Recording relevant information during the execution of your program is a good practice as a Python developer when you want to gain a better understanding of your code. This practice is called **logging**, and it's a very useful tool for your programmer's toolbox. It can help you discover scenarios that you might not have thought of while developing.

These records are called **logs**, and they can serve as an extra set of eyes that are constantly looking at your application's flow. Logs can store information, like which user or IP accessed the application. If an error occurs, then logs may provide more insights than a stack trace by telling you the state of the program before the error and the line of code where it occurred.

Python provides a logging system as part of its standard library. You can add logging to your application with just a few lines of code.

In this tutorial, you'll learn how to:

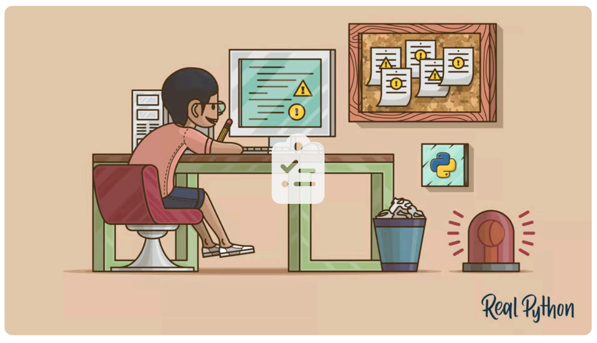
- Work with Python's `logging` **module**
- Set up a **basic logging configuration**
- Leverage **log levels**
- Style your log messages with **formatters**
- Redirect log records with **handlers**
- Define logging rules with **filters**

When you log useful data from the right places, you can debug errors, analyze the application's performance to plan for scaling, or look at usage patterns to plan for marketing.

You'll do the coding for this tutorial in the Python standard REPL. If you prefer Python files, then you'll find a full logging example as a script in the materials of this tutorial. You can download this script by clicking the link below:

Get Your Code: Click here to download the free sample code that you'll use to learn about logging in Python.

Take the Quiz: Test your knowledge with our interactive "Logging in Python" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Logging in Python

In this quiz, you'll test your understanding of Python's logging module. With this knowledge, you'll be able to add logging to your applications, which can help you debug errors and analyze performance.

If you commonly use Python's `print()` function to get information about the flow of your programs, then logging is the natural next step for you. This tutorial will guide you through creating your first logs and show you how to make logging grow with your projects.

Starting With Python's Logging Module

The `logging` module in Python's standard library is a ready-to-use, powerful module that's designed to meet the needs of beginners as well as enterprise teams.

Note: Since logs offer a variety of insights, the `logging` module is often used by other third-party Python libraries, too. Once you're more advanced in the practice of logging, you can integrate your log messages with the ones from those libraries to produce a homogeneous log for your application.

To leverage this versatility, it's a good idea to get a better understanding of how the `logging` module works under the hood. For example, you could take a stroll through the `logging` module's source code

The main component of the `logging` module is something called the **logger**. You can think of the logger as a reporter in your code that decides what to record, at what level of detail, and where to store or send these records.

Exploring the Root Logger

To get a first impression of how the `logging` module and a logger work, open the Python standard REPL and enter the code below:

Python

```
>>> import logging
>>> logging.warning("Remain calm!")
WARNING:root:Remain calm!
```

The output shows the severity level before each message along with `root`, which is the name the `logging` module gives to its default logger. This output shows the default format that can be configured to include things like a timestamp or other details.

In the example above, you're sending a message on the `root` logger. The **log level** of the message is `WARNING`. Log levels are an important aspect of logging. By default, there are five standard levels indicating the severity of events. Each has a corresponding function that can be used to log events at that level of severity.

Note: There's also a `NOTSET` log level, which you'll encounter later in this tutorial when you learn about custom logging handlers.

Here are the five default log levels, in order of increasing severity:

Log Level	Function	Description
`DEBUG`	`logging.debug()`	Provides detailed information that's valuable to you as a developer.
`INFO`	`logging.info()`	Provides general information about what's going on with your program.
`WARNING`	`logging.warning()`	Indicates that there's something you should look into.
`ERROR`	`logging.error()`	Alerts you to an unexpected problem that's occured in your program.
`CRITICAL`	`logging.critical()`	Tells you that a serious error has occurred and may have crashed your app.

