to plan for marketing.

Python provides a logging system as a part of its standard library, so you can quickly add logging to your application. In this article, you will learn why using this module is the best way to add logging to your application as well as how to get started quickly, and you will get an introduction to some of the advanced features available.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take your Python skills to the next level.

The Logging Module

The logging module in Python is a ready-to-use and powerful module that is designed to meet the needs of beginners as well as enterprise teams. It is used by most of the third-party Python libraries, so you can integrate your log messages with the ones from those libraries to produce a homogeneous log for your application.

Adding logging to your Python program is as easy as this:

Python
```python
import logging
```

With the logging module imported, you can use something called a "logger" to log messages that you want to see. By default, there are 5 standard levels indicating the severity of events. Each has a corresponding method that can be used to log events at that level of severity. The defined levels, in order of increasing severity, are the following:

- DEBUG
- INFO
- WARNING
- ERROR
- CRITICAL

The logging module provides you with a default logger that allows you to get started without needing to do much configuration. The corresponding methods for each level can be called as shown in the following example:

Python
```python
import logging

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 message')
```

The output of the above program would look like this:

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

The output shows the severity level before each message along with `root`, which is the name the logging module gives to its default logger. (Loggers are discussed in detail in later sections.) This format, which shows the level, name, and message separated by a colon (`:`), is the default output format that can be configured to include things like timestamp, line number, and other details.

Notice that the `debug()` and `info()` messages didn't get logged. This is because, by default, the logging module logs the messages with a severity level of `WARNING` or above. You can change that by configuring the logging module to lo

Help

events of all levels if you want. You can also define your own severity levels by changing configurations, but it is generally not recommended as it can cause confusion with logs of some third-party libraries that you might be using.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com

Basic Configurations

You can use the `basicConfig(**kwargs)` method to configure the logging:

> "You will notice that the logging module breaks PEP8 styleguide and uses `camelCase` naming conventions. This is because it was adopted from Log4j, a logging utility in Java. It is a known issue in the package but by the time it was decided to add it to the standard library, it had already been adopted by users and changing it to meet PEP8 requirements would cause backwards compatibility issues." (Source)

Some of the commonly used parameters for `basicConfig()` are the following:

- `level`: The root logger will be set to the specified severity level.
- `filename`: This specifies the file.
- `filemode`: If `filename` is given, the file is opened in this mode. The default is `a`, which means append.
- `format`: This is the format of the log message.

By using the `level` parameter, you can set what level of log messages you want to record. This can be done by passing one of the constants available in the class, and this would enable all logging calls at or above that level to be logged. Here's an example:

Python

```python
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('This will get logged')
```

Shell

```
DEBUG:root:This will get logged
```

All events at or above `DEBUG` level will now get logged.

Similarly, for logging to a file rather than the console, `filename` and `filemode` can be used, and you can decide the format of the message using `format`. The following example shows the usage of all three:

Python

```python
import logging

logging.basicConfig(filename='app.log', filemode='w', format='%(name)s - %(levelname)s - %(message)s')
logging.warning('This will get logged to a file')
```

Shell

```
root - ERROR - This will get logged to a file
```

The message will look like this but will be written to a file named `app.log` instead of the console. The filemode is set to `w`, which means the log file is opened in "write mode" each time `basicConfig()` is called, and each run of the program will rewrite the file. The default configuration for filemode is `a`, which is append.

You can customize the root logger even further by using more parameters for `basicConfig()`, which can be found he

It should be noted that calling `basicConfig()` to configure the root logger works only if the root logger has not been configured before. **Basically, this function can only be called once.**

debug(), info(), warning(), error(), and critical() also call basicConfig() without arguments automatically if it has not been called before. This means that after the first time one of the above functions is called, you can no longer configure the root logger because they would have called the basicConfig() function internally.

The default setting in basicConfig() is to set the logger to write to the console in the following format:

Shell

```
ERROR:root:This is an error message
```

Formatting the Output

While you can pass any variable that can be represented as a string from your program as a message to your logs, there are some basic elements that are already a part of the LogRecord and can be easily added to the output format. If you want to log the process ID along with the level and message, you can do something like this:

Python

```python
import logging

logging.basicConfig(format='%(process)d-%(levelname)s-%(message)s')
logging.warning('This is a Warning')
```

Shell

```
18472-WARNING-This is a Warning
```

format can take a string with LogRecord attributes in any arrangement you like. The entire list of available attributes can be found here.

Here's another example where you can add the date and time info:

Python

```python
import logging

logging.basicConfig(format='%(asctime)s - %(message)s', level=logging.INFO)
logging.info('Admin logged in')
```

Shell

```
2018-07-11 20:12:06,288 - Admin logged in
```

%(asctime)s adds the time of creation of the LogRecord. The format can be changed using the datefmt attribute, which uses the same formatting language as the formatting functions in the datetime module, such as time.strftime():

Python

```python
import logging

logging.basicConfig(format='%(asctime)s - %(message)s', datefmt='%d-%b-%y %H:%M:%S')
logging.warning('Admin logged out')
```

Shell

```
12-Jul-18 20:53:19 - Admin logged out
```

Help

You can find the guide here.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com

Logging Variable Data

In most cases, you would want to include dynamic information from your application in the logs. You have seen that the logging methods take a string as an argument, and it might seem natural to format a string with variable data in a separate line and pass it to the log method. But this can actually be done directly by using a format string for the message and appending the variable data as arguments. Here's an example:

Python

```python
import logging

name = 'John'

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

Shell

```
ERROR:root:John raised an error
```

The arguments passed to the method would be included as variable data in the message.

While you can use any formatting style, the f-strings introduced in Python 3.6 are an awesome way to format strings as they can help keep the formatting short and easy to read:

Python

```python
import logging

name = 'John'

