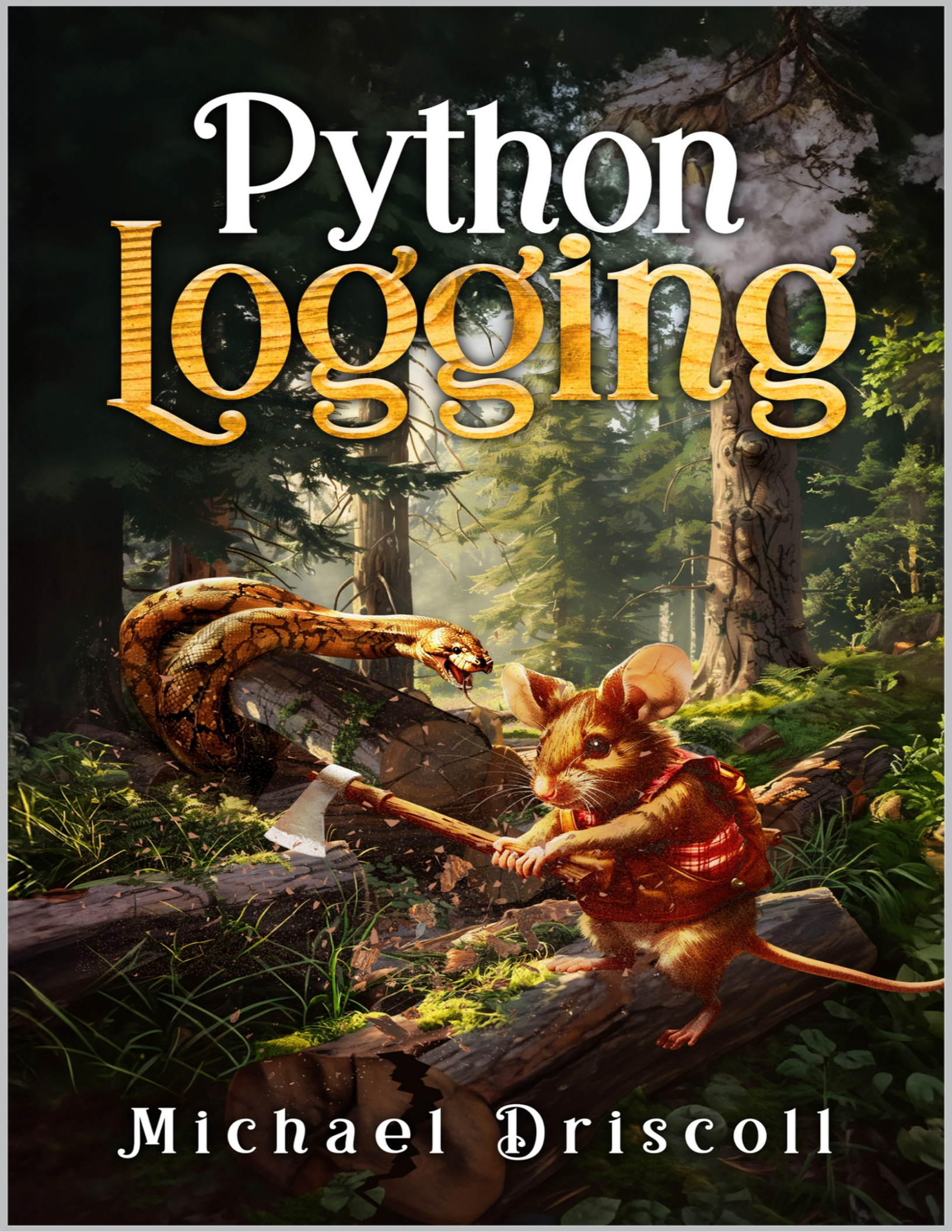
Python Logging



Michael Driscoll

Python Logging

Michael Driscoll

This book is for sale at http://leanpub.com/pythonlogging

This version was published on 2024-06-04

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

Acknowledgments

Thank you to all my readers and friends who continue to encourage me to write. Being an author can be a lonely practice with few who understand the ups and downs of publication.

Fortunately, I love writing and so far always come back for more. I have lots of ideas for new books and videos and I hope you will stick around and enjoy them as they come out.

Thanks again for all the positive feedback you give about my works.

Mike

Introduction to Python Logging

Welcome to **Python Logging**! Whether you're new to programming or an old hand, you have probably seen or interacted with logs.

A log is any information from the program that the original software developers thought would help debug their application. They usually consist of timestamped informational messages and different levels of errors.

Some companies require logs for auditing purposes, such as who was the last person to edit or change a setting or document. Most developers use logs to track down bugs. If you design your application correctly, you can use different logging levels to increase or decrease the log's verbosity.

Here are the topics that you will learn about in this book:

- Log levels
- Log handlers
- Log formatters
- Log Record objects
- Filter objects
- Lots of logging examples

That last one is vague, so here's some more context. The first half of the book will teach you all the things you need to do to be able to log in Python. The second half will be a kind of cookbook that shows a various code examples that help you learn to log effectively.

Audience

You don't need to be a programmer or engineer to use this book, but it helps. The primary target is people who want to learn about what logging in Python is and how to use it effectively. If you understand the basics of Python, then you'll be even better off!

About the Author

Mike Driscoll has been programming with Python for over a decade. When Mike isn't programming for work, he writes about Python on his blog and contributes to Real Python. He has worked with Packt Publishing and No Starch Press as a technical reviewer. Mike has also written several books. Mike also founded Teach Me Python where you can learn the Python programming language through his books and courses.

You can see a full listing of Mike's books on his website.

Mike frequently posts on X (formerly Twitter) about Python, writing, and other topics. You can follow him at @driscollis.

Conventions

Most technical books contain certain types of conventions. The primary convention in this book is code blocks that may or may not have syntax highlighting.

Here is an example:

```
1 my_string = "Welcome to logging with Python!"
```

These examples allow you to copy and paste the code directly from the book. However, if you have any issues with doing that, the next section will tell you where to download the code examples.

Book Source Code

If you ever need to grab a copy of some of the code from this book, then all you need to do is go to the book's GitHub repository here:

- https://github.com/driscollis/pythonlogging

Reader Feedback

I welcome feedback about my writing. If you'd like to let me know what you thought of the book, you can send comments to the following address:

- comments@pythonlibrary.org

Errata

I try to avoid publishing errors in my writing, but it happens occasionally. If you happen to see a mistake in this book, feel free to let me know by emailing me at the following:

- errata@pythonlibrary.org

Now, let's start learning!

Chapter 1 - An Intro to Logging

When you're a beginner and debugging your code, you probably use Python's handy `print()` function. There's nothing wrong with that. Using `print()` in this way is normal. However, when you push your code to production or share it on GitHub, you should remove those `print()` functions.

Why? Well, the problem is that you can accidentally leak sensitive information. For example, sometimes, you print out an API key or a password. When you use `print()`, it goes to standard out (stdout), and that can be seen by anyone running the code. You shouldn't share those kinds of things with just anyone. Someone might log in to your account, after all.

Of course, when you are logging in, you also don't want to log out your company's passwords or API keys. Anyone accessing those logs could steal those credentials and use them for nefarious purposes.

Your logs help you track your program's progress and fix it when it fails. Logs are also used as an auditing mechanism.

In this chapter, you will learn about the following:

- What are logs
- Why they are useful
- print() vs logging
- How to create a simple "Hello World" log

Let's get started!

What are logs?

A log is a human-readable text file. Sometimes, you might encounter a different format, but in 99% of cases, it's a file ending with the extension, `.log`. Most log files have a timestamp at the beginning of each line.

Here’s an example log message:

2023-08-02 15:37:40,592 - exampleApp - INFO - Program started

There are four parts to this log message:

- The timestamp
- The application name
- The log level (see chapter 2)
- The log message

You will spend most of this book learning all about these parts of the log message and how to create them. You’ll also learn how to add and remove the parts you need.

Why are logs useful?

Logs are useful for the following reasons:

- Debugging your code
- Tracing code paths
- Audits
- Root cause analysis
- Saves off error reports

Most of the items mentioned above are related to figuring out how your application works and what went wrong if anything. Audits are in the corporate domain and may be needed for governance or certification reasons.

print() versus logging

Python makes it easy to “log” out what’s happening in your program using the `print()` function.

There are several problems with using `print()` though:

- `print()` writes to standard out by default
- `print()` doesn't have timestamps or other niceties that are built into `logging`
- `print()` has no verbosity controls

Python has a solution to all these problems in the `logging` module, which comes with Python's standard library.

How to create a simple "Hello World" log

How do I create a log with Python? Thanks for asking. You will learn how to create a log with only a handful of lines of code!

Open up your favorite Python IDE or text editor and create a file named `hello_log.py`. Then enter the following code:

```
# hello_log.py

import logging

logging.basicConfig(filename="hello.log",
                    level=logging.INFO)

logging.debug("Hello debug")
logging.info("Hello info")
logging.error("Hello error!!!")
```

Here, you import Python's logging module and call `basicConfig()` to tell your logger where to write the log. You can pass in an absolute file path or a relative path. You can also set the logging level, which you'll learn about in the next chapter. For now, you set it to INFO for demonstration purposes.

The last three lines send debug, informational, and error log messages.

When you run this code, your code creates the `hello.log` file in the same location as your `hello_log.py` file.

If you open your log file, you will see the following contents:

```
INFO:root:Hello info
ERROR:root:Hello error!!!
```

Wait a minute! Shouldn't there be three lines of output in there? When you set the log level to INFO, you exclude any log levels below that level, such as debug. Anything at the INFO level and above will be logged into the file.

In this case, you get INFO and ERROR messages logged. Python's logger will automatically output the log level, where the name of the logger (root) and the log message are the default values. You will learn how to customize the log output in a future chapter.

Now you know the basics of creating a log! Congrats!

Wrapping Up

Logging is a great way to analyze the root causes of any issues that arise in your code.

In this chapter, you learned about the following:

- What are logs
- Why they are useful
- print() vs logging
- How to create a simple "Hello World" log

You are on your new journey into logging in using Python!

In the next chapter, you'll learn all about log levels. Get ready!

Chapter 2 - Log Levels

Logging levels allow you to control which messages you record in your logs. Think of log levels as verbosity levels. How granular do you want your logs to be?

Do you want to only log exceptions? Or would you like to log information and warnings too? You get to set all of these things using Python's `logging` module.

In this chapter, you will learn about the following:

- Available log levels
- Using log levels
- Logging exceptions

While this chapter will be pretty short, log levels are an important topic.

Let's get started!

Available log levels

Python's `logging` module comes with six logging levels built-in.

Here's a list of those log levels:

- notset (0) - Indicates that ancestor loggers should be consulted for the log level or that all events are logged (default setting)
- debug (10) - Detailed information that would be of interest to the developer for diagnostic purposes
- info (20) - Information that confirms that your application is working as expected
- warning (30) - Indicates that something unexpected happened or a problem may occur soon (like low disk space)
- error (40) - A serious issue has occurred, and the software was not able to function correctly

- critical (50) - A serious error that may cause or has caused the application to stop working

Each log level maps to a numeric value, the integer in parentheses above. Unless you plan to create your custom log levels, you don't need to know anything about those numeric values.

But if you want to define your custom log levels, you need to understand something. If you use the same numeric value as one of the pre-defined log levels, you will overwrite that level with your new one, which may have unintended consequences in your code.

So be careful!

Using log levels

Now that you know what a log level is and what you use it for, you must learn how to write the actual logging code!

Open up your Python editor of choice and create a new file named `log_levels.py`.

Then enter the following code:

```
1  # log_levels.py
2
3  import logging
4
5  logging.basicConfig(filename="log_levels.log",
6                      level=logging.DEBUG)
7
8  logging.debug("Hello debug")
9  logging.info("Hello info")
10 logging.warning("Hello warning")
11 logging.error("Hello error")
12 logging.critical("Hello critical!")
```

Here, you tell Python's logger to write to a new file named `log_levels.log`. Then, you write five different log messages using the five main log levels.

When you run this code, it will write out the following text to your log file:

```
DEBUG:root:Hello debug
INFO:root:Hello info
WARNING:root:Hello warning
ERROR:root:Hello error
CRITICAL:root:Hello critical!
```

To learn more about how setting a log level works, try changing the level you log at. But first, delete the original log, or you'll end up appending more data to it.

Now that you have deleted the log file, `log_levels.log`, change the `level` parameter in your Python script from `logging.DEBUG` to `logging.WARNING`.

Then re-run the code and open your new log. It should have the following information in it:

```
WARNING:root:Hello warning
ERROR:root:Hello error
CRITICAL:root:Hello critical!
```

By changing the log level, you control the granularity of the logs you save to your log file. You could add a command line interface or a graphical user interface to your application, allowing you to enable different log levels in your code to help debug.

For now, try changing the log level specified in the example code provided in this chapter. All you'll need to do is re-run the code each time.

Remember this: if you do not remove the log file before you run the code, the new run will append to the file. This can be potentially confusing since the log will have the same message repeatedly.

Logging Exceptions

You might think logging an error or a critical log message is the same as logging an exception, but you'd be mistaken. Python's `logging` module lets

you log an exception using the `exception()` method on the logger object. When you call that method, it will log the entire traceback that was raised when the exception occurred.

An example will help clarify what that vague statement means. Open up your Python IDE or text editor and create a new file named `log_exception.py`. Then enter the following code:

```
# log_exception.py

import logging

logger = logging.getLogger("excepter")
logger.setLevel(logging.INFO)

def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        logger.exception("An error has occurred!")
    except TypeError:
        logger.exception("Incompatible types!")
    else:
        return result

if __name__ == "__main__":
    divide(1, 0)
```

The code above will catch two types of errors:

- A `ZeroDivisionError` which occurs when you divide by zero
- A `TypeError` which occurs when you try to use incompatible types, such as strings and integers

When you use this code, you will hit the first exception handler and get the following output logged to your terminal or console:

```
An error has occurred!
Traceback (most recent call last):
  File "C:\code\log_exception.py", line 10, in divide
    result = a / b
             ~~^~~
ZeroDivisionError: division by zero
```

Logging exceptions allows you to troubleshoot your application and determine what went wrong. It is also a great tool for detecting bad user input.

Wrapping Up

Now that you know how log levels work in Python's logging module, you should be able to set the root logger's log level to whichever level you want to log.

Wait a minute! Why does this only apply to the root logger? Don't worry. You will learn how to create your custom logger soon. But it's good to learn the basics using the root logger as it uses less code and is easier to understand.

You can use the knowledge you've learned in this chapter to change the granularity of your logs. When you learn about creating your logger, you can use the concepts you learned in this chapter with your custom logger.

Python has great documentation about logging levels that you can refer to for additional information.

Chapter 3 - Log Handlers

Log handlers give you the ability to specify where your logs go. You don't have to write your logs to a file. You can write your logs to multiple other locations.

For example, you might want to log to standard out. That means you should write your logs to the terminal so you can read them in real-time. You could also create a log handler that rotates the log depending on how large the file gets or based on a certain amount of time, which you will learn more about later in this book.

Python's `logging` module comes with several different built-in logging handlers. This chapter does not teach you how to use all the different handlers, but you will at least learn what is available.

In this chapter, you will learn about the following:

- Available logging handlers
- Using logging handlers
- Using multiple handlers at once
- Other handler features

This chapter provides multiple examples of using logging handlers and teaches you about using a logger other than root.

Let's get started!

Available logging handlers

Many logging handlers are included in Python's `logging` module. You can read the full details in Python's documentation. However, you will learn about the most common ones in this chapter section.

Here are some of the ones you will most likely use:

- StreamHandler
- FileHandler
- RotatingFileHandler
- TimedRotatingFileHandler

If you happen to need to log using threads or processes, then you would probably use both of these:

- QueueHandler
- QueueListener

There are other handlers, too, but their use is much less frequent.

Let's move on and learn how to implement a logging handler!

Using basicConfig

The logging examples you have seen previously never used a handler directly. You technically used a file handler when you used `basicConfig()` in the last chapter, but it was used implicitly. Let's review!

You will again use the `basicConfig()` method to tell the `logging` module that you want to log into a file instead. Open up your favorite Python IDE or text editor and create a new file named `log2file.py`.

Then enter the following code:

```
# log2file.py

import logging

logging.basicConfig(filename="test.log",
                    level=logging.DEBUG)

logging.debug("Hello debug")
logging.info("Hello info")
```

Here, you tell Python you want to log to a file named `test.log` and then send a couple of log messages to it. Go ahead and run the code now.

You should see the following output in your test log file:

```
DEBUG:root:Hello debug
INFO:root:Hello info
```

Now you're ready to learn how to use an explicit logging handler!

Using a logging handler

Logging handlers are the way to go when creating a custom logger object. However, they do require a bit of boilerplate. What is boilerplate anyway? In programming terminology, boilerplate is repetitive code to setup or tear down something. If you create a lot of loggers, you can abstract this type of code into its module and reuse it that way.

But while learning, you don't need to worry about abstracting your code. You need to immerse yourself in how it works!

With that in mind, open up your Python IDE or text editor of choice and create a new file named `log2file_handler.py`. Then enter the following code:

```
# log2file_handler.py

import logging

# Create custom logger
logger = logging.getLogger(name="test")
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("test_handler.log")

# add handler to logger
logger.addHandler(file_handler)

logger.debug("Hello debug")
logger.info("Hello info")
```

The first new bit of code here uses the `getLogger()` method. You call `getLogger()` to get a logger object with a specific name. In this case, you create a "test" logger object.

Now that you have a logger object, you call `setLevel()` on your logger object to set the logging level. Now you're ready to learn about handlers!