The `logging` module provides you with a default logger that allows you to get started with logging without needing to do much configuration. However, the `logging` functions listed in the table above reveal a quirk that you may not expect:

Python

```
>>> logging.debug("This is a debug message")

>>> logging.info("This is an info message")

>>> logging.warning("This is a warning message")
WARNING:root:This is a warning message

>>> logging.error("This is an error message")
ERROR:root:This is an error message

>>> logging.critical("This is a critical message")
CRITICAL:root:This is a critical message
```

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

Adjusting the Log Level

To set up your basic logging configuration and adjust the log level, the `logging` module comes with a `basicConfig()` function. As a Python developer, this camel-cased function name may look unusual to you as it doesn't follow the PEP 8 naming conventions:

> "This is because it was adopted from Log4j, a logging utility in Java. It is a known issue in the package but by the time it was decided to add it to the standard library, it had already been adopted by users and changing it to meet PEP8 requirements would cause backwards compatibility issues." (Source)

Later in this tutorial, you'll learn about common parameters for `basicConfig()`. For now, you'll focus on the `level` parameter to set the log level of the `root` logger:

Python

```
>>> import logging

>>> logging.basicConfig(level=logging.DEBUG)
>>> logging.debug("This will get logged.")
DEBUG:root:This will get logged.
```

By using the `level` parameter, you can set what level of log messages you want to record. This can be done by passing one of the constants available in the class. You can use either the constant itself, its numeric value, or the string value as the `level` argument:

Constant	Numeric Value	String Value
`logging.DEBUG`	`10`	`DEBUG`
`logging.INFO`	`20`	`INFO`
`logging.WARNING`	`30`	`WARNING`
`logging.ERROR`	`40`	`ERROR`
`logging.CRITICAL`	`50`	`CRITICAL`

Setting a log level will enable all logging calls at the defined level and higher. For example, when you set the log level to `DEBUG`, all events at or above the `DEBUG` level will be logged.

Formatting the Output

By default, logs contain the log level, the logger's name, and the log message. That's good for a start. But you can enhance your logs with additional data by leveraging the `format` parameter of `basicConfig()`.

The `format` parameter accepts a string that can contain a number of predefined attributes. You can think of these attributes as placeholders that you format into the string. The default value of `format` looks like this:

Python

```
>>> import logging
>>> logging.basicConfig(format="%(levelname)s:%(name)s:%(message)s")
>>> logging.warning("Hello, Warning!")
WARNING:root:Hello, Warning!
```

This format is called `printf-style` string format. You may also find log formats with a dollar sign and curly braces (`${}`), which are related to the `string.Template()` class. If you're used to modern Python string formatting, then you're probably more familiar with using curly braces (`{}`) to format your strings.

You can define the style of your `format` string with the `style` parameter. The options for `style` are `"%"`, `"$"`, or `"{"`. When you provide a `style` argument, then your `format` string must match the targeted style. Otherwise, you'll receive a `ValueError`.

Note: Calling `basicConfig()` to configure the `root` logger only works if the `root` logger hasn't been configured before. All `logging` functions automatically call `basicConfig()` without arguments if `basicConfig()` has never been called. So, for example, once you call `logging.debug()`, you'll no longer be able to configure the `root` logger with `basicConfig()`.

Restart the REPL and start a logger with a different style format. Before trying out other attributes for your logs, stick with the default format structure from before:

Python

```
>>> import logging
>>> logging.basicConfig(format="{levelname}:{name}:{message}", style="{")
>>> logging.warning("Hello, Warning!")
WARNING:root:Hello, Warning!
```

As mentioned before, the `format` parameter accepts a string that can contain a number of predefined attributes. The ones you choose to use will depend on the insights that you want to get from your logs.

Besides the message text and the log level, it usually makes sense to also have a timestamp in the log record. A timestamp can give you the exact time the program sent the log message. This can help you monitor code performance or notice patterns around when some errors occur.