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

Shell

```
ERROR:root:John raised an error
```

Capturing Stack Traces

The logging module also allows you to capture the full stack traces in an application. Exception information can be captured if the `exc_info` parameter is passed as `True`, and the logging functions are called like this:

Python

```python
import logging

a = 5
b = 0

try:
  c = a / b
except Exception as e:
  logging.error("Exception occurred", exc_info=True)
```

Shell

```
ERROR:root:Exception occurred
Traceback (most recent call last):
  File "exceptions.py", line 6, in <module>
    c = a / b
```

```
ZeroDivisionError: division by zero
[Finished in 0.2s]
```

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:

Shell

```
ERROR:root:Exception occurred
```

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:

Python

```python
import logging

a = 5
b = 0
try:
  c = a / b
except Exception as e:
  logging.exception("Exception occurred")
```

Shell

```
ERROR:root:Exception occurred
Traceback (most recent call last):
  File "exceptions.py", line 6, in <module>
    c = a / b
ZeroDivisionError: division by zero
[Finished in 0.2s]
```

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.

Classes and Functions

So far, we have seen the default logger named root, which is used by the logging module whenever its functions are called directly like this: logging.debug(). You can (and should) define your own logger by creating an object of the Logger class, especially if your application has multiple modules. Let's have a look at some of the classes and functions in the module.

The most commonly used classes defined in the logging module are the following:

- **Logger:** This is the class whose objects will be used in the application code directly to call the functions.
- **LogRecord:** Loggers automatically create LogRecord objects that have all the information related to the event being logged, like the name of the logger, the function, the line number, the message, and more.
- **Handler:** Handlers send the LogRecord to the required output destination, like the console or a file. Handler is a base for subclasses like StreamHandler, FileHandler, SMTPHandler, HTTPHandler, and more. These subclasses send the logging outputs to corresponding destinations, like sys.stdout or a disk file.
- **Formatter:** This is where you specify the format of the output by specifying a string format that lists out the attributes that the output should contain.

Out of these, we mostly deal with the objects of the Logger class, which are instantiated using the module-level function logging.getLogger(name). Multiple calls to getLogger() with the same name will return a reference to the same Logger object, which saves us from passing the logger objects to every part where it's needed. Here's an example:

Help

Python

```
import logging

logger = logging.getLogger('example_logger')
logger.warning('This is a warning')
```

Shell

```
This is a warning
```

This creates a custom logger named `example_logger`, but unlike the root logger, the name of a custom logger is not part of the default output format and has to be added to the configuration. Configuring it to a format to show the name of the logger would give an output like this:

Shell

```
WARNING:example_logger:This is a warning
```

Again, unlike the root logger, a custom logger can't be configured using `basicConfig()`. You have to configure it using Handlers and Formatters:

> "It is recommended that we use module-level loggers by passing `__name__` as the name parameter to `getLogger()` to create a logger object as the name of the logger itself would tell us from where the events are being logged. `__name__` is a special built-in variable in Python which evaluates to the name of the current module." (Source)

Using 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:

Python

```python
# 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)
```

Help

```python
# 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')
```

Shell

```
__main__ - WARNING - This is a warning
__main__ - ERROR - This is an error
```

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:

Shell

```
2018-08-03 16:12:21,723 - __main__ - ERROR - This is an error
```

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 *logging_example*. Here's how it would look:

Python

```python
# run.py

import logging_example
```

Shell

```
logging_example - WARNING - This is a warning
logging_example - ERROR - This is an error
```

Other Configuration Methods

You can configure logging as shown above using the module and class functions or by creating a config file or a dictionary and loading it using `fileConfig()` or `dictConfig()` respectively. These are useful in case you want to change your logging configuration in a running application.

Here's an example file configuration:

Config File

```
[loggers]
keys=root,sampleLogger

[handlers]
keys=consoleHandler

[formatters]
```

Help

```
keys=sampleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_sampleLogger]
level=DEBUG
handlers=consoleHandler
qualname=sampleLogger
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=sampleFormatter
args=(sys.stdout,)

[formatter_sampleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

In the above file, there are two loggers, one handler, and one formatter. After their names are defined, they are configured by adding the words logger, handler, and formatter before their names separated by an underscore.

To load this config file, you have to use `fileConfig()`:

Python

```python
import logging
import logging.config

logging.config.fileConfig(fname='file.conf', disable_existing_loggers=False)

# Get the logger specified in the file
logger = logging.getLogger(__name__)

logger.debug('This is a debug message')
```

Shell

```
2018-07-13 13:57:45,467 - __main__ - DEBUG - This is a debug message
```

The path of the config file is passed as a parameter to the `fileConfig()` method, and the `disable_existing_loggers` parameter is used to keep or disable the loggers that are present when the function is called. It defaults to `True` if not mentioned.

Here's the same configuration in a YAML format for the dictionary approach:

YAML

```yaml
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
```

Help

```
loggers:
  sampleLogger:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

Here's an example that shows how to load config from a `yaml` file:

Python

```python
import logging
import logging.config
import yaml

with open('config.yaml', 'r') as f:
    config = yaml.safe_load(f.read())
    logging.config.dictConfig(config)

logger = logging.getLogger(__name__)

logger.debug('This is a debug message')
```

Shell

```
2018-07-13 14:05:03,766 - __main__ - DEBUG - This is a debug message
```

Keep Calm and Read the Logs

The logging module is considered to be very flexible. Its design is very practical and should fit your use case out of the box. You can add basic logging to a small project, or you can go as far as creating your own custom log levels, handler classes, and more if you are working on a big project.

If you haven't been using logging in your applications, now is a good time to start. When done right, logging will surely remove a lot of friction from your development process and help you find opportunities to take your application to the next level.

Mark as Completed

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Logging in Python**