For this example, you will create a `FileHandler()` to log to a text file on your computer. All you need to do to create a `FileHandler()` is pass in a path to the file you wish to log to.

The `logger` object has the handy `addHandler()` method to make adding a handler to your logger object easy. All you need to do is pass in the `file_handler` object you created to it.

The last two lines show how to emit logs at different levels: `debug` and `info`.

When you run this code, you will see a "test_handler.log" file created in the same folder as where you saved your Python code. When you open that file, you should see something like the following:

```
1 Hello debug
2 Hello info
```

Wait a minute! The text above looks different from the output you got in your previous code examples. What happened? Now that you have a custom handler, you need to add custom formatting to the output. Your log will only log the message, not the log level, timestamp, or anything else.

You will learn all about formatting your logs in the next chapter. For now, you need to focus on learning about the log handlers!

Speaking of which, it's time to learn how to log using two different handlers simultaneously!

Using multiple handlers at once

The beauty of having your custom logger is that you can add multiple handlers. That means you can log to a file, your terminal, and other locations all at the same time.

But why would you do something like that? Depending on your application, you may want to turn on real-time monitoring. That allows your application to log to the terminal so you can watch for errors and save those logs to a file for more in-depth research.

To make the code easier to digest, you will reuse the code from the last example and add a new handler to the mix.

Open up your Python IDE or text editor again and then enter the following code:

```
# two_handlers.py

import logging

# Create custom logger
logger = logging.getLogger(name="test")
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("two_handlers.log")
stream_handler = logging.StreamHandler()

# add handler to logger
logger.addHandler(file_handler)
logger.addHandler(stream_handler)

logger.debug("Hello debug")
logger.info("Hello info")
```

Your code has increased by two additional lines. In the first line, you create a `StreamHandler()` object. You can technically give `StreamHandler()` a stream to work with or use the default, which is to stream to standard out. You do the latter in this example.

In the second line, you call `addHandler()` to add your `stream_handler` object to your logger. That's it! You're done.

When you run this code in your IDE or your terminal, you will see the log messages printed out and saved to your log file. Give it a try and see for yourself!

Other handler features

There are a few other things you should know about logging handlers. A logging handler is thread-safe. They provide two methods you can use with threads:

- `acquire()` - acquire a thread lock via `createLock()`
- `release()` - release the thread lock you acquired

You can also add a formatter to format your log messages. You'll learn more about that in the next chapter.

Logs also support filters. A filter allows you to filter out what types of information you log. You'll learn more about filters later on in this book.

The other two methods you might find helpful with a log handler object are the following:

- `flush()` - Ensure all logging output is flushed. This method should be used by subclasses though
- `close()` - Used to tidy up any resources used by your handler

There are a few other logging handler methods, but their use is rare.

Wrapping Up

Python logging handlers are great. You can log to many different locations. For a full description of all the handlers that Python's `logging` module provides, see the official documentation.

In this chapter, you covered the following topics:

- Available logging handlers
- Using logging handlers
- Using multiple handlers at once
- Other handler features

Using logging handlers takes practice, but once you understand how they work, you'll be able to use them well, with little to no trouble at all. Try out some of the other logging handlers or experiment with the ones covered here. Soon, you, too, will be able to log with confidence!

Chapter 4 - Log Formatters

Log messages are simply strings. If you want to include additional information, you must format the message. But what does that mean anyway?

When you look at logs, you usually try to determine what went right and wrong with your application. To do that, you need more than just a message.

Common examples of additional useful information include the following:

- A timestamp
- A line number
- The log level
- The filename
- The module name

You could include other bits of information as well. You need to find a balance between how verbose you want your logs to be and how pithy you want them to be. You or your team will be reading them, so you want them to be informative.

In this chapter, you will cover these topics:

- What is a logging formatter?
- LogRecords
- LogRecord attributes
- Formatting your logs
- Formatting the date and time
- Using parameters

Soon you'll be formatting your logs like a pro.

Let's get started!

What is a Logging Formatter?

When it comes to creating logs in Python, you need three items:

- The logger object
- The handler object
- The formatter object

In the previous chapter, you learned about the logger and handler objects. Now, it's time to learn how to create a formatter object.

If you look at the Python documentation, you will see the following definition of a **Formatter** object:

```
logging.Formatter(fmt=None,
                  datefmt=None,
                  style='%',
                  validate=True, *,
                  defaults=None)
```

When you create a Formatter object, most examples will only show the `fmt` parameter getting used. You'll probably never even need to mess with the rest of those parameters.

However, it's good to know what they are for. So here is a brief definition of each of them from the Python documentation:

- fmt - A format string using the given *style* param for the logged output
- datefmt - A format string using the given *style* for the date/time portion of the logged output
- style - Uses one of `'%'`, `'{'` or `'$'` which determines how the format string will be merged; these formatters are then used by printf-style String Formatting (%), str.format() ({) or string.Template ($).
- validate - When `True` (default), an incorrect `fmt` and `style` will cause a `ValueError` to be raised
- defaults - A dict[str, Any] of default values that you will use in custom fields

As mentioned, other than `fmt`, you probably won't use any of the other parameters listed above except, perhaps `datefmt`.

Let's look at an example of using a logging Formatter now. Open up your favorite Python IDE or text editor and create a new file named `hello_formatter.py`.

Then enter the following code into your new file:

```
# hello_formatter.py

import logging

# Create custom logger
logger = logging.getLogger(name="test")
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("test_formatter.log")

# add handler to logger
logger.addHandler(file_handler)

# add formatter
formatter = logging.Formatter()
file_handler.setFormatter(formatter)

logger.debug("Hello debug")
logger.info("Hello info")
```

The code above looks familiar. That's because most of it is from the last example you saw in chapter 3! The code you want to focus on is in the last three lines before you write the log messages.

Here's the new code all by itself:

```
# add formatter
formatter = logging.Formatter()
file_handler.setFormatter(formatter)
```

To add a Formatter to your logger, you add the formatter object to your handler object. But why not simply add it directly to the logger? You may want to have different formats for different handlers.

For example, you might want to add a very verbose formatter to log to a file and a lighter, easier-to-read formatter to stream out to the console. Logs that scroll by quickly are hard to read, and if they are jam-packed with information, they will be even harder to read.

When you run this code, you'll get the following output:

```
Hello debug
Hello info
```

Oops! That log isn't formatted after all! Your **Formatter** object doesn't have any formatting instructions applied to it. You left it to use the defaults, which means your logger will only emit the log message and nothing else.

Let's copy the code above into a new file named `log_formatting.py` and then update the **Formatter** object with some formatting instructions:

```
# log_formatting.py

import logging

# Create custom logger
logger = logging.getLogger(name="test")
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("formatted.log")

# add handler to logger
logger.addHandler(file_handler)

# add formatter
formatter = logging.Formatter(
    ("%(asctime)s - %(name)s - %(levelname)s - "
     "%(message)s"))
file_handler.setFormatter(formatter)

logger.debug("Hello debug")
logger.info("Hello info")
```

Here, you call `Formatter` with the string `'%(asctime)s–%(name)s–%(levelname)s–%(message)s'`, which is a very special logging format. If you dig through the documentation or the source code, you will find that the `logging` module creates `LogRecord` objects when it does log formatting.

These `LogRecord` objects have many different parameters, including `asctime`, `name`, `levelname`, and `message`.

Rather than defining those right this second, try running the code and examining the output.

You should see something like this:

```
2024-02-07 08:29:44,734 - test - DEBUG - Hello debug
2024-02-07 08:29:44,734 - test - INFO - Hello info
```

Can you deduce what those parameters are doing now? If not, have no fear! You'll dive right into `LogRecords` in the next section and their attributes soon after that!

LogRecords

The logger object automatically creates LogRecord instances for you every time a new string is logged. Technically, you can also manually create LogRecord objects via the `makeLogRecord()` method call, which is very rare.

Here is the `LogRecord` class signature:

```
class logging.LogRecord(
     name,
     level,
     pathname,
     lineno,
     msg,
     args,
     exc_info,
     func=None,
     sinfo=None)
```

While you won't be creating log records manually in this book, it is good to understand how to do so. Remember, the logger object automatically creates a log record when something is logged.

The logger will instantiate the `LogRecord` using one or more of these parameters:

- name (str) - The name of the logger object
- level (int) - The numeric value of the log level (i.e. 10 for DEBUG, 20 for INFO, etc)
- pathname (str) - The full path to the source file where the logging call originates
- lineno (int) - The line number in the source file
- msg (Any) - The event description message
- args (tuple or dict) - Variable data that will be merged into the `msg` argument to create the event's description
- exc_info (tuple[type[BaseException], BaseException, types.TracebackType] | None) - The exception tuple that contains the current exception information, if any, from sys.exc_info() or None
- func (str or None) - The name of the function or method that the logging call comes from
- sinfo (str or None) - A text string that represents stack information

If you'd like to learn more about `LogRecords`, please refer to the Python documentation on the subject.

Now you're ready to learn about the formatting you can control directly!

LogRecord Attributes

While you usually won't create the `LogRecord` objects yourself, you will set their attributes via the formatting string that you pass to your **Formatter** object. You can set quite a few different attributes.

Here's a listing of most of them:

- asctime - `%(asctime)s` - Human-readable creation time. By default, it follows this form: "2024-08-08 16:55:45,777"
- created - `%(created)f` - The creation time of the log record as returned from `time.time()`

- filename - `%(filename)s` - The filename part of the `pathname`
- funcName - `%(funcName)s` - The name of the function that contains the logging call
- levelname - `%(levelname)s` - The text logging level of the message (‘DEBUG’, ‘INFO’, ‘WARNING’, ‘ERROR’, ‘CRITICAL’)
- levelno - `%(levelno)s` - The numeric logging level for the message
- lineno - `%(lineno)d` - The source’s line number where the logging call came from
- message - `%(message)s` - The logged message, which is set when `Formatter.format()` is called
- module - `%(module)s` - The module name (i.e. the name portion of `filename`)
- msecs - `%(msecs)d` - The millisecond portion of the time when the log record is created
- name - `%(name)s` - The name of the logger used to log the message
- pathname - `%(pathname)s` - The full path of the source file where the logging came from (if available)
- process - `%(process)d` - The process ID (if available)
- processName - `%(processName)s` - The process name (if available)
- relativeCreated - `%(relativeCreated)d` - The time in milliseconds when the log was created, relative to when the logging module was loaded
- thread - `%(thread)d` - The thread ID (if available)
- threadName - `%(threadName)s` - The thread’s name (if available)
- taskName - `%(taskName)s` - The `asyncio.Task` name (if available)

These attributes are what you use to format your log messages. In an earlier section in this chapter, you saw the following line of code:

```
1 formatter = logging.Formatter(
2     ("%(asctime)s - %(name)s - %(levelname)s - "
3      "%(message)s"))
```

Here, you are creating a `Formatter` object that will record a human-readable timestamp of when the log was created, the name of the logger that

emitted the log message, the name of the log level, and the log message itself.

When you ran that code, the log will contain messages like the following:

```
2024-02-07 08:29:44,734 - test - DEBUG - Hello debug
2024-02-07 08:29:44,734 - test - INFO - Hello info
```

The first twenty-three characters represent that human-readable timestamp. The timestamp is separated from the next piece of information by a couple of spaces and a dash. Then you get the logger's name, "test". If you go back to the original code, you can see the logger name getting set like this:

```
logger = logging.getLogger(name="test")
```

The third piece of information in the log message is your log level. These two lines show you logged a DEBUG and an INFO level message. Finally, you can see the log message at the end of each line.

If you want to learn more, refer to the Python documentation, which is about `LogRecord` attributes.

Let's move on and see a couple more examples of formatting your logs!

Formatting Your Logs

The best way to learn how to do something well is through practice. In this section, you will try out several variations of the log record attribute string to format your log messages.

Open up your Python IDE or text editor and create a new file called `more_log_formatting.py`. Then enter the following code:

```
# more_log_formatting.py

import logging

# Create custom logger
logger = logging.getLogger(name="formatting")
```

```
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("formatting.log")

# add handler to logger
logger.addHandler(file_handler)

# add formatter
fmt = ("%(asctime)s - %(filename)s - %(lineno)d "
       "- %(message)s")
formatter = logging.Formatter(fmt)
file_handler.setFormatter(formatter)

logger.debug("Hello debug")
logger.info("Hello info")
```

This code is virtually the same as the previous examples. What you want to focus on is the formatter string which contains the log record attributes:

```
fmt = ("%(asctime)s - %(filename)s - %(lineno)d "
       "- %(message)s")
```

This formatter string has two new log record attributes in it:

- %(filename)s
- %(lineno)d

These two log record attributes will log out the file name of the module from which the log message is coming, as well as the line number from which the log is sent.

When you run this code and open the log file (`formatting.log`), you will see something like the following:

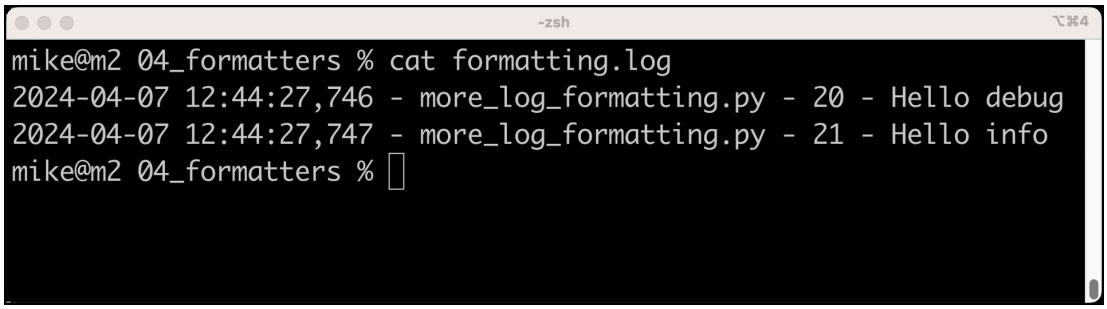


Figure 1. Log output

Here, you can see which file creates the log and which line emits the log messages. You could format the string better by adding explanatory text to it so that you know what those numbers are.

You could change it like this, for example:

```
fmt = ("%(asctime)s - %(filename)s - line number: "
       "%(lineno)d - %(message)s")
```

Now, you are ready to try changing a couple more of your log record attributes. Try changing your code's `fmt` variable to the following:

```
fmt = ("%(asctime)s - %(pathname)s - %(module)s - "
       "%(message)s")
```

In this example, you swapped out the two new log record attributes for the following:

- %(pathname)s
- %(module)s

The pathname log attribute is the fully qualified path to the Python script that creates the log file. The module is the name portion of the `filename` log attribute, which means it's the file name without the file extension.

Try running the code again and then open up the log file. You should see some new entries that look something like this:

```
2024-05-07 09:57:15,884 - C:\books\more_log_formatting.py\
 - more_log_formatting - Hello debug
2024-05-07 09:57:15,884 - C:\books\more_log_formatting.py\
 - more_log_formatting - Hello info
```

Here, you can see the fully qualified path to the Python script and the module name. Good job!

You can try out some of those other `LogRecord` attributes. Try adding some of them to your code or swapping out some of the ones in the example for

others in the list above. Then, re-run the code to see how the output changes.

When you are done, you can move on to the next section, where you will learn how to change the date and time portion of the log message.

Formatting the Date and Time

Your organization may want to use a date and time format other than the default one provided by Python's logging module. Fortunately, the `Formatter` class lets you change the datetime formatting via the `datefmt` parameter.

The `datefmt` parameter uses the date-time format codes that Python's `datetime` module uses. If you need a refresher on how those work, check out the Python documentation.

Open up your Python editor and create a new file named `date_formatting.py`. Then enter the following code:

```
# date_formatting.py

import logging

# Create custom logger
logger = logging.getLogger(name="datefmt")
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("datefmt.log")

# add handler to logger
logger.addHandler(file_handler)

# add formatter
datefmt = "%a %d %b %Y"
formatter = logging.Formatter(
    ("%(asctime)s - %(name)s - %(levelname)s - "
     "%(message)s"),
    datefmt=datefmt)
file_handler.setFormatter(formatter)

logger.debug("Hello debug")
logger.info("Hello info")
```

The changes you want to focus on here are contained in the following two lines of code:

```
datefmt = "%a %d %b %Y"
formatter = logging.Formatter(
    ("%(asctime)s - %(name)s - %(levelname)s - "
     "%(message)s"),
    datefmt=datefmt)
```

Yes, the date format codes are weird. Here's the definition of what that first line means:

- %a - The weekday's abbreviated name
- %d - The day of the month as a zero-padded decimal number
- %b - The month's abbreviated name
- %Y - The year including the century as a decimal number

What does that look like though? Well, run the code and open up the `datefmt.log` file and you will see something like this:

```
Fri 09 Feb 2024 - datefmt - DEBUG - Hello debug
Fri 09 Feb 2024 - datefmt - INFO - Hello info
```

Ta-da! Isn't that cool?

Now let's try a different `datefmt` string. Here's an example one you can try:

```
datefmt = "%d/%m/%y %H:%M:%S"
```

This one will print out the day/month/year and the hour, minute, and second. Swap that line in and re-run your code.

When you do so, you're new output will look similar to the following:

```
09/02/24 13:28:02 - datefmt - DEBUG - Hello debug
09/02/24 13:28:02 - datefmt - INFO - Hello info
```

Nice! If none of these formats works for you, be sure to read up on the datetime format codes and try creating your own formatting with them.

Using Parameters

What are logging parameters? When you create a log message, you sometimes want to pass in parameters. There are several methods for formatting log messages. The recommended way is to use printf-style string substitution, which comes from C and C++.