To add a timestamp to your logs, you can use the `asctime` attribute in your `format` string of `baseConfig()`. By default, `asctime` also shows you milliseconds. If you don't need to be that exact or if you want to customize the timestamp, then you must add `datefmt` to your `baseConfig()` call:

Python

```
>>> import logging
>>> logging.basicConfig(
...     format="{asctime} - {levelname} - {message}",
...     style="{",
...     datefmt="%Y-%m-%d %H:%M",
... )

>>> logging.error("Something went wrong!")
2024-07-22 09:26 - ERROR - Something went wrong!
```

In the example above, you prefix your logs with a timestamp. The **directives** you use to format the timestamp in the `datefmt` string are year (`%Y`), month (`%m`), day (`%d`), hour (`%H`), and minutes (`%M`). For an overview of all the date directives that you can embed into the format string, you can have a look at the `time.strftime()` documentation.

Additional information, like the time of your log message, becomes even more important when you want to keep a log of incidents over time or when you want to persist your logs in an external file.

Logging to a File

So far, you've logged the messages into your console. But if you want to archive your logs, then it's a good idea to save log messages in a file that grows over time.

To save your logs in a file, you can set up your logger's `baseConfig()` with the `filename` argument. Just like when working with files in Python and using the `open()` function, you must provide a filepath. It's also good practice to set an encoding and the mode the file should be opened in:

Python

```
>>> import logging
>>> logging.basicConfig(
...     filename="app.log",
...     encoding="utf-8",
...     filemode="a",
...     format="{asctime} - {levelname} - {message}",
...     style="{",
...     datefmt="%Y-%m-%d %H:%M",
... )

>>> logging.warning("Save me!")
```

With the configuration above, you save your logs in an `app.log` file instead of showing the messages in the console. To add all logs to the file and not overwrite any existing logs, you set `filemode` to `a`, which is short for "append".

The `app.log` is a basic text file that you can open in any text editor:

Text `app.log`

```
2024-07-22 09:55 - WARNING - Save me!
```

Besides formatting your log records, it can also be a good idea to archive your logs in date-formatted folders and adjust the names of the log files. You can even get creative and format your log records in such a way that you can save your logs as CSV files and create your own programs to practice parsing CSV files.

Displaying Variable Data

In most cases, you want to include dynamic information from your application in the logs. You've seen that the logging functions take a string as an argument. By leveraging Python's f-strings, you can create verbose debug messages that contain variable information:

Python

```
>>> import logging
>>> logging.basicConfig(
...     format="{asctime} - {levelname} - {message}",
...     style="{",
...     datefmt="%Y-%m-%d %H:%M",
...     level=logging.DEBUG,
... )

>>> name = "Samara"
>>> logging.debug(f"{name=}")
2024-07-22 14:49 - DEBUG - name='Samara'
```

First, you configure your logger and set the debug level to `DEBUG` to show the debug messages. Then, you define a variable `name` with the value `"Samara"`. Using self-documenting expressions, you can interpolate a variable name and its value in an f-string by appending an equal sign (`=`) to the variable name.

Note: F-strings are eagerly evaluated. That means that they are interpolated even if the log message is never handled. If you're interpolating a lot of lower level log messages, you should consider using the modulo operator (`%`) for interpolation instead of f-strings. This style is supported by `logging` natively, such that you can write code like the following:

Python

```
>>> import logging
>>> logging.basicConfig(
...     format="%(asctime)s - %(levelname)s - %(message)s",
...     style="%",
...     datefmt="%Y-%m-%d %H:%M",
...     level=logging.DEBUG,
... )

>>> name = "Samara"
>>> logging.debug("name=%s", name)
2024-07-22 14:51 - DEBUG - name=Samara
```

In this case, you use `%s` as a placeholder for the string referred to by `name`. Note that `name` is passed as a parameter to `logging.debug()`. If the debug message is not handled, then Python will never do the interpolation.

Using the `logging` module to see the current value of variables is a good first step when debugging your application. If you want to get even more insights about your code, then it can make sense to send exceptions to your logger.

Capturing Stack Traces

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

Python

```
>>> import logging
>>> logging.basicConfig(
...     filename="app.log",
...     encoding="utf-8",
...     filemode="a",
...     format="{asctime} - {levelname} - {message}",
...     style="{",
...     datefmt="%Y-%m-%d %H:%M",
... )

>>> donuts = 5
>>> guests = 0
>>> try:
...     donuts_per_guest = donuts / guests
... except ZeroDivisionError:
...     logging.error("DonutCalculationError", exc_info=True)
...
```

Since you're logging into the `app.log` file, you can keep track of stack traces in the file:

Text app.log

```
2024-07-22 15:04 - ERROR - DonutCalculationError
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

If `exc_info` isn't set to `True`, the output of the above program wouldn't tell you anything about the exception, which, in a real-world scenario, might not be as simple as a `ZeroDivisionError`.

Since logging errors is such a common task, `logging` comes with a function that can save you some typing. If you're logging from an exception handler, which you'll learn more about later, then you can use the `logging.exception()` function. This function logs a message with the level `ERROR` and adds exception information to the message.

Here's an example of how you'll get the same output as above using `logging.exception()`:

Python

```
>>> try:
...     donuts_per_guest = donuts / guests
... except ZeroDivisionError:
...     logging.exception("DonutCalculationError")
...
```

Calling `logging.exception()` is like calling `logging.error(exc_info=True)`. Since the `logging.exception()` function always dumps exception information, you should only call `logging.exception()` from an exception handler.

When you use `logging.exception()`, it shows a log at the level of `ERROR`. If you don't want that, you can call any of the other logging functions from `debug()` to `critical()` and pass the `exc_info` parameter as `True`.

Creating a Custom Logger

So far, you've seen the default logger named `root`, which is used by the `logging` module whenever functions like `logging.debug()`, `logging.warning()`, and so on are called. Calling the default logger directly is a handy way to get a first impression of how logging works.

The downside of working with the `root` logger directly is that the configuration can be cumbersome as you're relying on a single `basicConfig()`. For bigger projects, you'll need more flexibility for your logging needs.

Generally, it's a good idea to define your own custom logger. You can do so by creating an object of the `Logger` class, which you can find in the `logging` module.

Instantiating Your Logger

You can instantiate a `Logger` class by calling the `logging.getLogger()` function and providing a name for your logger:

Python

```
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> logger.warning("Look at my logger!")
Look at my logger!
```

While you could use any string as the name, it's good practice to pass `__name__` as the name parameter. That way, your logger's name is always the module's name in the Python package namespace.

When you call `logger.warn()`, you notice that you don't see any additional logging information, such as the logger's name or the log level. To format the log, you may be tempted to call `.basicConfig()` on your custom logger. However, unlike the `root` logger, you can't configure a custom logger using `basicConfig()`. Instead, you have to configure your custom logger using handlers and formatters, which give you way more flexibility.

Using Handlers

Handlers come into the picture when you want to configure your own loggers. For example, when you want to send the log messages to different destinations like the standard output stream or a file.

Note: A logger that you create can have one or more handlers. That means you can send your logs to multiple places when they're generated.

Here's an example of adding two handlers to a custom logger. Start by importing `logging` to create your logger and then two handlers:

Python

```
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> console_handler = logging.StreamHandler()
>>> file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
```

The `StreamHandler` class will send your logs to the console. The `FileHandler` class writes your log records to a file. To define where and how you want to write the logs, you provide a file path, an opening mode, and the encoding.

After you've instantiated your handlers, you must add them to the logger. For this, you use the `.addHandler()` method:

Python