Here is an example of using the printf-style string substitution using a Python shell or REPL session:

```
>>> name = "Mike"
>>> print("Nice to meet you, %s" % name)
Nice to meet you, Mike
```

To do string substitution in this way, you need to insert a “%s” into your string to tell Python that you want to insert a string there. Then, you need to use a percent sign outside of your Python string followed by the variable name you want to insert.

The `logging` module is similar, but it uses a shorter version of this format. Open up your Python IDE and create a new file named `using_parameters.py`.

Then enter the following code:

```
# using_parameters.py

import logging

# Create custom logger
logger = logging.getLogger(name="test")
logger.setLevel(logging.DEBUG)

file_handler = logging.FileHandler("params.log")

# add handler to logger
logger.addHandler(file_handler)

# add formatter
formatter = logging.Formatter()
file_handler.setFormatter(formatter)

```

```
18 name = "Mike"
19 logger.debug("Nice to meet you, %s", name)
```

The only part you need to focus on are the last two lines. You do not need the extra percent sign between the string and the parameter(s) you are inserting.

But wait!? Why not use Python's f-strings here instead? There are security concerns about using those. The long and short of it is that you can insert a dictionary of arbitrary size into an f-string. If you allow f-strings, they may be vulnerable to someone passing in a gigantic string that causes a Denial-of-Service type attack.

There are some advocates for using PEP675, which has a `LiteralString` type in it that may overcome this issue. However, this solution has not been accepted at the time of writing. For now, use the string substitution method above as much as possible.

Wrapping Up

Learning how to format your logs is fundamental. You want your logs to contain enough information to track down bugs but small enough to be easy to read. Finding that balance is key.

In this chapter, you learned about these topics:

- What is a logging formatter?
- LogRecords
- LogRecord attributes
- Formatting your logs
- Formatting the date and time
- Using parameters

The next step to take is to practice the concepts in this chapter and the ones you've learned about in the previous chapters. Soon you'll have some nice logs you can search through and learn from.

Chapter 5 - Log Configuration

The Python `logging` module provides three primary methods for configuring your logs. But what is a log configuration anyway?

The log configuration tells Python's logger the following:

- Where you want the log to (the handlers)
- How you want to format the log messages (the formatters)
- What level of log you want (DEBUG, INFO, CRITICAL, etc)
- and sometimes a bit more

In this chapter, you will learn about the three different methods for configuring your logs in Python.

Here is a quick preview:

- Configuring using `logging` API
- Configuring using `fileConfig`
- Configuring using `dictConfig`

These different configuration methods give you a lot of flexibility. Use the one that makes the most sense and makes your logging easier. Choosing that method depends greatly on your previous experience and the type of project you are working on.

Don't worry if you don't pick the right method the first time. It's normal to make mistakes and correct them when learning something new. In this case, there isn't a "right" answer anyway, as this is all about making your code easier for you or your team.

Let's get started!

Configuring Using the `logging` API

Configuring your logs with code uses Python's `logging` methods and attributes. You have already been using this method in the previous couple of chapters. But let's review how configuring using `logging` methods works again.

You should review the following two configuration options because they use much of the same terminology as using the `logging` module directly. If you have a good understanding of doing it by hand, in code, then the other two methods will be easier to pick up, too.

Open up your favorite editor and create a new file called `code_config.py`. Then enter the following code:

```
# code_config.py

import logging
import time


def main():
    """
    The main entry point of the application
    """
    logger = logging.getLogger("example_app")
    logger.setLevel(logging.INFO)

    # create the logging file handler
    file_handler = logging.FileHandler("example.log")
    formatter = logging.Formatter(
        ("%(asctime)s - %(name)s - %(levelname)s - "
         "%(message)s"))
    file_handler.setFormatter(formatter)

    # create the stream handler
    stream_handler = logging.StreamHandler()
    stream_formatter = logging.Formatter(
        ("%(asctime)s - %(filename)s - %(lineno)d - "
         "%(message)s"))
    stream_handler.setFormatter(stream_formatter)

    # add handlers to logger object
    logger.addHandler(file_handler)
    logger.addHandler(stream_handler)

    logger.info("Program started")

```

```
34      # Pretend to do some work
35      time.sleep(2)
36
37      logger.info("Done!")
38
39
40  if __name__ == "__main__":
41      main()
```

This example is similar to some of the others in this book. The primary difference is that you put all the logger configuration code into a function called `main()`.

In production-ready code, you would probably put your logging code into its module and perhaps in a function called `create_logger()` or `get_logger()`. But no matter what you call it, you want to make the code more reusable.

Take a look at the example above. Here you have a logger with the following characteristics:

- Name: `example_app`
- Logging level: INFO
- Two handlers: file and stream
- Two separate formatters, one per handler

The last couple of lines exercise your logger and include a call to `time.sleep()` to emulate your application executing a long-running process.

When you run this code, you should see something similar to the following output to your console:

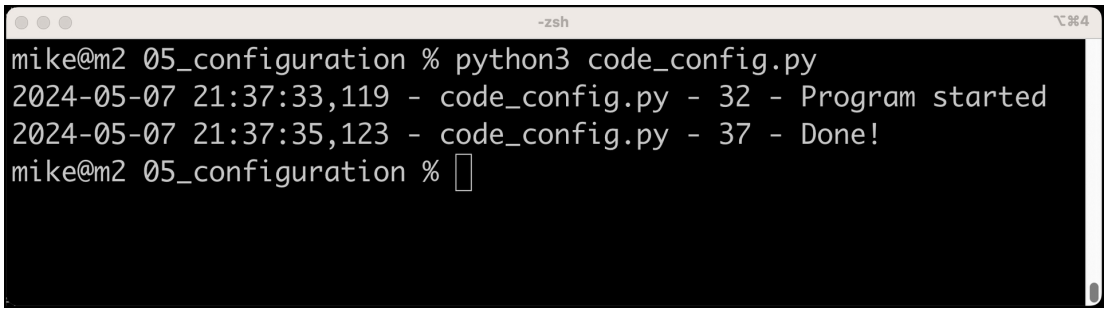


Figure 2. Console output

Your log file will look slightly different:

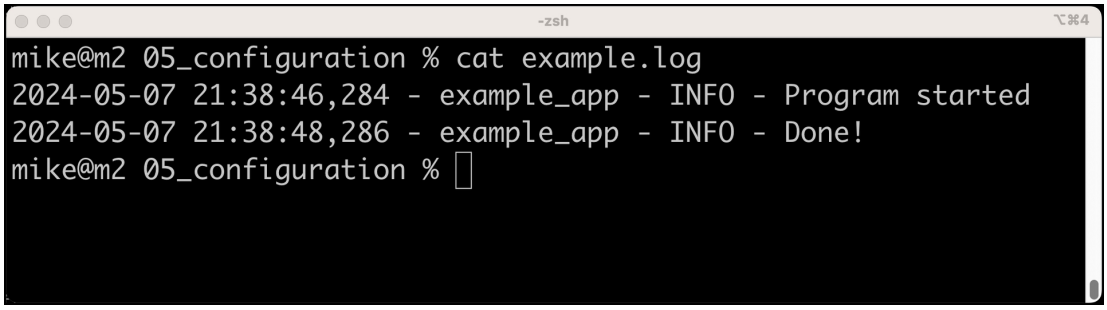


Figure 3. Log file output

The reason is that you are using separate formatters for your two handlers.

Now that you have reviewed how to configure your logs using code, you can learn how to configure using a file!

Configuring Using `fileConfig()`

Your mission is to recreate the configuration you did with the code in this chapter using a file configuration. But how does that work anyway? The configuration file for Python's `logging` module is quite similar to a Windows INI configuration file. The file format that `fileConfig` supports comes from Python's configparser module, which itself follows a format similar to Microsoft Windows INI files.

You create a section in the configuration like this: `[loggers]`. The square braces demarcate the beginning and end of the section's name. The text inside the square braces is the section name.

After creating a section, you can create a series of key and value pairs, with the key on the left, separated by an equals sign, and the value on the right. Here's an example: `level=CRITICAL`. Each key/value pair is on a separate line.

The sections' names are important, but you'll need to look at an example to see why. Go ahead and open your favorite text editor or IDE and create a new file named `logging.conf`. The file extension doesn't really matter, but `.conf` is a good choice as it indicates that you are creating a config file. You could also use `.ini`, as that is a very common configuration file extension, too.

Now that you have the file created, enter the following text into it:

```
[loggers]
keys=root,example_app

[handlers]
keys=fileHandler, consoleHandler

[formatters]
keys=file_formatter, stream_formatter

[logger_root]
level=CRITICAL
handlers=consoleHandler

[logger_example_app]
level=INFO
handlers=fileHandler
qualname=example_app

[handler_consoleHandler]
class=StreamHandler
formatter=stream_formatter
args=(sys.stdout,)

[handler_fileHandler]
class=FileHandler
level=DEBUG
formatter=file_formatter
args=("config.log",)

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

```
33 datefmt=
34
35 [formatter_stream_formatter]
36 format=%(asctime)s - %(filename)s - %(lineno)s - %(messag\
37 e)s
38 datefmt=%a %d %b %Y
```

The rest of the configuration file is derived from the first three sections. So take a look at these first:

- loggers
- handlers
- formatters

Each of these first three sections contains a list of comma-separated `keys`. These keys tell Python which logger, handler, and formatter objects are available.

The following sections are named using the section name, underscore, and key name. Here's an example: `logger_root`

Note that it's not named `loggers_root` (i.e. plural), but `logger_root`. When you set up your logger, tell it what logger level you want to use and which handler to emit log output to. In the case of `logger_root`, you set the level to CRITICAL and the handler to the console. The `logger_root` section is REQUIRED!

But what is a root logger, anyway? The root is the default or lowest-level logger, and it is required when using a configuration file. If you want to, you could use a NullHandler for your root to prevent it from outputting anything, at least temporarily.

The individual **handler** configuration sections (ex. `handler_consoleHandler`) allow you to set the following:

- class - The type of handler to use
- level - The log level you want to use
- formatter- Which formatter to use

- args - The arguments to pass to that formatter object

Note that the `args` value here is where you pass in the path to the log file to which you want to write in the case of your **FileHandler** object.

Finally, the formatter sections give you the ability to format the formatters, using values similar to the arguments you passed to the formatter class:

- format - The Log Record formatting string
- datefmt - The datetime formatting used to format datetime objects

In the example above, you configure two handlers: a FileHandler and a StreamHandler. You also configure two formatter objects, one for the FileHandler and the other for the StreamHandler. The formats and the date formats for each are slightly different too.

Now that you have finished the configuration, open your Python IDE back up and create a Python file named `log_with_config.py`. Then enter the following code in it:

```
# log_with_config.py

import logging
import logging.config
import time


def main():
    logging.config.fileConfig("logging.conf")
    logger = logging.getLogger("example_app")

    logger.info("Program started")
    time.sleep(3)
    logger.info("Done!")


if __name__ == "__main__":
    main()
```

Your code is considerably smaller now that you use a file configuration for your logs. All that boilerplate is now abstracted away. You could do the same by making a custom module where you do all your logging setup

using Python's `logging` API, but using a configuration file is a nice alternative.

Loading a file-based logging configuration is via the `logging.config` submodule, which has a `fileConfig()` method. You then pass in the path to your logging config file to the `fileConfig()` method. This loads the configuration into memory, and you can then get the logger objects using `logging.getLogger()`.

When you run this code, you will see something like the following printed to stdout (i.e. your console):

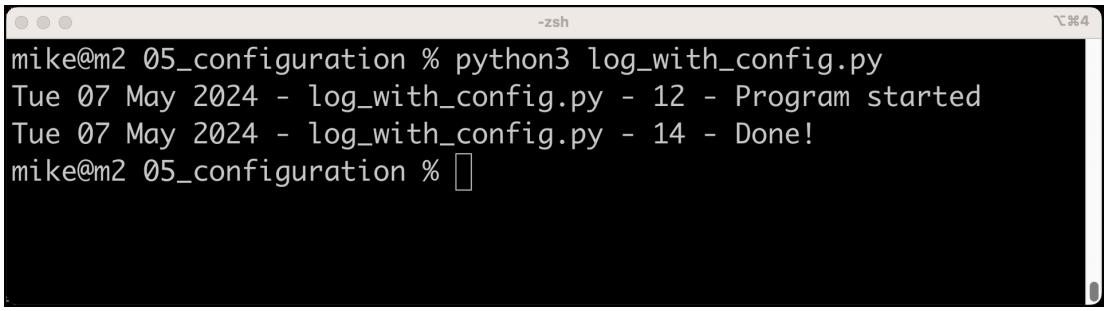


Figure 4. File config console output

If you open up your copy of `config.log`, you will see something similar to the following inside of it:

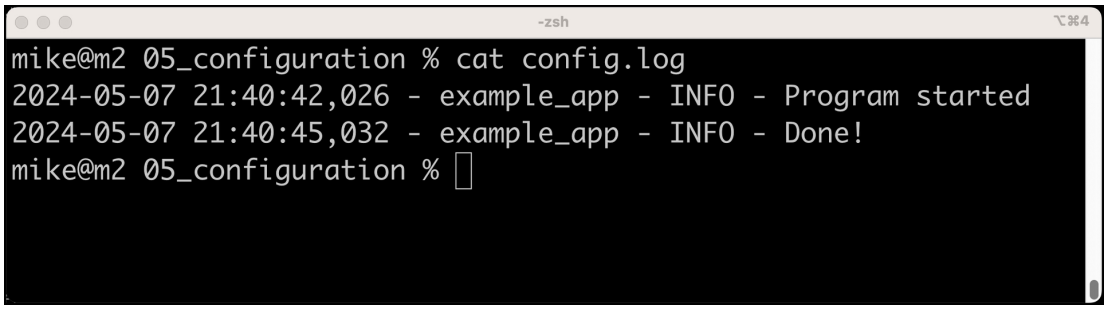


Figure 5. File config log file output

Take note that the datetime format is different in each of these. The console log uses a custom datetime format, while the file log uses the default datatime format. You are also logging out different log record attributes in each, except for the last part, the log message.

Start experimenting with your copy of the configuration file. Change it up. Add some new handlers or loggers. Then test it out and see how your logs change!

Now you're ready to move on and learn about the last way to configure your logs.

Configuring Using `dictConfig()`

The final type of logging configuration you can apply is via the `dictConfig()` method, another method from the `logging.config` submodule you learned about in the previous section.

As the name implies, you will create a Python dictionary to hold your logging configuration rather than using a file or the logging API itself.

You can start by opening up your Python IDE or favorite text editor. Create a new file named `log_with_dict_config.py` and add the following code to it:

```
# log_with_dict_config.py

import logging
import logging.config
import time


def main():
    log_config_dict = {
        "version": 1,
        "loggers": {
            "example_app": {
                "handlers": ["fileHandler", "consoleHandl\
er"],
                "level": "INFO",
            },
        },
        "handlers": {
            "fileHandler": {
                "class": "logging.FileHandler",
                "formatter": "file_formatter",
                "filename": "settings.log",
            },
```

```
            "consoleHandler": {
                "class": "logging.StreamHandler",
                "formatter": "stream_formatter",
            },
        },
        "formatters": {
            "file_formatter": {
                "format": ("%(asctime)s - %(name)s - %(le\
velname)s"
                " - %(message)s"),
            },
            "stream_formatter": {
                "format": ("%(asctime)s - %(filename)s - \
%(lineno)s"
                " - %(message)s"),
                "datefmt": "%a %d %b %Y",
            },
        }
    logging.config.dictConfig(log_config_dict)
    logger = logging.getLogger("example_app")

    logger.info("Program started")
    time.sleep(3)
    logger.info("Done!")


if __name__ == "__main__":
    main()
```

You must focus on the `log_config_dict`, which defines your log's configuration. There are four top-level keys:

- version - The version of your log dict
- loggers - The logger objects
- handlers - Which log handlers you want to use
- formatters - The formatter objects to apply to your handlers

You don't use the `version` explicitly in this example, but `logger.config` requires this key and will raise a `ValueError` if you don't include it. The version must also be set to one or a `ValueError` will be raised.

You `loggers` key specifies which logger objects you support. You do not need to specify a `root` logger when using a `dictConfig`. If you didn't set up

a `loggers` key though, you will not log anything, so be sure to set that up AND also call `getLogger()` to get your logger object.

The `handlers` key is where you specify which handlers you want to support for your logger. In this case, you recreate the `FileHandler` and the `StreamHandler` you had in your `fileConfig`. For the `FileHandler`, you have a nested dictionary where you can specify which formatter you want to apply and what file to save the logs to. Add a separate formatter for the `StreamHandler` to keep things interesting, as you did in the previous example.

The `formatters` key has a nested dictionary where you define one or more formatters. Create a `format` key and a Log Record formatted string to add a format. You may also add a `datefmt` key here where you can use the datetime formatting string to change how the timestamps are recorded in your logs.

To use the dictionary config, you must pass the dictionary object to `logging.config.dictConfig()`. Then, it would be best to use `logging.getLogger()` and pass it the logger name you created in your dictionary.

When you run this code, the output to the console and the file will be in the same formats as in the `fileConfig` section.

The `dictConfig` format is newer than the `fileConfig` format. Python's documentation notes that future enhancements to the `logging` module's configuration functionality will be added to `dictConfig`, so that is the recommended format if you want to use the latest logging features.

Best practice is to move the dictionary config to a file named `settings.py` or `conf.py`. Here is an example:

```
1 # settings.py
2
3 import logging
4 import logging.config
5
```

```
log_config_dict = {
    "version": 1,
    "loggers": {
        "example_app": {
            "handlers": ["fileHandler", "consoleHandler"],
            "level": "INFO",
        },
    },
    "handlers": {
        "fileHandler": {
            "class": "logging.FileHandler",
            "formatter": "file_formatter",
            "filename": "settings.log",
        },
        "consoleHandler": {
            "class": "logging.StreamHandler",
            "formatter": "stream_formatter",
        },
    },
    "formatters": {
        "file_formatter": {
            "format": ("%(asctime)s - %(name)s - %(leveln\
ame)s"
            " - %(message)s"),
        },
        "stream_formatter": {
            "format": ("%(asctime)s - %(filename)s - %(li\
neno)s"
            " - %(message)s"),
            "datefmt": "%a %d %b %Y",
        },
    },
}
logging.config.dictConfig(log_config_dict)
```

Then you could update your main code to the following:

```
# log_with_settings.py

import logging
import time

import settings


def main():
    logger = logging.getLogger("example_app")

    logger.info("Program started")
    time.sleep(3)
```

```
14      logger.info("Done!")
15
16
17  if __name__ == "__main__":
18      main()
```

You can see that the main code is now much shorter. You configure the logger when you import `settings`, and you can access it using `logging.getLogger("example_app")`. At that point, everything will work the same way as before.

Wrapping Up

Python's `logging` module provides flexibility in configuring your logs. You should be familiar with all three methods you can use.

In this chapter, you learned about the following methods:

- Configuring using `logging` API
- Configuring using `fileConfig`
- Configuring using `dictConfig`

The `logging` module's API is the most well-known. You will find many examples of that in tutorials all over the Internet. In many ways, the logging API is the most flexible. Many developers will create their custom logging modules and classes to get the most flexibility out of their logs.

However, the logging API is not the only way to configure your Python application's logs. You can also use `fileConfig` or `dictConfig`. If you choose between those two, you should use the `dictConfig` as it will receive the most support in the future. If you want to keep your code modular, you can place the dictionary in its own Python module and import it.

Start experimenting with these three different methods of configuring your logs and see which one you like best!

Chapter 6 - Log Filters

Python's `logging` module provides support for filters. You can add a Filter to a `Handler` or a `Logger` object for more sophisticated filtering than you can get through log levels.

Python's documentation gives the following example of how you might use a logging filter:

> A filter initialized with 'A.B' will allow events logged by loggers 'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' etc. but not 'A.BB', 'B.A.B' etc. If initialized with the empty string, all events are passed.

A handler object with a Filter object applied to it will filter before the handler emits the event. If you apply the filter to the logger, then the filter applies before the event is sent to the handler. Why does that matter? Well, if you have descendant logger objects, they will not be filtered unless you add the filter to the descendants too.

Here is what you will learn in this chapter:

- Creating a filter
- Applying the filter to a logger
- Applying the filter to a handler

Let's get started!

Creating a filter

The first step you must understand is how to create a logging filter. There are two primary methods:

- Subclassing from `logging.Filter`
- Creating a class with a `filter()` method

You will look at an example of both of these methods.

To start, open up your Python IDE or a text editor and create a new file named `logging_filter.py`. Then enter the following code:

```
# logging_filter.py

import logging

class MyFilter(logging.Filter):
    def filter(self, record):
        if record.funcName.lower().startswith("a"):
            return False
        return True
```

Here is an example of subclassing `logging.Filter`. Then, you create a `filter()` method that takes in a Log Record instance. At that point, you have a Log Record object that you can use to get more information about it.

For this example, you check to see if the function that the Log Record emits has a name that begins with “a”. If it does, you return `False`. Otherwise, you return `True`. But what does this mysterious code do?

When you return `False`, it tells the `logging` object (or the handler) that you are filtering out any function or method whose name starts with “a” and is attempting to log. For whatever reason, you don’t need those logs right now, so they are filtered out. When you return `True`, those log records will get logged.

Now you’re ready to look at an example where you create your class without subclassing `logging.Filter`.

Create a new file named `logging_filter_no_subclass.py` and enter the following code:

```
# logging_filter_no_subclass.py

class MyFilter():
    def filter(self, record):
        if record.funcName.lower().startswith("a"):
```

```
            return False
        return True
```

Wait a minute! That code's the same as the last example!!! Well, not quite. This time you're not subclassing anything. That's the difference. Otherwise, you're right. It is the same!

Starting in **Python 3.12**, there is a third way to create a filter. You can simply use a function or other callable as a filter. The `logging` module will check if the filter object has a `filter` attribute. If it does, then that will be called. Otherwise, the `logging` module will assume it's a callable and just pass in the log record as a single parameter.

Here is an example function that does the same thing as the class above:

```
def filter(record: logging.LogRecord):
    if record.funcName.lower().startswith("a"):
        return False
    return True
```

Of course, creating a filter class or function is fun, but you probably want to see this code in action. Have no fear—that's what you will learn next.

Applying the Filter to a Logger

The' addFilter() ' method makes adding a filter to a logger object easy. You'll soon see how easy it is to take the filter from the previous section and apply it to a logger object.

Open up your IDE again and create a new file named `logging_filter.py`. Then enter the following code:

```
# logging_filter.py

import logging
import sys


class MyFilter(logging.Filter):
    def filter(self, record):
```

```
 9          if record.funcName == "a":
10              return False
11          return True
12
13
14  def a():
15      """
16      Ignore this function's log messages
17      """
18      logger.debug("Message from function a")
19
20
21  def b():
22      logger.debug("Message from B")
23
24
25  if __name__ == "__main__":
26      logging.basicConfig(stream=sys.stderr,
27                          level=logging.DEBUG)
28      logger = logging.getLogger("filter_test")
29      logger.addFilter(MyFilter())
30      a()
31      b()
```

The first third of this code contains some imports and your filter, which is called `MyFilter` and subclasses `logging.Filter`. You follow this up with two functions, `a()` and `b()`. All these functions log a message using the debug log level.

The last block of code creates a logger object. You set the logger to output to stderr, which is the same as creating a `StreamHandler`. Then you add the filter to your logger object like this: `logger.addFilter(MyFilter())`.

Pretty easy, right? When you run this code, you will see the following output in your terminal:

```
1  DEBUG:filter_test:Message from B
```

Your filter was a success! You only logged from function `b()` and the logs from `a()` were filtered out.

Applying the Filter to a Handler

Applying a filter to a handler is similar to adding one to a logger. For this one, you will create a filter that adds two new attributes to the Log Record object using monkey patching to provide contextual information about the logs.

The following example is based on one from Python's logging cookbook.

Open up your Python IDE or text editor and create a new file named `context_filter.py`. Then enter the following code:

```
# context_filter.py

import logging
from random import choice


class ContextFilter:
    USERS = ["Mike", "Stephen", "Rodrigo"]
    LANGUAGES = ["Python", "PHP", "Ruby", "Java", "C++"]

    def filter(self, record):
        record.user = choice(ContextFilter.USERS)
        record.language = choice(ContextFilter.LANGUAGES)
        return True


if __name__ == "__main__":
    levels = (
        logging.DEBUG,
        logging.INFO,
        logging.WARNING,
        logging.ERROR,
        logging.CRITICAL,
    )
    logger = logging.getLogger(name="test")
    logger.setLevel(logging.DEBUG)

    handler = logging.StreamHandler()
    my_filter = ContextFilter()
    handler.addFilter(my_filter)
    logger.addHandler(handler)

    fmt = ("%(name)s - %(levelname)-8s "
           "User: %(user)-8s Lang: %(language)-7s "
           "%(message)s")
    formatter = logging.Formatter(fmt)
    handler.setFormatter(formatter)

```

```
39      logger.info("This is an info message with %s",
40                  "silly parameters")
41      for _ in range(10):
42          level = choice(levels)
43          level_name = logging.getLevelName(level)
44          logger.log(level, "A message with %s level",
45                     level_name)
```

Your `filter()` method modifies the log record to add a `user` and `language` attribute. You choose a random user and language and set them. Then in the following code, you create your boilerplate:

- logger - a logger named “test”
- handler - a `StreamHandler` instance that emits to stdout
- my_filter - an instance of `ContextFilter`
- formatter - a formatter object where you insert your new logger attributes

Pay attention in this code. Here you are doing `handler.addFilter(my_filter)` instead of `logger.addFilter()`. If you were using multiple handlers like one of the previous chapters, the other handler would not have the filter applied to it.

The last few lines of your code log out ten lines at random log levels.

When you run this code, you will get something similar to this, albeit your results will be considerably different since the users and languages are randomly picked:

```
test - INFO     User: Rodrigo  Lang: C++     This is an i\
nfo message with silly parameters
test - CRITICAL User: Rodrigo  Lang: PHP     A message wi\
th CRITICAL level
test - CRITICAL User: Mike     Lang: Python  A message wi\
th CRITICAL level
test - WARNING  User: Mike     Lang: Python  A message wi\
th WARNING level
test - CRITICAL User: Stephen  Lang: Ruby    A message wi\
th CRITICAL level
test - ERROR    User: Mike     Lang: Ruby    A message wi\
th ERROR level
test - ERROR    User: Stephen  Lang: Python  A message wi\
```

```
14 th ERROR level
15 test - CRITICAL User: Stephen  Lang: C++     A message wi\
16 th CRITICAL level
17 test - INFO     User: Rodrigo  Lang: PHP     A message wi\
18 th INFO level
19 test - WARNING  User: Rodrigo  Lang: Ruby    A message wi\
20 th WARNING level
21 test - DEBUG    User: Rodrigo  Lang: Python  A message wi\
22 th DEBUG level
```

Hmmm. That output looks like some fields get set to a static size. How did that happen? Go back to the code and examine the formatting string. You will notice a “-8s” on some of the formatted objects (e.g., “%(levelname)-8s”). That tells the `Formatter` object to make that field eight characters wide. If the inserted string is less than eight characters, your code will pad it with spaces.

It looks good, right? Try changing the number of characters to other values and formatting the various fields as you see fit. Practicing that is a great way to cement what you’re learning.

Wrapping Up

Logging filters allow you to refine what ends up in your logs further. Most engineers rely on log levels to control what goes in their logs, but you can use filters to give you even more fine-grained control.

In this chapter, you learned about the following:

- Creating a filter
- Applying the filter to a logger
- Applying the filter to a handler

Remember that filters only apply to the loggers or handlers you attach them to. If you notice that some items are slipping through, you probably forgot to add the filter to a descendant logger object or to a handler.

As always, practice is the best way to learn to use something new. Try logging filters and see if they can help improve your logging!

Chapter 7 - Logging from Multiple Modules

As a beginner, you usually write your application code in a single file. But soon, you'll find that you have hundreds or thousands of lines of code in a file, which can become hard to manage.

Then, you break up your code into multiple files. In Python, each Python file is considered a module. A group of related modules is called a package.

Why does that matter? You'll need to add logging to your application, and most applications have multiple files. This chapter shows you how to add logging to a multi-module application.

To keep the code simpler, you will learn how to add logging using the following methods:

- Logging using the logging API
- Logging using `dictConfig`

You'll start your journey using the logging API method first!

Logging Using the Logging API

The logging API is where you use logging classes, methods, and attributes to configure and log data. In this section, you will create two modules to log data from.

Here are the names of the two files:

- `main.py` - The main entry point of your fake application
- `other_mod.py` - A second module to log from

Your `main.py` will set up your logger object and anything else you need. The `other_mod.py` file will only need to call `getLogger()` with the appropriate name to use the logger you configured in `main.py`.

While this sounds a bit complicated, you'll soon see it's fairly straightforward. Open up your favorite Python IDE or text editor and create the `main.py` file.

Then enter the following code:

```
# main.py

import logging
import other_mod


def main():
    """
    The main entry point of the application
    """
    logger = logging.getLogger(name="test")
    logger.setLevel(logging.DEBUG)
    file_handler = logging.FileHandler("multi.log")
    logger.addHandler(file_handler)
    formatter = logging.Formatter(
        "%(asctime)s - %(filename)s - %(levelname)s "
        "- %(message)s")
    file_handler.setFormatter(formatter)

    logger.info("Program started")
    result = other_mod.add(7, 8)
    logger.info("Done!")
    return result


if __name__ == "__main__":
    main()
```

Here, you create a logger named "test", give it a `FileHandler`, and apply a formatter. Then, you log an information message, call `other_mod.add()`, log another message, and return.

Pretty simple, right? You have seen 99% of this type of code already. The only new bit is calling another module. But that's where the logging to the same logger across multiple modules comes in.

To see how that works, go ahead and create the `other_mod.py` file in your Python IDE and add the following code to it:

```python
# other_mod.py

import logging


def add(x, y):
    logger = logging.getLogger(name="test")
    logger.info("added %s and %s to get %s", x, y, x + y)
    return x + y
```

Your code imports the `logging` module and then uses `logging.getLogger()` to get the "test" logger instance. Now you can log to the same file as you were logging to in `main.py`. You don't need to create a handler or a formatter or do any other setup in this module. That's already been taken care of.

When you run `main.py`, you will see something like the following log output in your `multi.log` file:

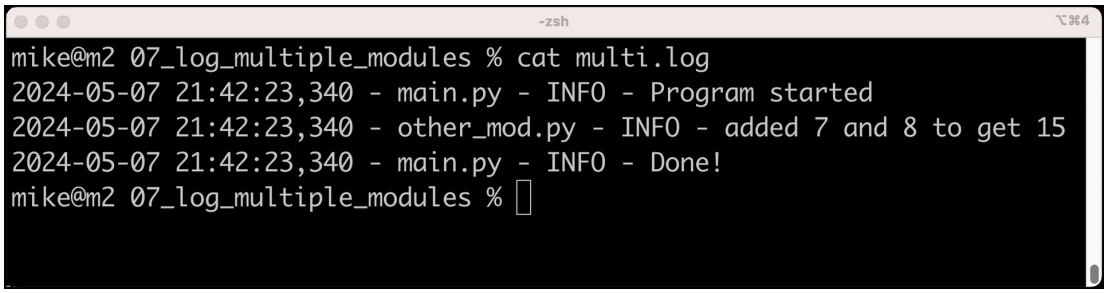


Figure 6. Multi.log output

You can look at these logs and tell when a log message is written to the log. You can also see which Python file the log messages come from. But what if you wanted to use a `dictConfig`? You'll learn how to do that next!

Logging Using `dictConfig`

For this example, you will create a fake Python package or application that you want to add logging to. Experimenting with logging in these varied scenarios will help you add logging to your own applications.

Here is an example tree view of a folder that you will need to create on your computer:

```
sample_package
|- __init__.py
|- main.py
|- settings.py
|- utils
    |- __init__.py
    |- minimath.py
```

The first step is to create a `sample_package` directory. Inside of that directory, you need the following items:

- `__init__.py`
- `main.py`
- `settings.py`
- `utils` directory

Then inside the `utils` sub-directory, you will need the following files:

- `__init__.py`
- `minimath.py`

When you dig into Python's `logging` documentation and guide, you may notice that it says you can use `__name__` for your logger object's name. Using the `__name__` object is a dynamic method of creating separate loggers for each module that you want to add logging to. Unlike the previous example, which only used the "test" logger, this one will create several logger objects.

Because you will have several loggers, your setup will be more complex. However, you will quickly see how powerful the `logging` module is and how granular you can be with it.

Start by re-opening your Python IDE or text editor and creating an empty `__init__.py` file and your new `main.py` file.

The empty `__init__.py` files may seem strange at first. However, these files tell Python that the folder they are in is now a part of a package. That

means you can now import the folder itself as if it were a Python module. You can do more with these files, but that is outside the scope of this book.

Then add the following code to `main.py`:

```
# main.py

import logging.config
import settings
from utils import minimath

logging.config.dictConfig(settings.LOG_CONFIG)


def main():
    log = logging.getLogger(__name__)
    log.debug("Logging is configured.")

    minimath.add(4, 5)

if __name__ == "__main__":
    main()
```

Here, you import the `settings` module and the `utils.minimath` module, which you still need to create. Then, you load up the `logging` config from the `settings` module. The `main()` function gets a logger called `__name__`.

This brings up the strange quirk in Python where you see `if __name__ == "__main__"` at the bottom of Python files. What this conditional is doing is checking to see if you ran the module directly or if you imported it. If you run the file directly, it's `__name__` equals `"__main__"`, otherwise it equals the actual name of the module.

You then log out a debug message and call `minimath.add()`. That's it!

You are now ready to learn how to configure your code to make this log work. In your Python IDE or text editor, create a `settings.py` file and save it in the same location as your `main.py` file from above. Then enter the following code into it:

```
# settings.py

```

```
LOG_CONFIG = {
    "version": 1,
    "loggers": {
        "": {  # root logger
            "handlers": ["default"],
            "level": "WARNING",
            "propagate": False,
        },
        "utils.minimath": {
            "handlers": ["fileHandler", "default"],
            "level": "DEBUG",
            "propagate": False,
        },
        "__main__": {  # if __name__ == '__main__'
            "handlers": ["default"],
            "level": "DEBUG",
            "propagate": False,
        },
    },
    "handlers": {
        "fileHandler": {
            "class": "logging.FileHandler",
            "formatter": "file_formatter",
            "filename": "settings.log",
        },
        "default": {
            "class": "logging.StreamHandler",
            "formatter": "stream_formatter",
        },
    },
    "formatters": {
        "file_formatter": {
            "format": "%(asctime)s - %(name)s - %(levelna\
me)s - %(message)s",
        },
        "stream_formatter": {
            "format": "%(asctime)s - %(name)s - %(filenam\
e)s - %(lineno)s - %(message)s",
            "datefmt": "%a %d %b %Y",
        },
    },
}
```

Your dictionary configuration defines three loggers:

- “” - An empty string which maps to the root logger
- `utils.minimath` - The name of the submoduke you create in the `utils` subfolder

- `"__main__"` - The logger to use if a module is executed directly

Your dictionary also defines two formatters:

- “fileHandler” - An instance of `logging.FileHandler` that writes to “settings.log”
- “default” - The default handler that writes to stdout via a `logging.StreamHandler`

Finally, you also configure two separate formatter objects, one for the file handler and one for the stream handler.

Next, you will need to create a `utils` directory in the same directory as your `main.py`, `settings.py`, and `__init__.py` files. Inside the `utils` folder, you must create an empty `__init__.py` file and the `minimath.py` file.

For the `minimath.py` file, you will need to add the following code:

```
# minimath.py

import logging

module_logger = logging.getLogger(__name__)


def add(x, y):
    module_logger.info("Adding %s and %s", x, y)
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x / y
```

Here, you once again create a logger using the `__name__` object and add a quick informational log message to the `add()` function.

Now you are ready to try running `main.py`. Give it a try. You should see the following output in your terminal or your IDE's console:

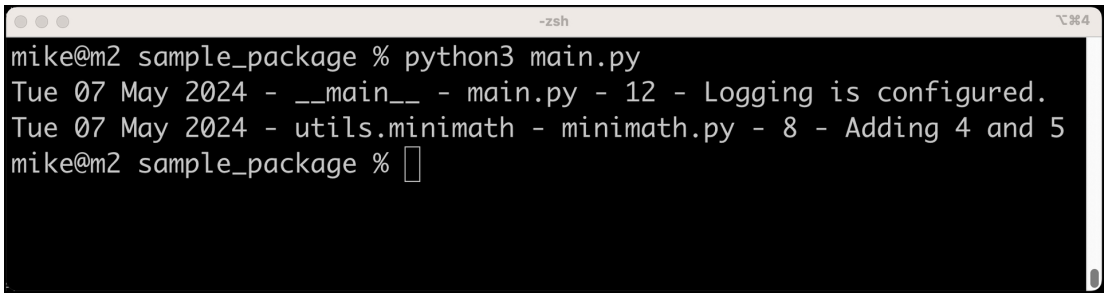


Figure 7. Main terminal output

Here, the name emitted from `main.py` is `__main__` while the name emitted from `minimath.py` is `utils.minimath`. The latter is a submodule. When you added that `__init__.py` script to the `utils` folder, it made the `utils` folder into a package name that you can import; this also allows you to import any Python files that `utils` contains.

If you open the `settings.log` file, you will see that it contains only a single log that is similar to the following:

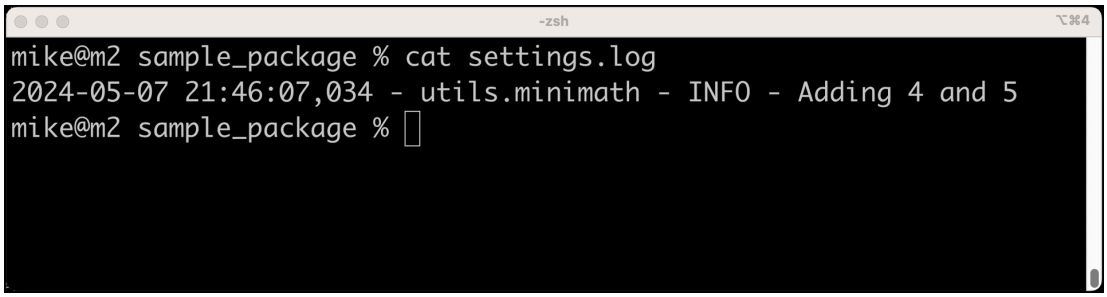


Figure 8. Settings.log output

You will need to study the configuration file closely. When you do, you will find that the "**main**" logger is connected to the `StreamHandler` only. The "utils.minimath" is attached to BOTH the `StreamHandler` and the `FileHandler`, which is why you see the same output for that module in both the console and in the file.

If you added the `if __name__ == "__main__"` line to your minimath.py file, it could be run directly instead, but the output would be a little

different. You can try that out on your own as an exercise.

Wrapping Up

Python's `logging` module gives you great flexibility when logging from multiple modules in your code.

You learned how to do this task in a couple of different ways:

- Logging using the logging API
- Logging using `dictConfig`

These aren't the only ways to log from multiple modules, but they are the most common. However, you will almost always have your logging API code in a separate module from `main.py`. Using a `dictConfig`, a `fileConfig`, or the logging API, you can create very powerful log handlers and formatters. You can also add filters here if you want to.

Start experimenting and see if you can add some logging to your code!

Chapter 8 - Creating a Logging Decorator

Logging boilerplate can make adding logging to your code more complex. One workaround is to wrap up your logging code in a decorator. Then, you can add a decorator to any functions that you want to add logging to.

This chapter will cover the following topics:

- What is a decorator?
- Creating an exception logging decorator
- Passing a logger to a decorator

You will focus on adding an exception logging decorator. However, you can extend or modify the decorator to log more than exceptions. That part is up to you.

Let's get started!

What is a Decorator?

A decorator is a function that takes another function as its argument. The decorator function will then extend the functionality of the function that it is decorating in some way. In the case of logging, you will create a decorator that catches exceptions and logs them if they occur in the decorated function.

That sounds like double-talk. Writing a decorator helps you to see how they work.

Open up your Python editor and create a file named `hello_decorator.py`, then enter the following code:

```
1 # hello_decorator.py
2
3 def info(func):
4     def wrapper(*args):
5         print(f"Function name: {func.__name__}")
6         print(f"Function docstring: {func.__doc__}")
```

```
        return func(*args)

    return wrapper


@info
def doubler(number):
    """Doubles the number passed to it"""
    return number * 2


print(doubler(4))
```

The `info()` function above takes in a function as its sole argument. Python implicitly passes in the function's arguments, so you see a `*args` in the nested `wrapper()` function inside of `info()`. The `wrapper()` function extends the functionality of the function being decorated.

In this case, you print out the function's name and docstring, so it's not an enhancement to the function. However, this example demonstrates how decorators work, and you can log the decorated function here if you want to.

When you run this code, you will see the following printed out:

```
Function name: doubler
Function docstring: Doubles the number passed to it
8
```

Note that decorators will replace the function's name and docstring with the decorator's name and function name. You can fix this behavior with Python's `functools.wraps`.

To demonstrate this, replace your code's `print()` function with the following: `print(doubler.__name__)`. When you run this, you will see that it prints out "wrapper" instead of "doubler".

Here's how to fix that issue with `functools`:

```
# hello_decorator.py

import functools
```

```
4
5  def info(func):
6      @functools.wraps(func)
7      def wrapper(*args):
8          print(f"Function name: {func.__name__}")
9          print(f"Function docstring: {func.__doc__}")
10         return func(*args)
11
12     return wrapper
13
14
15 @info
16 def doubler(number):
17     """Doubles the number passed to it"""
18     return number * 2
19
20
21 print(doubler.__name__)
```

All you need to do is import `functools` and add a decorator line to the nested `wrapper()` function inside `info()`. When you do this, you need to pass the `func` reference to `@functools.wraps()` to make it work.

Now let's move on and learn how to create a logging decorator!

Creating an Exception Logging Decorator

Creating a decorator that will log an exception will utilize much of what you've learned throughout this book. You will use Python's `logging` API to create a logger object and log to a file using a specific formatter.

To start, open up your Python IDE or text editor and create a new file named `exception_decor.py`. Then enter the following code:

```
1  # exception_decor.py
2
3  import functools
4  import logging
5
6
7  def create_logger():
8      """
9      Creates a logging object and returns it
10     """
```

```
    logger = logging.getLogger("example_logger")
    logger.setLevel(logging.INFO)

    # create the logging file handler
    fh = logging.FileHandler("test.log")

    fmt = ("%(asctime)s - %(name)s - %(levelname)s "
           "- %(message)s")
    formatter = logging.Formatter(fmt)
    fh.setFormatter(formatter)

    # add handler to logger object
    logger.addHandler(fh)
    return logger


def exception(function):
    """
    A decorator that wraps the passed in function and logs
    exceptions should one occur
    """

    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        logger = create_logger()
        try:
            return function(*args, **kwargs)
        except:
            # log the exception
            err = "There was an exception in  "
            err += function.__name__
            logger.exception(err)

            # re-raise the exception
            raise

    return wrapper
```

The `create_logger()` method uses the `logging` API to create an "example_logger" object that writes out to `test.log`. You also make a formatter using a format string you have seen several times. The last line of the function returns the logger object.

The `exception` function is your decorator. This function wraps a call to the function you are decorating with an exception handler (i.e., `try / except`) to catch any exceptions the decorated function might throw.

Your decorator emits a log message that includes the decorated function's name and the exception information.

Now, you need to test that the decorator works. In your Python IDE, create a second file named `main.py` and save it in the same location as `exception_decor.py`. Then add the following code:

```
# main.py

from exception_decor import exception


@exception
def zero_divide():
    1 / 0


if __name__ == "__main__":
    zero_divide()
```

The first step is to import your `exception` decorator. Then, you apply the decorator to the `zero_divide()` function, which foolishly divides an integer by zero. You raise an exception when you divide by zero in Python. The exception decorator should catch the exception and log it.

Try running the code in your terminal using `python main.py`, and you should see something like the following in your `test.log` file:

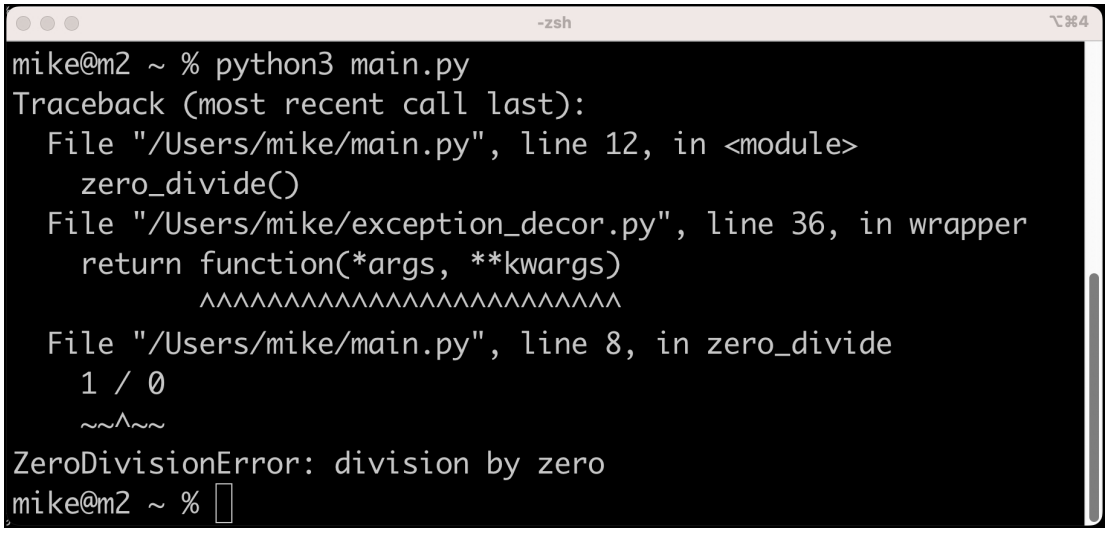


Figure 9. Zero division exception

Nice! You caught the exception and logged it successfully.

Passing a Logger to a Decorator

Complex code might have multiple logger objects. If that's the case, you will want to make your decorator be able to take in a logger object as an argument.

You will re-use much of the code from the previous example with a few small tweaks.

The first step is to open your Python IDE and create a new file named `exception_logger.py`. In that file, you will put all your logger API code.

Here's the code:

```
# exception_logger.py

import logging


def create_logger():
    """
    Creates a logging object and returns it
    """
    logger = logging.getLogger("example_logger")
    logger.setLevel(logging.INFO)

    # create the logging file handler
    fh = logging.FileHandler("test.log")

    fmt = ("%(asctime)s - %(name)s - %(levelname)s - "
           "%(message)s")
    formatter = logging.Formatter(fmt)
    fh.setFormatter(formatter)

    # add handler to logger object
    logger.addHandler(fh)
    return logger


logger = create_logger()
```

This code is nearly identical to the previous logger API code you had earlier. The difference here is

- The code is in a separate module and
- The logger instance is created at the end

Now, create the `exception_decor.py` file and save it in the same location as your `exception_logger.py` file. The decorator code needs to be updated to take in an argument.

Here's the code that will allow that:

```
# exception_decor.py

import functools


def exception(logger):
    """
    A decorator that wraps the passed in function and logs
    exceptions should one occur

    @param logger: The logging object
    """

    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            try:
                return function(*args, **kwargs)
            except:
                # log the exception
                err = "There was an exception in  "
                err += function.__name__
                logger.exception(err)

            # re-raise the exception
            raise

        return wrapper

    return decorator
```

The decorator now has two functions nested inside of it instead of one. The `exception()` function uses the `logger` object as its argument. The

`decorator()` function inside of `exception()` takes in the implicit function that `exception()` is decorating. The `wrapper()` function takes in the `args` and `kwargs` for the `function` that was implicitly passed in.

Otherwise, the code is the same as it was before.

The last step is to create a new `main.py` file in your Python IDE or text editor. The code is slightly different, so make sure you check it over:

```
# main.py

from exception_decor import exception
from exception_logger import logger


@exception(logger)
def zero_divide():
    1 / 0


if __name__ == "__main__":
    zero_divide()
```

This `main.py` imports your `logger` object and custom `exception` decorator. You pass in your `logger` object to the `exception()` decorator to be able to use it. You could create a different logger object and use it with the `exception()` decorator on a different function, which makes this coding pattern handy.

It would help if you created other functions that raise different exceptions as practice. Then, write up a couple of logger objects and pass them to each decorated function to see how flexible this is.

Wrapping Up

Decorators are powerful tools for extending regular functions. In a logging context, they can add logging to functions and methods without needing to import or configure logging in the module you decorate. All you need is the decorator.

In this chapter, you covered the following topics related to decorators and logging:

- What is a decorator?
- Creating an exception logging decorator
- Passing a logger to a decorator

Using decorators can make your code cleaner. The downside is that the tracebacks that occur can sometimes be more complex. However, decorators are a great design pattern to try. Just don’t overdo it!

Chapter 9 - Rotating Logs

The Python `logging` package comes with many built-in logging handlers. You have used `StreamHandler` to write out to stdout (i.e., your terminal) and `FileHandler` to write your logs to disk. However, you can use over a dozen additional logging handlers.

While this book won't cover all of them, it is good to understand how to see a couple of other examples. One of the most common patterns for logging is rotating logs. The idea of rotating a log is to prevent a log file from getting too large. When you rotate, you create a new log file and rename the old one. Your dev/ops team or system administrator can then move the old log to long-term storage.

In this chapter, you will learn about rotating logs in the following sections:

- Rotating logs based on file size
- Timed rotations
- Rotating logs using a config
- Customization using the Rotator and Namer

By this chapter's end, you can use Python to rotate your logs using several different approaches!

Rotating Logs Based on File Size

The most common way of rotating logs is to rotate them when you reach a specific file size. You might choose the file size based on your storage limits, the cost of parsing larger files, or other reasons.

Regardless of why you choose your log's file size, Python makes rotating your logs easier by providing a `RotatingFileHandler`. Your file will get rolled over right around when its size is about to be exceeded. You can also tell Python how many times you want it to roll over via the `backupCount` parameter.

Let's pretend that you want to set `backupCounter` to three. That tells Python to roll over the file three times. The fourth time, it will overwrite one of your previous logs. Remember, if your logs are important, you want to backup the log files when you roll over so you don't accidentally overwrite them.

Now that you understand the theory, you can write code to see log rotation in action! Create a new file called `simple_rotating.py` and then enter the following code:

```
# simple_rotating.py

import logging
import time

from logging.handlers import RotatingFileHandler


def create_rotating_log(path):
    """
    Creates a rotating log
    """
    logger = logging.getLogger("Rotating Log")
    logger.setLevel(logging.INFO)

    handler = RotatingFileHandler(
        path, maxBytes=20, backupCount=5)
    logger.addHandler(handler)

    for i in range(10):
        logger.info("This is test log line %s" % i)
        time.sleep(1.5)


if __name__ == "__main__":
    log_file = "test.log"
    create_rotating_log(log_file)
```

The `RotatingFileHandler` is found in `logging.handlers`, so ensure you import that correctly. The code in `create_rotating_log()` uses the simple root logger you get using the `logging` module's `getLogger()` method. You use this logger to keep the code short, but you will see more complex examples shortly.

When you create an instance of `RotatingFileHandler()`, you pass in three parameters:

- path - The file path to the initial log
- `maxBytes` - The maximum number of bytes for the file to grow to before rotating
- `backupCount` - The number of old log files to create

You can pass other parameters, such as the log's `encoding` and the file writing `mode`, but the three parameters listed above are by far the ones you will use the most.

When a log file is rotated, the rotated log will have a number appended to it. So, if you have a log file named `py.log`, when it rotates, the old file would be renamed `py.log.1`. The file that is always written to is the initial filename.

For your example, you set the file size to 20 bytes, a ridiculously small number. You would never do that in an actual application, as the log file would need more information to be useful. However, this is a quick way to demonstrate how file rotation works. You also set up the backup count to five, so there will be a maximum of five log files.

Go ahead and run the code. It generates six files with a single log message in each. If you open all of them, you will find that some files have been overwritten. You can tell that because the initial log message should be "This is test log line 0," but that string is not found in any of the files. Check and see for yourself.

Take a few moments to practice your new skills by changing the number of backups or the file size. You can also add a proper logger and formatter object to the mix.

Once you finish your experiments, you can continue to learn about timed rotations!

Timed Rotations

The `TimedRotatingFileHandler` allows you to create a rotating log based on how much time has elapsed. You can set it to rotate the log on the following time conditions:

- second (s)
- minute (m)
- hour (h)
- day (d)
- w0-w6 (weekday, 0=Monday)
- midnight

To set one of these conditions, pass it in the **when** parameter, which is the 2nd argument to `TimedRotatingFileHandler`. You will also need to set the **interval** parameter for the amount of time.

Open up your Python IDE and create a new file named `simple_timed_rotation.py`. Then enter the following code:

```
1  # simple_timed_rotation.py
2
3  import logging
4  import time
5
6  from logging.handlers import TimedRotatingFileHandler
7
8
9  def create_timed_rotating_log(path):
10     logger = logging.getLogger("Rotating Log")
11     logger.setLevel(logging.INFO)
12
13     handler = TimedRotatingFileHandler(path,
14                                        when="m",
15                                        interval=1,
16                                        backupCount=5)
17     logger.addHandler(handler)
18
19     for i in range(6):
20         logger.info("This is a test!")
21         time.sleep(75)
22
23
```

```
if __name__ == "__main__":
    log_file = "timed_test.log"
    create_timed_rotating_log(log_file)
```

Your code will rotate the log files every minute with a backup count of five. When you run this code, you should end up with six files. The initial log file is named `timed_test.log` while the other five will have a timestamp appended to the file name using Python's strftime (see the `time` or `datetime` modules) format of **%Y-%m-%d_%H-%M-%S**. Here's an example name: `timed_test.log.2024-03-19_11-45`.

Besides the **when** and **interval** parameters, the `TimedRotatingFileHandler` also takes in an **atTime** and a **utc** parameter. You use **atTime** when using the Weekday (W0-W6) or "midnight" option for the **when** parameter. If **atTime** is not None, it has to be set to a `datetime.time` instance. The `utc` parameter can be `None` or `True`. If you set `utc` to `True`, the logs will use UTC for the timestamp. Otherwise, the local time is the default.

Once you are done experimenting with your newfound knowledge, you will be ready to learn how to rotate logs using a configuration instead of the logging API.

Rotating Logs Using a Config

The `dictConfig` is the future of configurations in Python's `logging` module, so you will use that. Keeping your code lean and modular is usually recommended, so open up your Python IDE and create a new file named `settings.py` for your configuration file.

Then enter the following dictionary code into it:

```
# settings.py


LOGGING_CONFIG = {
    "version": 1,
    "loggers": {
        "__main__": {
            "handlers": ["rotatorFileHandler",
```

```
                            "consoleHandler"],
                "level": "INFO",
            },
        },
        "handlers": {
            "rotatorFileHandler": {
                "class": "logging.handlers.RotatingFileHandle\
r",
                "formatter": "file_formatter",
                "filename": "rotating.log",
                "maxBytes": 20,
                "backupCount": 5,

            },
            "consoleHandler": {
                "class": "logging.StreamHandler",
                "formatter": "stream_formatter",
            },
        },
        "formatters": {
            "file_formatter": {
                "format": "%(asctime)s - %(message)s",
            },
            "stream_formatter": {
                "format": "%(asctime)s - %(message)s",
                "datefmt": "%a %d %b %Y",
            },
        },
}
```

The main difference in this configuration versus previous config dictionaries is that you need to specify the `RotatingFileHandler` using its full import path: “logging.handlers.RotatingFileHandler.” You pass along the appropriate parameters to your handler by creating them as additional keys, such as “maxBytes” and “backupCount.”

Now that your configuration is defined, go back to your Python IDE and create a new file in the same location as `settings.py`. You will name this new file `main.py`.

Then you will need to add the following code:

```
# main.py

import logging.config
import settings
```

```
import time

logging.config.dictConfig(settings.LOGGING_CONFIG)


def main():
    logger = logging.getLogger(__name__)
    logger.debug("Logging is configured.")

    for i in range(10):
        logger.info("This is test log line %s" % i)
        time.sleep(1.5)


if __name__ == "__main__":
    main()
```

Here, you import `logging.config`, Python's `time` module, and your `settings` module; then load your custom logging configuration. The rest of the code creates a logger object and then logs ten informational log messages.

The last item you will learn about is modifying the rotator and namer. Find out what those terms mean in the next section!

Customization Using the Rotator and Namer

Python's logging cookbook, which is a part of Python's documentation, mentions that the two rotating log handlers have two special attributes that you can modify:

- rotator - Calls a function or callable when it's time to rotate the log
- namer - Calls a function or callable to rename the log when it rotates

Unfortunately, the `dictConfig()` and the `fileConfig()` methods do not currently have a direct way to set those attributes in the configuration itself. You need to get access to the logging handler itself to modify those. So that's what you will learn how to do in this section.

Open up your Python IDE and create a new folder for this mini-project. Copy in the `settings.py` file from the previous section into this new folder.

Then, create a new `main.py` file in the same location and add the following code to it:

```
# main.py

import gzip
import logging.config
import os
import shutil
import time

import settings

logging.config.dictConfig(settings.LOGGING_CONFIG)


def namer(filename):
    return f"{filename}.gz"

def rotator(source, destination):
    with open(source, "rb") as f_source:
        with gzip.open(destination, "wb") as f_dest:
            shutil.copyfileobj(f_source, f_dest)
    os.remove(source)

def main():
    logger = logging.getLogger(__name__)

    for handler in logger.handlers:
        if handler.name == "rotatorFileHandler":
            handler.namer = namer
            handler.rotator = rotator

    logger.debug("Logging is configured.")

    for i in range(10):
        logger.info("This is test log line %s" % i)
        time.sleep(1.5)


if __name__ == "__main__":
    main()
```

Let’s go over each function individually. The first function is `namer()`:

```
def namer(filename):
    return f"{filename}.gz"
```

This function takes in the file path of a file and appends ".gz" to the end of it.

The next function is the `rotator()` function:

```
def rotator(source, destination):
    with open(source, "rb") as f_source:
        with gzip.open(destination, "wb") as f_dest:
            shutil.copyfileobj(f_source, f_dest)
    os.remove(source)
```

When you rotate or rename a log file, your `rotator()` function takes in the `source` and `destination` file paths. In this example, you copy the file's contents and compress it with Python's `gzip` module. You then delete the original file since you compressed it.

In this case, the log only contains one line of text, so compressing it seems a little premature. However, in a real-world example, this could be very handy.

The last function is `main()`, which looks like this:

```
def main():
    logger = logging.getLogger(__name__)

    for handler in logger.handlers:
        if handler.name == "rotatorFileHandler":
            handler.namer = namer
            handler.rotator = rotator

    logger.debug("Logging is configured.")

    for i in range(10):
        logger.info("This is test log line %s" % i)
        time.sleep(1.5)
```

There isn't a direct way to access a specific handler, so instead, you loop over all the handlers in your logger object, searching for the one named "rotatorFileHandler." When you find that handler, you set the `namer` and

`rotator` attributes to the functions you created. You are not required to name the functions “namer” and “rotator.” Those are just convenient names.

The rest of the code is the same. Try running your code; you will get one log file and several gzipped files.

Wrapping Up

Engineers encounter log rotation regularly. Understanding how log rotation works and how to do it programmatically is a great tool for your bag of tricks.

In this chapter, you learned how to do the following:

- Rotating logs based on file size
- Timed rotations
- Rotating logs using a config
- Customization using the Rotator and Namer

You can take what you’ve learned here and start practicing. Use the code examples and modify them with different timings (seconds, minutes, days) or file sizes. Try writing your own rotator or naming methods to modify how your logs rotate. You’ll understand it so much better by working through the code yourself!

Chapter 10 - Logs and Concurrency

You usually log in a single thread. But what happens if you want to use concurrency in Python? Does the `logging` module still work in those situations? The answer is a resounding YES!

The Python core development team made the `logging` module thread-safe. However, logging is not quite as easy to use in `multiprocessing` or asynchronous scenarios. But logging is still doable with only a little extra work.

In this chapter, you will learn how to do logging in the following scenarios:

- Using `threading`
- Using `multiprocessing`
- Using threads and processes
- Using `concurrent.futures`
- Asynchronous Logging

Fair warning: The code examples in this chapter are more involved than earlier examples in this book. But that's normal when it comes to working with threads and processes. Soon enough, you will have all the knowledge you need to confidently log concurrently in Python

Using threading

Threading in Python is a bit complicated. In most programming languages, threading executes across multiple cores. However, in Python, threads are limited to one core by the Global Interpreter Lock or GIL. The GIL makes memory management and other threading activities easier and faster, but you get a different type of concurrency. Starting in Python 3.13, the GIL may get a switch to turn it off, so this limitation may lift soon.

Regardless, the Python programming language made the `logging` module thread-safe, so you don't have to do anything special to make logging work

with threads.

Open up your favorite Python IDE and create a new file named `threaded_logging.py`. Then enter the following code:

```
# threaded_logging.py

import logging
import threading
import time

FMT = '%(relativeCreated)6d %(threadName)s %(message)s'

def worker(message):
    logger = logging.getLogger("main-thread")
    while not message["stop"]:
        logger.debug("Hello from worker")
        time.sleep(0.05)

def main():
    logger = logging.getLogger("main-thread")
    logger.setLevel(logging.DEBUG)
    file_handler = logging.FileHandler("threaded.log")

    formatter = logging.Formatter(FMT)
    file_handler.setFormatter(formatter)
    logger.addHandler(file_handler)

    msg = {"stop": False}
    thread = threading.Thread(target=worker, args=(msg, ))
    thread.start()
    logger.debug("Hello from main function")
    time.sleep(2)
    msg["stop"] = True

    thread.join()

if __name__ == '__main__':
    main()
```

To keep things simpler, you use the logging API and get a logger that you name “main-thread.” Then, you set up a FileHandler()`instance to write your log to disk to a file named`threaded.log`. Next, you create a thread and tell it to run the` worker()` function in a thread.

The `worker()` takes in a `message` dictionary and logs out a message every so often. You then change the dictionary in your `main()` function, which causes the loop in `worker()` to end. The thread is then exited by calling `join()`.

If you open up the log file, you will find output that looks similar to the following:

```
     4 Thread-1 (worker) Hello from worker
     4 MainThread Hello from main function
    55 Thread-1 (worker) Hello from worker
   106 Thread-1 (worker) Hello from worker
   158 Thread-1 (worker) Hello from worker
   209 Thread-1 (worker) Hello from worker
   260 Thread-1 (worker) Hello from worker
   311 Thread-1 (worker) Hello from worker
```

Using threads and the `logging` module is straightforward. However, if you have used threads with a graphical user interface (GUI) such as Tkinter or wxPython, you will know that you need to use special thread-safe methods so you don't block the GUI's thread or the GUI will appear unresponsive.

However, you don't need to worry about blocking the `logging` module's ability to emit logs, as `logging` was designed to be thread-safe.

Now you're ready to see what's different when you use processes instead of threads!

Using multiprocessing

Processes are harder to work with than threads regarding Python's `logging` module. The main problem is that, by default, each process writes to its file. If you told them to all write to the same file, it wouldn't be serialized. That means you have each process stomping on each other, and the writes could get jumbled. You may even miss parts of the log messages.

If you want to serialize the process's messages so that they can all write successfully to a single file, then you'll want to use a queue. To do that, you

will use be using a code example that is based on one from the Python Documentation's Logging Cookbook.

Open up your Python IDE and create a new file named `process_logging.py`. Once you have the empty file saved, add the following code:

```
# process_logging.py

import logging
import logging.handlers
import multiprocessing
import time

from random import choice
from random import randint


LEVELS = [logging.DEBUG,
          logging.INFO,
          logging.WARNING,
          logging.ERROR,
          logging.CRITICAL]

LOGGERS = ["py_worker", "php_worker"]

MESSAGES = ["working hard",
            "taking a nap",
            "ERROR, ERROR, ERROR",
            "processing..."]
```

Wait a minute!? This code won't log anything! That's true, but this code example will be a bit longer than most and will be easier to look at if you write out the code bit-by-bit.

The code above contains the imports you need, along with three constants:

- LEVELS - A list of logging levels
- LOGGERS - A list a logger names
- MESSAGES - A list of logging messages

Now add the following two functions underneath the constants you added above:

```
def setup_logging():
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)

    file_handler = logging.FileHandler("processed.log")
    formatter = logging.Formatter(
        ("%(asctime)s - %(processName)-10s - "
         "%(levelname)s - %(message)s")
    )
    file_handler.setFormatter(formatter)
    root.addHandler(file_handler)


def listener_process(queue, configurer):
    configurer()
    while True:
        try:
            log_record = queue.get()
            if log_record is None:
                break
            logger = logging.getLogger(log_record.name)
            logger.handle(log_record)
        except Exception:
            import sys, traceback
            print("Error occurred in listener",
                  file=sys.stderr)
            traceback.print_exc(file=sys.stderr)
```

Here you have two functions:

- `setup_logging()` - Used for setting up logging for the listener process
- `listener_process()` - The process that writes each processes log message serially to the log handler

The `setup_logging()` code uses the root logger object directly. For this example, you write to disk using a `FileHandler` instance, but if you wanted to get fancy, you could swap in a `RotatingFileHandler` or any of the other handlers here instead.

The `listener_process()` will run the configurer, which is the `setup_logging()` function, as you will see later in the code. You will then create a loop that gets log records from the queue, grabs the logger by extracting the logger name from the log record, and then handles the log

record. That means you then emit the log message, which gets written to disk or stdout or whatever the handler you have set up is supposed to do.

Your next step is to configure and code up the worker-related functions:

```
def worker_configurer(queue):
    queue_handler = logging.handlers.QueueHandler(queue)
    root = logging.getLogger()
    root.addHandler(queue_handler)
    root.setLevel(logging.DEBUG)

def worker_process(queue, configurer):
    configurer(queue)
    name = multiprocessing.current_process().name
    print(f"Worker started: {name}")
    for _ in range(10):
        time.sleep(randint(1, 5))
        logger = logging.getLogger(choice(LOGGERS))
        level = choice(LEVELS)
        message = choice(MESSAGES)
        logger.log(level, message)
    print(f"Worker finished: {name}")
```

Each worker also needs to configure its own logging handler. In this case, since you want all the logs serialized, you will use the `QueueHandler`. The `QueueHandler` supports sending logging messages to a Python's `queue` module or `multiprocessing.Queue`. This queue lets you have a separate thread that does all the logging work.

The `worker_process()` is where you put your code to do real work. Since this is a contrived example, you don't do any real work here, but you simulate the work using a random amount of `time.sleep()`. Then, you log out what you did by choosing from a random logger, log level, and log message.

Here's the final piece of code you need to add:

```
def main():
    queue = multiprocessing.Queue(-1)
    listener = multiprocessing.Process(
        target=listener_process,
        args=(queue, setup_logging)
    )
    listener.start()
```

```
    workers = []
    for _ in range(10):
        worker = multiprocessing.Process(
            target=worker_process,
            args=(queue, worker_configurer)
        )
        workers.append(worker)
        worker.start()
    for task in workers:
        task.join()

    queue.put_nowait(None)
    listener.join()

if __name__ == '__main__':
    main()
```

Here, `main()` will create a `multiprocessing.Queue()` instance and pass that queue along to the listener process and each worker process. After you start all ten of your workers, you call `join()` to wait for them to finish.

The last steps are to put `None` into the queue to make the queue itself end and then `join()` the listener process.

When you run the code, you will see output that looks similar to the following:

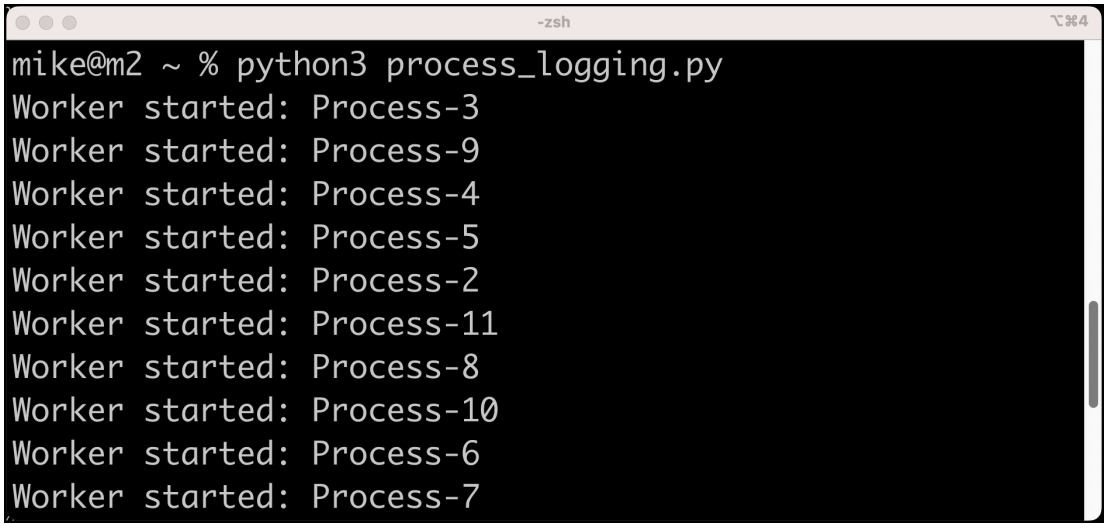


Figure 10. Multi.log output

However, other ways exist to work with processes and the logging module. You can also put threads into the mix!

Using Threads and Processes

Using a thread for the listener process rather than a `multiprocessing.Process()` object will simplify your code, but putting the log configuration into a `dictConfig` will reduce the line count considerably.

To start, create a new file in your Python IDE called `thread_and_process.py`. Save your empty file and then add this code to it:

```
# thread_and_process.py

import logging
import logging.config
import logging.handlers
import random
import threading
import time
import settings

from multiprocessing import Process, Queue


logging.config.dictConfig(settings.LOGGING_CONFIG)


def logger_thread(queue):
    while True:
        log_record = queue.get()
        if log_record is None:
            break
        logger = logging.getLogger(log_record.name)
        logger.handle(log_record)
```

The first step is you import a few new things:

- `logging.config` - You'll need this to load the `dictConfig()`
- `threading` - For creating your listener thread
- `settings` - A custom module that contains your dictionary config

Then you write up the `logger_thread()` function, which you will call using a `Thread()`. Much like the code in the previous section, the `logger_thread()` takes in a `queue` object and gets the log records from the queue. Then, it extracts the logger's name and emits the message so it gets handled by the logging handler.

Now you're ready to add the `worker_process()`:

```
def worker_process(q):
    queue_handler = logging.handlers.QueueHandler(q)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(queue_handler)
    levels = [logging.DEBUG,
              logging.INFO,
              logging.WARNING,
              logging.ERROR,
              logging.CRITICAL]
    loggers = ['py', 'py.egg', 'py.egg.baby',
               'spam', 'spam.ham', 'spam.ham.eggs']
    for i in range(100):
        lvl = random.choice(levels)
        logger = logging.getLogger(random.choice(loggers))
        logger.log(lvl, 'Message no. %d', i)
```

Once again, you will use the `QueueHandler()` to help serialize the logging messages. You also have all the logging configurations stuffed into this process function, while in the previous section, you split the logging configuration into a separate function.

Here, you pick a random logger and logging level and then log out a message.

The final piece of the puzzle in this file is your `main()` function:

```
def main():
    queue = Queue()
    workers = []
    for i in range(5):
        worker = Process(
            target=worker_process,
            name='worker %d' % (i + 1),
            args=(queue,))
        workers.append(worker)
```

```
        worker.start()
    listener_process = threading.Thread(
        target=logger_thread,
        args=(queue, ))
    listener_process.start()
    # Do some work here
    time.sleep(3)
    for worker in workers:
        worker.join()
    # End the logger listener process
    queue.put(None)
    listener_process.join()


if __name__ == '__main__':
    main()
```

Here, you create a `multiprocessing.Queue()`, five worker processes, and a listener thread. You pass the queue to each worker and the thread and start them all up. The rest of the code is the same as before.

The last item on your TODO list is to create the `settings.py` file and add the following dictionary config:

```
# settings.py

LOGGING_CONFIG = {
    "version": 1,
    "formatters": {
        "detailed": {
            "class": "logging.Formatter",
            "format": "%(asctime)s %(name)-15s %(levelnam\
e)-8s %(processName)-10s %(message)s",
        }
    },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "level": "INFO",
        },
        "file": {
            "class": "logging.FileHandler",
            "filename": "thread_and_process.log",
            "mode": "w",
            "formatter": "detailed",
        },
        "pyfile": {
            "class": "logging.FileHandler",
```

```
            "filename": "thread_and_process-py.log",
            "mode": "w",
            "formatter": "detailed",
        },
        "errors": {
            "class": "logging.FileHandler",
            "filename": "thread_and_process-errors.log",
            "mode": "w",
            "level": "ERROR",
            "formatter": "detailed",
        },
    },
    "loggers": {"py": {"handlers": ["pyfile"]}},
    "root": {"level": "DEBUG", "handlers": ["console", "f\
ile", "errors"]},
}
```

If you don't want to enter this code, feel free to download it from the book's GitHub repo (see the introduction).

Try running the code and then check the output. It should look something like the following:

Figure 11. Threads and processes output

With this code, you create three log files. The above output is similar to what you'll find in all three.

Now you're ready to move on and learn how to simplify this code even more using Python's handy `concurrent.futures` module!

Using concurrent.futures

Python has a special wrapper around its threading and multiprocessing modules called `concurrent.futures`. This module further simplifies the use of these modules.

But how do you use `concurrent.futures` with `logging`? Well, you're about to find out! To try this out, copy `thread_and_process.py` and `settings.py` to a new folder. However, rename the `thread_and_process.py` to `futures_thread_and_process.py` to help you keep track of which is which.

Then, update the imports in that file to match the following:

```
# futures_thread_and_process.py

import concurrent.futures
import logging
import logging.config
import logging.handlers
import multiprocessing
import random
import threading
import time
import settings
```

The primary change here is that you are also importing `concurrent.futures`.

You will also need to update `main()` to the following:

```
def main():
    queue = multiprocessing.Manager().Queue(-1)

    with concurrent.futures.ProcessPoolExecutor(
        max_workers=10) as executor:
        for i in range(10):
            executor.submit(worker_process, queue)

    listener_process = threading.Thread(
        target=logger_thread,
        args=(queue, ))
    listener_process.start()
```

```
    # Do some work here
    time.sleep(3)

    # End the logger listener process and queue
    queue.put(None)
    listener_process.join()
```

The first change is to swap out:

```
queue = multiprocessing.Queue(-1)
```

for the following:

```
queue = multiprocessing.Manager().Queue(-1)
```

The latter would have worked in the original version of the code too.

The next step is to drop the `worker` creation loops from earlier. You no longer need any of that. Instead, you will replace that code with this:

```
with concurrent.futures.ProcessPoolExecutor(
    max_workers=10) as executor:
    for i in range(10):
        executor.submit(worker_process, queue)
```

The code above will create a Process Pool with ten workers. It is equivalent to this code:

```
workers = []
for i in range(10):
    worker = Process(
        target=worker_process,
        name='worker %d' % (i + 1),
        args=(queue,))
    workers.append(worker)
    worker.start()
for worker in workers:
    worker.join()
```

That’s all you need to change. Try running the code; you should see similar output in your log files!

Now you’re ready to learn the last topic of logging with `asyncio`.

Asynchronous Logging

Python has asynchronous support via its `asyncio` library. Using the `asyncio` along with some special syntax, you can write asynchronous code in Python. But the `logging` module is known as a "blocking" library because when it writes to disk, that code is not asynchronous.

You can solve this problem using the `QueueHandler` and the `QueueListener`.

To see how this works, return to your Python IDE and create a new file named `async_logging.py`. Then enter the following code:

```
# async_logging.py

import asyncio
import logging
import logging.handlers
import random

from queue import SimpleQueue

MESSAGES = ["working hard",
            "taking a nap",
            "ERROR, ERROR, ERROR",
            "processing..."]


async def setup_logging():
    log_queue = SimpleQueue()
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)

    # Create a non-blocking handler
    queue_handler = logging.handlers.QueueHandler(
        log_queue)
    root.addHandler(queue_handler)

    # Create a blocking handler
    file_handler = logging.FileHandler("queued.log")
    formatter = logging.Formatter(
        ("%(asctime)s - %(name)s - %(levelname)s "
         "- %(message)s")
    )
    file_handler.setFormatter(formatter)
```

```
33
34      listener = logging.handlers.QueueListener(
35          log_queue, file_handler)
36      try:
37          listener.start()
38          logging.debug("Async logging started")
39          while True:
40              await asyncio.sleep(60)
41      finally:
42          logging.debug("Logger is being shut down!")
43          listener.stop()
44
45
46  async def task(number):
47      logging.info(f"Starting task #{number}")
48      await asyncio.sleep(random.randint(1, 5))
49      msg = random.choice(MESSAGES)
50      logging.info(f"Task {number} is {msg}")
51      logging.info(f"Task #{number} is finished")
52
53
54  async def main():
55      # initialize the logger
56      asyncio.create_task(setup_logging())
57      await asyncio.sleep(0.1)
58
59      logging.info("Main function started")
60
61      async with asyncio.TaskGroup() as group:
62          for t in range(10):
63              group.create_task(task(t))
64
65      logging.info("All work done")
66
67
68  if __name__ == "__main__":
69      asyncio.run(main())
```

That was a large chunk of code! To make it easier to digest, you will review each function individually.

You can start by looking at `setup_logging()`:

```
1  async def setup_logging():
2      log_queue = SimpleQueue()
3      root = logging.getLogger()
4      root.setLevel(logging.DEBUG)
5
6      # Create a non-blocking handler
```

```
    queue_handler = logging.handlers.QueueHandler(
        log_queue)
    root.addHandler(queue_handler)

    # Create a blocking handler
    file_handler = logging.FileHandler("queued.log")
    formatter = logging.Formatter(
        ("%(asctime)s - %(name)s - %(levelname)s "
         "- %(message)s")
    )
    file_handler.setFormatter(formatter)

    listener = logging.handlers.QueueListener(
        log_queue, file_handler)
    try:
        listener.start()
        logging.debug("Async logging started")
        while True:
            await asyncio.sleep(60)
    finally:
        logging.debug("Logger is being shut down!")
        listener.stop()
```

If you are unfamiliar with async in Python, you might get thrown by the above syntax. Whenever you see `async def`, that means that function is an asynchronous function. In `setup_logging()`, you create a `queue` in much the same way as you have in the past. However, the difference is that you use the `queue` module rather than `multiprocessing`.

Next, you create a `QueueHandler`, which is the only non-blocking logging handler in the `logging` module. However, you do need a blocking handler to log out of the queue. So, you create that next and add the blocking handler to a `QueueListener` instance.

Using this coding pattern, the blocking handler is essentially put into a separate thread that can't block your asynchronous program.

Now you're ready for the `task()` function:

```
async def task(number):
    logging.info(f"Starting task #{number}")
    await asyncio.sleep(random.randint(1, 5))
    msg = random.choice(MESSAGES)
```

```
    logging.info(f"Task {number} is {msg}")
    logging.info(f"Task #{number} is finished")
```

In asynchronous terminology, a task is analogous to a thread or process. Tasks are where you do your work in an asynchronous application. Here, you pretend to work by using a random sleep time. Then you log out some messages.

The last function to look at is `main()`:

```
async def main():
    # initialize the logger
    asyncio.create_task(setup_logging())
    await asyncio.sleep(0.1)

    logging.info("Main function started")

    async with asyncio.TaskGroup() as group:
        for t in range(10):
            group.create_task(task(t))

    logging.info("All work done")


if __name__ == "__main__":
    asyncio.run(main())
```

The first step in `main()` is configuring logging, which also sets up your listener. Then, you log something out from `main()` and start all the worker tasks. In this example, you create ten tasks. Then, you finish by logging once more.

Pretty straightforward, right?

When you run this code, you will see output that is similar to the following in your log file:

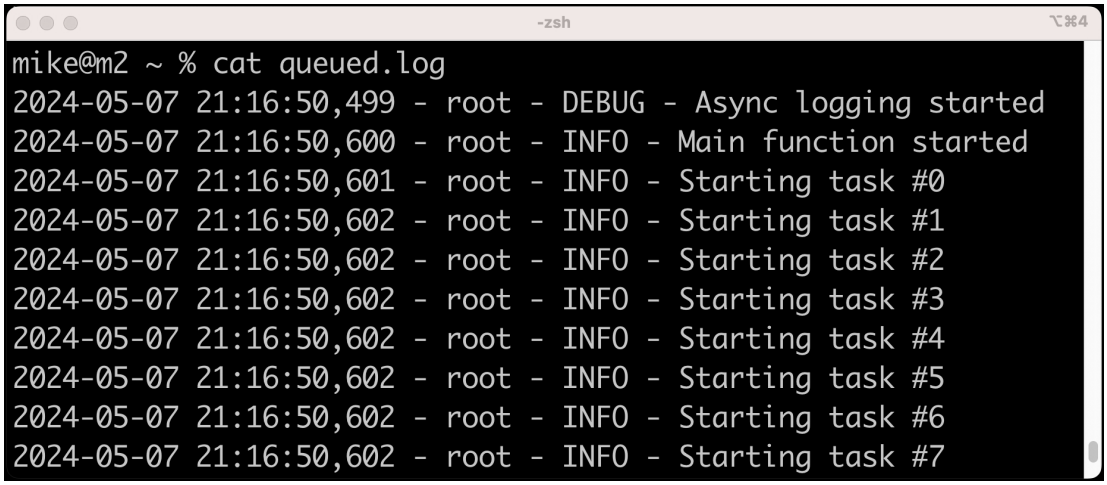


Figure 12. Async output

There is an alternative method for logging asynchronously, and that is to use an external package instead of Python's built-in `logging` module. One of the most popular is aiologger, which is tailor-made for logging in an asynchronous context.

Wrapping Up

Concurrency in Python can make your applications run faster and more efficiently. However, if you need to audit your programs by logging, you need to understand how to use the `logging` module in a concurrent context.

In this chapter, you learned how to do just that in the following scenarios:

- Using `threading`
- Using `multiprocessing`
- Using threads and processes
- Using `concurrent.futures`
- Asynchronous Logging

The `logging` module provides all the tools you need to log in concurrent contexts successfully. With a little practice, you will be able to add logging to your concurrent code confidently. Try the examples in this chapter, then remix them and try some variations. You'll be logging in no time!

Chapter 11 - Logging with Loguru

Python's `logging` module isn't the only way to create logs. There are several third-party packages you can use, too. One of the most popular is Loguru. Loguru intends to remove all the boilerplate you get with the Python logging API.

You will find that Loguru greatly simplifies creating logs in Python.

This chapter has the following sections:

- Installation
- Logging made simple
- Handlers and formatting
- Catching exceptions
- Terminal logging with color
- Easy log rotation

Let's find out how much easier Loguru makes logging in Python!

Installation

You will need to install Loguru before you can start using it. After all, the Loguru package doesn't come with Python.

Fortunately, installing Loguru is easy with pip. Open up your terminal and run the following command:

```
1 python -m pip install loguru
```

Pip will install Loguru and any dependencies it might have for you. If you see no errors, you will have a working package installed.

Now let's start logging!

Logging Made Simple

Logging with Loguru can be done in two lines of code. Loguru is really that simple!

Don’t believe it? Then open up your Python IDE or REPL and add the following code:

```
# hello.py

from loguru import logger

logger.debug("Hello from loguru!")
logger.info("Informed from loguru!")
```

One import is all you need. Then, you can immediately start logging! By default, the log will go to stdout.

Here’s what the output looks like in the terminal:

```
2024-05-07 14:34:28.663 | DEBUG    | __main__:<module>:5 \
- Hello from loguru!
2024-05-07 14:34:28.664 | INFO     | __main__:<module>:6 \
- Informed from loguru!
```

Pretty neat! Now, let’s find out how to change the handler and add formatting to your output.

Handlers and Formatting

Loguru doesn’t think of handlers the way the Python logging module does. Instead, you use the concept of `sinks`. The sink tells Loguru how to handle an incoming log message and write it somewhere.

Sinks can take lots of different forms:

- A file-like object, such as `sys.stderr` or a file handle
- A file path as a string or `pathlib.Path`
- A `callable`, such as a simple function
- An asynchronous `coroutine function` that you define using `async def`

- A built-in `logging.Handler`. If you use these, the Loguru records convert to `logging` records automatically

To see how this works, create a new file called `file_formatting.py` in your Python IDE. Then add the following code:

```
# file_formatting.py

from loguru import logger

fmt = "{time} - {name} - {level} - {message}"

logger.add("formatted.log", format=fmt, level="INFO")
logger.debug("This is a debug message")
logger.info("This is an informational message")
```

If you want to change where the logs go, use the `add()` method. Note that this adds a new sink, which, in this case, is a file. The logger will still log to stdout, too, as that is the default, and you are adding to the handler list. If you want to remove the default sink, add `logger.remove()` before you call `add()`.

When you call `add()`, you can pass in several different arguments:

- sink - Where to send the log messages
- level - The logging level
- format - How to format the log messages
- filter - A logging filter

There are several more, but those are the ones you would use the most. If you want to know more about `add()`, you should check out the documentation.

You might have noticed that the formatting of the log records is a little different than what you saw in Python's `logging` module.

Here is a listing of the formatting directives you can use for Loguru:

- elapsed - The time elapsed since the app started

- exception - The formatted exception, if there was one
- extra - The dict of attributes that the user bound
- file - The name of the file where the logging call came from
- function - The function where the logging call came from
- level - The logging level
- line - The line number in the source code
- message - The unformatted logged message
- module - The module that the logging call was made from
- name - The `__name__` where the logging call came from
- process - The process in which the logging call was made
- thread - The thread in which the logging call was made
- time - The aware local time when the logging call was made

You can also change the time formatting in the logs. In this case, you would use a subset of the formatting from the Pendulum package. For example, if you wanted to make the time exclude the date, you would use this: `{time:HH:mm:ss}` rather than simply `{time}`, which you see in the code example above.

See the documentation for details on formatting time and messages.

When you run the code example, you will see something similar to the following in your log file:

```
2024-05-07T14:35:06.553342-0500 - __main__ - INFO - This \
is an informational message
```

You will also see log messages sent to your terminal in the same format as you saw in the first code example.

Now, you're ready to move on and learn about catching exceptions with Loguru.

Catching Exceptions

Catching exceptions with Loguru is done by using a decorator. You may remember that when you use Python's own `logging` module, you use `logger.exception` in the `except` portion of a `try/except` statement to record the exception's traceback to your log file.

When you use Loguru, you use the `@logger.catch` decorator on the function that contains code that may raise an exception.

Open up your Python IDE and create a new file named `catching_exceptions.py`. Then enter the following code:

```
# catching_exceptions.py

from loguru import logger

@logger.catch
def silly_function(x, y, z):
    return 1 / (x + y + z)

def main():
    fmt = "{time:HH:mm:ss} - {name} - {level} - {message}"
    logger.add("exception.log", format=fmt, level="INFO")
    logger.info("Application starting")
    silly_function(0, 0, 0)
    logger.info("Finished!")