```
>>> logger.addHandler(console_handler)
>>> logger.addHandler(file_handler)
>>> logger.handlers
[
  <StreamHandler <stderr> (NOTSET)>,
  <FileHandler /Users/RealPython/Desktop/app.log (NOTSET)>
]
```

You can list all the handlers that a logger uses by looking at the `.handlers` property. In the example above, you can see the string representations of both handlers.

Besides the class name, the representation of the handlers shows where the log will be written. For the `StreamHandler`, the logs are written to the **standard error stream** (`stderr`) stream, which is the output channel that Python uses by default. For the `FileHandler`, you can see the location where your log records will be saved.

In the parentheses of the class representation, you'll see the log level of your handlers. Currently, the log level is `NOTSET`. As you might expect, `NOTSET` means that the log level for the logging handlers is not set, yet.

You'll revisit the log levels again later in the tutorial. For now, put your logging handler into action:

Python

```
>>> logger.warning("Watch out!")
Watch out!
```

When you call `logger.warning()`, then both handlers take over the message. The output from `StreamHandler` is displayed right away in your console. To see if `FileHandler` also did its job, open `app.log`:

Text `app.log`

```
Watch out!
```

Perfect! Both handlers work as expected. With one call of your custom logger, you can distribute your log messages into different directions using handlers.

The `logging` module comes with many handy handlers for specific purposes. For example, `RotatingFileHandler`, which creates a new log file once a file size limit is reached, or `TimedRotatingFileHandler`, with which you can create a new log file for defined intervals.

So far, the messages look a bit plain. As you learned before, one of the strengths of logging is that it can enrich your information with metadata like timestamps or log levels. This is where formatters come into play!

Adding Formatters to Your Handlers

Handlers send your logs to the output destination you define. With a **formatter**, you can control the output format by specifying a string format as you did before with the `format` argument of `logging.basicConfig()`.

Just like with handlers, you need to instantiate a class first before working with a formatter. For formatters, you use the `Formatter` class of the `logging` module.

To get a first impression of how a formatter works, you'll revisit the code from before and add `logging.Formatter()` to the mix. Start by adding the formatter to `StreamHandler` and try it out:

Python

```
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> console_handler = logging.StreamHandler()
>>> file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
>>> logger.addHandler(console_handler)
>>> logger.addHandler(file_handler)
>>> formatter = logging.Formatter(
...    "{asctime} - {levelname} - {message}",
...     style="{",
...     datefmt="%Y-%m-%d %H:%M",
... )

>>> console_handler.setFormatter(formatter)
>>> logger.warning("Stay calm!")
2024-07-22 15:58 - WARNING - Stay calm!
```

Here, you instantiate a formatter that shows a timestamp, the log level, and the log message. By calling `.setFormatter()` with `formatter` as an argument, you define the formatting of the handler you attach the formatter to.

Note: Unlike `.addHandler()`, which is a method of `Logger`, `.setFormatter()` is a method of `Handler`.

With both `console_handler` and `file_handler` added to `logger`, your `logger.warning()` call also ended up in `app.log`. But since you only set the formatter for `console_handler`, the log record in `app.log` remains unstyled.

By defining and setting different formatters for your handlers, you can control the amount of additional information that displays in your log messages. But you don't have to stop there! You can even show debug messages in your console but save more severe log levels in a file, or vice versa.

Setting the Log Levels of Custom Loggers

Similar to calling `logging.basicConfig()`, you can also set the log level in handlers. This is useful when you want to set multiple handlers for the same logger but want different severity levels for each.

For example, while developing an application, you may want logs with level `DEBUG` and higher to be logged to the console, but everything with level `WARNING` and above to be saved to a file.

Start by creating a custom logger and explore its default log level:

Python

```
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> logger.level
0

>>> logger
<Logger __main__ (WARNING)>

>>> logger.parent
<RootLogger root (WARNING)>
```

The default log level of your custom level is `0`, which stands for `NOTSET`. Still, the string representation of your logger shows the `WARNING` log level. That's because a custom logger inherits the log level of its parent logger if you haven't yet set the log level manually.

Besides having a look at the string representation of a logger, you can also call the `.getEffectiveLevel()` method:

Python

```
>>> logger.getEffectiveLevel()
30
```

The returned value of `.getEffectiveLevel()` is an integer that stands for the log level. Here's an overview of the numeric representations of log levels:

Numeric Value	Log Level
0	NOTSET
10	DEBUG
20	INFO
30	WARNING
40	ERROR
50	CRITICAL

You can use either the numeric value or a log level string to set the log level of your custom logger:

Python