if __name__ == "__main__":
    main()
```

According to Loguru's documentation, the' @logger.catch` decorator will catch regular exceptions and also work with applications with multiple threads. Add another file handler to the stream handler and start logging for this example.

Then you call `silly_function()` with a bunch of zeroes, which causes a `ZeroDivisionError` exception.

Here's the output from the terminal:

```
2024-05-07 14:38:30.258 | INFO     | __main__:main:12 - A\
pplication starting
2024-05-07 14:38:30.259 | ERROR    | __main__:main:13 - A\
n error has been caught in function 'main', process 'Main
```

```
Process' (8920), thread 'MainThread' (22316):
Traceback (most recent call last):

  File "C:\books\11_loguru\catching_exceptions.py", line \
17, in <module>
    main()
    └ <function main at 0x00000253B01AB7E0>

> File "C:\books\11_loguru\catching_exceptions.py", line \
13, in main
    silly_function(0, 0, 0)
    └ <function silly_function at 0x00000253ADE6D440>

  File "C:\books\11_loguru\catching_exceptions.py", line \
7, in silly_function
    return 1 / (x + y + z)
                │   │   └ 0
                │   └ 0
                └ 0

ZeroDivisionError: division by zero
2024-05-07 14:38:30.264 | INFO     | __main__:main:14 - F\
inished!
```

Note: This looks much better if you run the Python code in your terminal as you can see the colors there!

If you open up the `exception.log`, you will see that the contents are a little different because you formatted the timestamp and also because logging those funny lines that show what arguments were passed to the `silly_function()` don't translate that well:

```
14:38:30 - __main__ - INFO - Application starting
14:38:30 - __main__ - ERROR - An error has been caught in\
 function 'main', process 'MainProcess' (8920), thread 'M
ainThread' (22316):
Traceback (most recent call last):

  File "C:\books\11_loguru\catching_exceptions.py", line \
17, in <module>
    main()
    â"" <function main at 0x00000253B01AB7E0>

> File "C:\books\11_loguru\catching_exceptions.py", line \
13, in main
    silly_function(0, 0, 0)
    â"" <function silly_function at 0x00000253ADE6D440>
```

```
16
17   File "C:\books\11_loguru\catching_exceptions.py", line \
18 7, in silly_function
19     return 1 / (x + y + z)
20                  â”,   â”,   â”” 0
21                  â”,   â”” 0
22                  â”” 0
23
24 ZeroDivisionError: division by zero
25 14:38:30 - __main__ - INFO - Finished!
```

Overall, using the `@logger.catch` is a nice way to catch exceptions.

Now, you're ready to move on and learn about changing the color of your logs in the terminal.

Terminal Logging with Color

Loguru will print out logs in color in the terminal by default if the terminal supports color. Colorful logs can make reading through the logs easier as you can highlight warnings and exceptions with unique colors.

You can use markup tags to add specific colors to any formatter string. You can also apply bold and underlining to the tags.

Open up your Python IDE and create a new file called `terminal_formatting.py`. After saving the file, enter the following code into it:

```
1  # terminal_formatting.py
2  import sys
3  from loguru import logger
4
5  fmt = ("<red>{time}</red> - "
6         "<yellow>{name}</yellow> - "
7         "{level} - {message}")
8
9  logger.add(sys.stdout, format=fmt, level="DEBUG")
10 logger.debug("This is a debug message")
11 logger.info("This is an informational message")
```

You create a special format where you set the “time” portion to red and the “name” to yellow. Then, you `add()` that format to the logger. You will now have two sinks: the default root handler, which logs to stderr, and the new sink, which logs to stdout. You do formatting to compare the default colors to your custom ones.

Go ahead and run the code. You should see something like this:

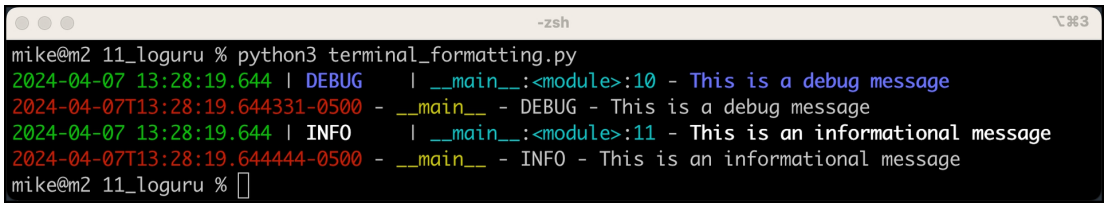


Figure 13. Loguru terminal color formatting

Note: This example is best done in your terminal as it is hard to reproduce in a book format.

Neat! It would be best if you now spent a few moments studying the documentation and trying out some of the other colors. For example, you can use hex, RGB, and a handful of named colors.

The last section you will look at is how to do log rotation with Loguru!

Easy Log Rotation

Loguru makes log rotation easy. You don’t need to import any special handlers. Instead, you only need to specify the `rotation` argument when you call `add()`.

Here are a few examples:

- `logger.add("file.log", rotation="100 MB")`
- `logger.add("file.log", rotation="12:00")`
- `logger.add("file.log", rotation="1 week")`

These demonstrate that you can set the rotation at 100 megabytes at noon daily or even rotate weekly.

Open up your Python IDE so you can create a full-fledged example. Name the file `log_rotation.py` and add the following code:

```
# log_rotation.py

from loguru import logger

fmt = "{time} - {name} - {level} - {message}"

logger.add("rotated.log",
           format=fmt,
           level="DEBUG",
           rotation="50 B")
logger.debug("This is a debug message")
logger.info("This is an informational message")
```

Here, you set up a log format, set the level to DEBUG, and set the rotation to every 50 bytes. You will get a couple of log files when you run this code. Loguru will add a timestamp to the file's name when it rotates the log.

What if you want to add compression? You don't need to override the rotator like you did with Python's `logging` module. Instead, you can turn on compression using the `compression` argument.

Create a new Python script called `log_rotation_compression.py` and add this code for a fully working example:

```
# log_rotation_compression.py

from loguru import logger

fmt = "{time} - {name} - {level} - {message}"

logger.add("compressed.log",
           format=fmt,
           level="DEBUG",
           rotation="50 B",
           compression="zip")
logger.debug("This is a debug message")
logger.info("This is an informational message")
for i in range(10):
    logger.info(f"Log message {i}")
```

The new file is automatically compressed in the zip format when the log rotates. There is also a `retention` argument that you can use with `add()` to tell Loguru to clean the logs after so many days:

```
`logger.add("file.log",
            rotation="100 MB",
            retention="5 days")
```

If you were to add this code, the logs that were more than five days old would get cleaned up automatically by Loguru!

Wrapping Up

The Loguru package makes logging much easier than Python's `logging` library. It removes the boilerplate needed to create and format logs.

In this chapter, you learned about the following:

- Installation
- Logging made simple
- Handlers and formatting
- Catching exceptions
- Terminal logging with color
- Easy log rotation

Loguru can do much more than what is covered here, though. You can serialize your logs to JSON or contextualize your logger messages. Loguru also allows you to add lazy evaluation to your logs to prevent them from affecting performance in production. Loguru also makes adding custom log levels very easy. For full details about all the things Loguru can do, you should consult Loguru's website.

Chapter 12 - Logging with Structlog

There are several different Python logging packages out there. You don't have to use Python's built-in `logging` module if you don't want to. However, Python's `logging` module is well documented and used more commonly than third-party logging packages.

However, due to the `logging` module's boilerplate, there are a couple of popular alternatives. One of the most popular is `structlog`, which was created by Hynek Schlawack. Structlog works through functions that return dictionaries. You will find that `structlog` is simpler to use, powerful, and fast.

In this chapter, you will cover the following:

- Installing `structlog`
- structlog's log levels
- Log formatting with structlog
- Updating the timestamp
- Serializing to JSON
- Logging exceptions with structlog
- Logging to disk

Let's start by learning how to get set up with `structlog`!

Installing structlog

The `structlog` package is not built into the Python programming language, so you will need to install it. Fortunately, you can do that easily using the `pip` installer.

Open up your terminal or cmd.exe and run the following command:

```
1 python -m pip install structlog
```

The `pip` installer is usually quick, and you will have `structlog` installed in less than a minute unless your internet connection is really slow.

If you want pretty exceptions in your terminal, you can install Rich or better-exceptions. The screenshots in structolog's documentation show the output using Rich. If you are a Windows user, you should install Colorama to add colorful output.

Now that everything is installed, you're ready to learn how to log!

structlog's Log Levels

The `structlog` package uses the same log levels as Python's logging module. That means that you can still use the following log levels as usual:

- debug
- info
- warning
- error
- critical

These levels are applied when you call the logger methods using the same name.

Open up your Python IDE and create a new file named `hello_structlog.py`. Then enter the following code into it:

```
1 # hello_structlog.py
2
3 import structlog
4
5 logger = structlog.get_logger()
6 logger.info("Hello %s", "Mike", key=12, my_list=[5, 6, 7])
```

You import `structlog` and get the logger object via the `get_logger()` method call. Note that in `structlog`, you use an underscore, while in Python's `logging` module, you would use a type of camelcase (i.e., `getLogger()`).

If you run this code in your terminal, you will see that `structlog` prints its log output in color, assuming you have installed the optional packages mentioned in the previous section:

```
2024-05-07 21:21:31 [info     ] Hello Mike                   \
       key=12 my_list=[5, 6, 7]
```

Note: To see this in color, you will need to run the code in your terminal.

What if you want to set the log level in `structlog`, though? To do so, you can import Python's logging module and use it in conjunction with `structlog`'s `make_filtering_bound_logger()` method, which you use inside of `structlog.configure()`. Alternatively, you could use the integer value the logging levels map to. For example, `logging.INFO` is 20.

You can see how this works with another code example. Create a new file named `structlog_set_level.py` and enter the following code:

```
# structlog_set_level.py

import logging
import structlog

structlog.configure(
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO)
)
logger = structlog.get_logger()
logger.debug("This is a debug message")
logger.info("Hello %s", "Mike", key=12, my_list=[5, 6, 7])
```

Configuring `structlog` is pretty straightforward. Here, you set the logging level to INFO, which means that when you log out a debug message, the logger will ignore the message and not emit it. The only log message you will see is the info message.

Now that you know how logging levels work in `structlog`, you can learn how to format your log messages!

Log Formatting with structlog

Formatting your logs is important. You want to be able to record whatever is useful for you regarding audit trails or debugging. Fortunately, `structlog` makes formatting easy by creating a `processors` list to pass along to the `configure()` method.

Open up your Python IDE and, create a new file called `formatting.py`, and then enter the following code:

```
# formatting.py

import logging
import structlog

structlog.configure(
    processors=[
        structlog.processors.add_log_level,
        structlog.dev.ConsoleRenderer(),
    ],
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO)
)
logger = structlog.get_logger()
logger.info("This is an info message")
```

The first item you add to `processors` is `structlog.processors.add_log_level`, which adds the log level to the log message. You also add `structlog.dev.ConsoleRenderer()`, which is the default log handler. You can think of the ConsoleRenderer() `as equivalent to the StreamHandler()` from the `logging` module.

When you run this code, you will see the following output in your terminal:

```
[info     ] This is an info message
```

However, this log does not have a timestamp. That’s an important piece of information. You will learn how to add that next!

Updating the Timestamp

Timestamps are important. When did someone access the SQL server or delete a file? What time of day did the application stop working? You will only know if you record the time!

You can add a timestamp to your logs in `structlog` by adding a call to `TimeStamper()`. If you don't provide any arguments to `TimeStamper()`, it will use the number of seconds since the epoch, which is pretty useless for humans.

To make something human-readable, you should use "ISO" or create your own format, like "%Y-%m-%d %H:%M:%S." You can also set the `utc` argument to `True` or `False`.

You should write some code to see how this works. Open up your Python IDE and, create a file named `formatting_timestamp.py`, and add this code to it:

```
# formatting_timestamp.py

import logging
import structlog

structlog.configure(
    processors=[
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.add_log_level,
        structlog.dev.ConsoleRenderer(),
    ],
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO)
)
logger = structlog.get_logger()
logger.info("This is an info message")
```

When you run this code, you will see a timestamp added to the output that will look similar to the following:

```
2024-04-04T15:43:02.653557Z [info     ] This is an info m\
essage
```

If you dislike that timestamp, swap it out with "%Y-%m-%d %H:%M:%S" or make your own custom timestamp!

Now that you have timestamps sorted out, you are ready to learn how to serialize your logs to JSON.

Serializing to JSON

Serializing your logs to JSON can make the logs easier to digest by other services, such as DataDog. The `structlog` package makes this easy as it has a `JSONRenderer()` processor to add to your `processors` list.

Create a new file named `serializing_json.py` in your Python IDE and add this code:

```
# serializing_json.py

import logging
import structlog

structlog.configure(
    processors=[
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.add_log_level,
        structlog.processors.JSONRenderer(),
    ],
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO)
)
logger = structlog.get_logger()
logger.info("This is an info message")
```

You no longer need the `ConsoleRenderer()` when you use the `JSONRenderer()`. The output will still go to your terminal, but this time, it will be in JSON format.

Here's what your output will look like (minus the extra carriage returns):

```
{"event": "This is an info message",
 "timestamp": "2024-04-04T16:01:35.636957Z",
 "level": "info"}
```

That's pretty handy! Try adding some additional log messages and see how the output changes.

Now you're ready to learn about catching exceptions!

Logging Exceptions using structlog

As you might expect, `structlog` can log exceptions. In fact, `structlog` uses the same syntax as Python's `logging` module to log exceptions.

As usual, though, you should write a small, runnable example. Open up your Python IDE and create a file called `structlog_exception.py`. Then add this code to the file:

```
# structlog_exception.py

import logging
import structlog

structlog.configure(
    processors=[
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.add_log_level,
        structlog.dev.ConsoleRenderer()
    ],
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO)
)
logger = structlog.get_logger()
logger.info("A message before the exception")

try:
    10 / 0
except ZeroDivisionError:
    logger.exception("You cannot divide by zero!")
```

Here, you add almost the same code as the previous example except that you use a `ConsoleRenderer()` and divide by zero. Note that you call the `exception()` method to catch and log the exception.

If you run this code in your terminal, you will get lovely color-enhanced output. Be sure to open your terminal and run your code there to see it as it can't be reproduced nicely in book form.

If you don't need the traceback, you can use the `JSONRenderer()` instead, like this:

```
# structlog_exception_json.py

import logging
import structlog

structlog.configure(
    processors=[
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.add_log_level,
        structlog.processors.JSONRenderer()
    ],
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO)
)
logger = structlog.get_logger()
logger.info("A message before the exception")

try:
    10 / 0
except ZeroDivisionError:
    logger.exception("You cannot divide by zero!")
```

When you run this code, you will get the following JSON (which has been formatted to fit the page better):

```
{"event": "A message before the exception",
"timestamp": "2024-04-04T17:44:20.779068Z",
"level": "info"}
{"exc_info": true, "event":
"You cannot divide by zero!",
"timestamp": "2024-04-04T17:44:20.779068Z",
"level": "error"}
```

Note that this example does NOT contain the traceback. Some logging services may find this version easier to digest, but it may not be very useful for debugging purposes unless you add the line and module information to the log records.

The last thing to cover is writing your logs to a file!

Logging to Disk

The `structlog` documentation doesn't talk much about logging to disk, but if you go digging, you will find there is a `WriteLoggerFactory()` that you can use to write to a file. However, this factory doesn't seem to work without using the `JSONRenderer()`.

Regardless, to see this in action, you will need to write more code. So open up your Python IDE and create a file called `structlog_file.py`. Then enter the following code:

```
# structlog_file.py

import logging
import structlog

from pathlib import Path


structlog.configure(
    processors=[
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.add_log_level,
        structlog.processors.JSONRenderer()
    ],
    wrapper_class=structlog.make_filtering_bound_logger(
        logging.INFO),
    logger_factory=structlog.WriteLoggerFactory(
            file=Path("app").with_suffix(
                ".log").open("wt")
            )
)
logger = structlog.get_logger()
logger.info("This is an info message")
```

You can pass a file handle to the `file` argument of the `WriteLoggerFactory()`, but this example uses a `pathlib.Path()` object to create the handle for you. The rest of the code is the same as in the earlier examples.

When you log, the output will go to the file rather than your console or terminal.

Wrapping Up

The `structlog` package is flexible and lets you do logging effectively. You don't need all the boilerplate that Python's `logging` API requires.

In this chapter, you covered the following:

- Installing `structlog`
- structlog's log levels
- Log formatting with structlog
- Updating the timestamp
- Serializing to JSON
- Logging exceptions with structlog
- Logging to disk

These topics only scratch the surface of what you can do with `structlog`. The documentation talks about adding filters, using built-in async methods, type hints, frameworks, recipes, and much, much more. Check it out if you'd like to learn more!

Afterword

Over the years as a software engineer, I have come to enjoy Python's `logging` module and appreciate logging in general. I do a lot of software testing and root cause analysis and logs have helped me solve many mysteries in my programs as well as the companies I have worked for.

Logs have also helped me discover who did what and when, which can be good teaching opportunities too. You can learn a lot about when the IT department pushes out updates to servers and causes your CI/CD system to crash, for example. Or you might discover that a root user is rebooting in a cron job. You can discover many interesting things in logs!

I hope that after reading this book, you will see not just the value of logging, but understand how to do it effectively with the Python programming language. Most software engineers who use Python also use the `logging` module. But I wanted to point out that there are a couple of great third party logging packages that you can try out too. Perhaps they will make logging even easier for you.

Thanks again for reading this book and I hope to hear how much you enjoyed it.

Mike