```
>>> logger.setLevel(10)
>>> logger
<Logger __main__ (DEBUG)>

>>> logger.setLevel("INFO")
>>> logger
<Logger __main__ (INFO)>
```

You use the `.setLevel()` method to set the log level of your logger. Any handler that you add to the logger will recognize this log level:

Python

```
>>> formatter = logging.Formatter("{levelname} - {message}", style="{")
>>> console_handler = logging.StreamHandler()
>>> console_handler.setFormatter(formatter)
>>> logger.addHandler(console_handler)
>>> logger.debug("Just checking in!")
>>> logger.info("Just checking in, again!")
INFO - Just checking in, again!
```

Since the log level of `logger` is set to `INFO`, the `console_handler` that you added to `logger` doesn't show logs that have a log level lower than `INFO`.

You may argue that you haven't set the log level of `console_handler` yet, and you're right. At this point, `console_log` has the level `NOTSET`:

Python

```
>>> console_handler
<StreamHandler <stderr> (NOTSET)>
```

But even setting a handler's log level lower than the associated logger won't show messages below the logger's log level:

Python

```
>>> console_handler.setLevel("DEBUG")
>>> logger.debug("Just checking in!")
>>> console_handler
<StreamHandler <stderr> (DEBUG)>
```

The debug message still doesn't show, even though you allowed `console_log` to show log records for the level `DEBUG` and above. This behavior of your logger and its handlers can be confusing at first.

Note: You define the lowest allowed log level on the logger itself. Handlers can't show logs lower than the defined log level of the logger they're connected to.

With this behavior in mind, it can be helpful during development to set your logger's log level to `DEBUG` and let each handler decide their lowest log level:

Python

```
>>> import logging
>>> logger = logging.getLogger(__name__)
>>> logger.setLevel("DEBUG")
>>> formatter = logging.Formatter("{levelname} - {message}", style="{")

>>> console_handler = logging.StreamHandler()
>>> console_handler.setLevel("DEBUG")
>>> console_handler.setFormatter(formatter)
>>> logger.addHandler(console_handler)

>>> file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
>>> file_handler.setLevel("WARNING")
>>> file_handler.setFormatter(formatter)
>>> logger.addHandler(file_handler)

>>> logger.debug("Just checking in!")
DEBUG - Just checking in!

>>> logger.warning("Stay curious!")
WARNING - Stay curious!

>>> logger.error("Stay put!")
ERROR - Stay put!
```

In the example above, you set different log levels for `console_handler` and `file_handler`. Since the log level of `console_handler` is `DEBUG`, you can see all logs in the console. If you have a look at the `app.log` file that `file_handler` writes to, then you can verify that the `WARNING` and `ERROR` logs were saved to the file:

Text `app.log`

```
WARNING - Stay curious!
ERROR - Stay put!
```

By leveraging different log levels, you can control where you log information. When you do so, it's important to remember that handlers can never log levels below their logger's log level. Instead, handlers log any level above the set log level.

In other words, setting the log level will filter out any log messages below that level, and every log message at or above that level will be shown. If you only want to show a specific log level, then you can gain even more control by adding a filter to a handler.

Remove ads

Filtering Logs

If you're interested in the `WARNING` log records your Python program produces, you're probably also interested in more severe log levels like `ERROR` or even `CRITICAL`. So, most of the time, you'll be fine collecting all log records of a certain log level and above with specific handlers.

However, there are situations where it may make sense to handle messages for a specific log level differently. That's when a `Filter` can come in handy. The important part of the specifications for the `Filter` object is this:

> The filtering logic will check to see if the filter object has a `filter` attribute: if it does, it's assumed to be a `Filter` and its `filter()` method is called. Otherwise, it's assumed to be a callable and called with the record as the single parameter. The returned value should conform to that returned by `filter()`. (Source)

In other words, there are three approaches to creating filters for logging. You can create a:

1. **Subclass** of `logging.Filter()` and overwrite the `.filter()` method
2. **Class** that contains a `.filter()` method
3. **Callable** that resembles a `.filter()` method

For the subclass and class, `.filter()` must accept a log record and return a Boolean. Inside the method's body, it makes sense to define a conditional statement that checks the provided record.

The callable can be a basic function with one parameter for the log record that your handler passes in. The return value must be a Boolean. Using a callable is arguably the most convenient when creating basic filters, so you'll explore this approach further.

With the theory in place, it's time to put the logging filter into practice. The filter you'll create will make sure the handler that outputs logs into the console only shows `DEBUG` messages:

Python

```python
>>> import logging
>>> def show_only_debug(record):
...     return record.levelname == "DEBUG"
...

>>> logger = logging.getLogger(__name__)
>>> logger.setLevel("DEBUG")
>>> formatter = logging.Formatter("{levelname} - {message}", style="{")

>>> console_handler = logging.StreamHandler()
>>> console_handler.setLevel("DEBUG")
>>> console_handler.setFormatter(formatter)
>>> console_handler.addFilter(show_only_debug)
>>> logger.addHandler(console_handler)

>>> file_handler = logging.FileHandler("app.log", mode="a", encoding="utf-8")
>>> file_handler.setLevel("WARNING")
>>> file_handler.setFormatter(formatter)
>>> logger.addHandler(file_handler)

>>> logger.debug("Just checking in!")
DEBUG - Just checking in!

>>> logger.warning("Stay curious!")
>>> logger.error("Stay put!")
```

First, you create a callable named `show_only_debug()` with a `record` parameter. The passed in log record will be an instance of `LogRecord`.

In `show_only_debug`, you return `True` if the `.levelname` attribute of your log record is "DEBUG". Any log record that isn't on the `DEBUG` log level will return `False` and will not be shown by the handler you attach the filter to.

To add a filter to a handler, you use the `.addFilter()` method of the `Handler` class. The argument for `.addFilter()` must be a filter. In the code above, you're passing in a reference to `show_only_debug()` to attach the filter to `console_handler`.

For `console_handler`, you've set the log level to `DEBUG`. Without any other adjustments, the handler will show all log records of the `DEBUG` level and above. With your filter in place, you're surpressing higher log levels and only showing debug messages in the console. Since you didn't add a filter to `file_handler`, this handler happily logs records for the set log level and above.

Your custom loggers become highly customizable tools that can show exactly the output you want to see. A custom logger can be as basic as a root logger. But by combining handlers, formatters, and filters, you can make your custom loggers into an eloquent field reporter for your code.

Conclusion

The logging module is considered to be very flexible. Its design is very practical and should fit your use case out of the box. You can add basic logging to a small project, or you can go as far as creating your own custom logger that can grow with your coding project.

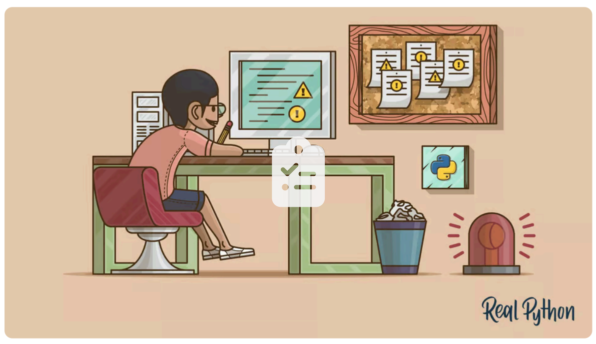
In this tutorial, you learned how to:

- Work with Python's **`logging` module**
- Set up a **basic logging configuration**
- Leverage **log levels**
- Style your log messages with **formatters**
- Redirect log records with **handlers**
- Define logging rules with **filters**

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

Get Your Code: Click here to download the free sample code that you'll use to learn about logging in Python.

Take the Quiz: Test your knowledge with our interactive "Logging in Python" quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

Logging in Python

In this quiz, you'll test your understanding of Python's logging module. With this knowledge, you'll be able to add logging to your applications, which can help you debug errors and analyze performance.

Mark as Completed Share

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

🐍 Python Tricks 💌