# Python For Large Applications

Aaron Maxwell

aaron@powerfulpython.com

# Contents

# Chapter 1

# Logging in Python

Logging is critical in many kinds of software. For long-running software systems, it enables continuous telemetry and reporting. And for *all* software, it can provide priceless information for troubleshooting and post-mortems. The bigger the application, the more important logging becomes. But even small scripts can benefit.

Python provides logging through the `logging` module. In my opinion, this module is one of the more technically impressive parts of Python's standard library. It's well-designed, flexible, thread-safe, and richly powerful. It's also complex, with many moving parts, making it hard to learn well. This chapter gets you over most of that learning curve, so you can fully benefit from what `logging` has to offer. The payoff is well worth it, and will serve you for years.

Broadly, there are two ways to use `logging`. One, which I'm calling the *basic interface*, is appropriate for scripts - meaning, Python programs that are small enough to fit in a single file. For more substantial applications, it's typically better to use *logger objects*, which give more flexible, centralized control, and access to logging hierarchies. We'll start with the former, to introduce key ideas.

## 1.1 The Basic Interface

Here's the easiest way to use Python's logging module:

```
import logging
logging.warning('Look out!')
```

Save this in a script and run it, and you'll see this printed to standard output:

```
WARNING:root:Look out!
```

You can do useful logging right away, by calling functions in the `logging` module itself. Notice you invoke `logging.warning()`, and the output line starts with `WARNING`. You can also call `logging.error()`, which gives a different prefix:

```
ERROR:root:Look out!
```

We say that `warning` and `error` are at different *message log levels*. You have a spectrum of log levels to choose from, in order of increasing severity:[1]

**debug** Detailed information, typically of interest only when diagnosing problems.

**info** Confirmation that things are working as expected.

**warning** An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.

**error** Due to a more serious problem, the software has not been able to perform some function.

**critical** A serious error, indicating that the program itself may be unable to continue running.

You use them all just like `logging.warning()` and `logging.error()`:

```
logging.debug("Small detail. Useful for troubleshooting.")
logging.info("This is informative.")
logging.warning("This is a warning message.")
logging.error("Uh oh. Something went wrong.")
logging.critical("We have a big problem!")
```

Each has a corresponding uppercased constant in the library (e.g., `logging.WARNING` for `logging.warning()`). You use these when defining the *log level threshold*. Run the above, and here is the output:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

[1] These beautifully crisp descriptions, which I cannot improve upon, are taken from https://docs.python.org/-3/howto/logging.html .

Where did the debug and info messages go? As it turns out, the default logging threshold is `logging.WARNING`, which means only messages of that severity or greater are actually generated; the others are ignored completely. The order matters in the list above; debug is considered strictly less severe than `info`, and so on. Change the log level threshold using the `basicConfig` function:

```
logging.basicConfig(level=logging.INFO)
logging.info("This is informative.")
logging.error("Uh oh. Something went wrong.")
```

Run this new program, and the `INFO` message gets printed:

```
INFO:root:This is informative.
ERROR:root:Uh oh. Something went wrong.
```

Again, the order is `debug()`, `info()`, `warning()`, `error()` and `critical()`, from lowest to highest severity. When we set the log level threshold, we declare that we only want to see messages of that level or higher. Messages of a lower level are not printed. When you set `level` to `logging.DEBUG`, you see everything; set it to `logging.CRITICAL`, and you only see critical messages, and so on.

The phrase "log level" means two different things, depending on context. It can mean the severity of a message, which you set by choosing which of the functions to use - `logging.warning()`, etc. Or it can mean the threshold for ignoring messages, which is signaled by the constants: `logging.WARNING`, etc.

You can also use the constants in the more general `logging.log` function - for example, a debug message:

```
logging.log(logging.DEBUG,
    "Small detail. Useful for troubleshooting.")
logging.log(logging.INFO, "This is informative.")
logging.log(logging.WARNING, "This is a warning message.")
logging.log(logging.ERROR, "Uh oh. Something went wrong.")
logging.log(logging.CRITICAL, "We have a big problem!")
```

This lets you modify the log level dynamically, at runtime:

```
def log_results(message, level=logging.INFO):
    logging.log(level, "Results: " + message)
```

### 1.1.1 Why do we have log levels?

If you haven't worked with similar logging systems before, you may wonder why we have different log levels, and why you'd want to control the filtering threshold. It's easiest to see this if you've written Python scripts that include a number of `print()` statements - including some useful for diagnosis when something goes wrong, but a distraction when everything is working fine.

The fact is, some of those `print()` statements are more important than others. Some indicate mission-critical problems you always want to know about - possibly to the point of waking up an engineer, so they can deploy a fix immediately. Some are important, but can wait until the next work day - and you definitely do NOT want to wake anyone up for that. Some are details which may have been important in the past, and might be in the future, so you don't want to remove them; in the meantime, they are just line noise.

Having log levels solves all these problems. As you develop and evolve your code over time, you continually add new logging statements of the appropriate severity. You now even have the freedom to be proactive. With "logging" via `print()`, each log statement has a cost - certainly in signal-to-noise ratio, and also potentially in performance. So you might debate whether to include that print statement at all. But with `logging`, you can insert `info` messages, for example, to log certain events occurring as they should. In development, those INFO messages can be very useful to verify certain things are happening, so you can modify the log level to produce them. On production, you may not want to have them cluttering up the logs, so you just set the threshold higher. Or if you are doing some kind of monitoring on production, and temporarily need that information, you can adjust the log level threshold to output those messages; when you are finished, you can adjust it back to exclude them again.

When troubleshooting, you can liberally introduce `debug`-level statements to provide extra detailed statements. When done, you can just adjust the log level to turn them off. You can leave them in the code without cost, eliminating any risk of introducing more bugs when you go through and remove them. This also leaves them available if they are needed in the future.

The log level symbols are actually set to integers. You can theoretically use these numbers instead, or even define your own log levels that are (for example) a third of the way between `WARNING` and `ERROR`. In normal practice, it's best to use the predefined logging levels. Doing otherwise makes your code harder to read and maintain, and isn't worthwhile unless you have a compelling reason.

For reference, the numbers are 50 for `CRITICAL`, 40 for `ERROR`, 30 for `WARNING`, 20 for `INFO`,

and 10 for DEBUG. So when you set the log level threshold, it's actually setting a number. The only log messages emitted are those with a level greater than or equal to that number.

## 1.2 Configuring The Basic Interface

You saw above you can change the loglevel threshold by calling a function called basicConfig:

```
logging.basicConfig(level=logging.INFO)
logging.debug("You won't see this message!")
logging.error("But you will see this one.")
```

If you use it at all, basicConfig must be called exactly once, and it must happen before the first logging event. (Meaning, before the first call to debug(), or warning(), etc.) Additionally, if your program has several threads, it must be called from the main thread - and *only* the main thread. [2]

You already met one of the configuration options, level. This is set to the log level threshold, and is one of DEBUG, INFO, WARNING, ERROR, or CRITICAL. Some of the other options include:

**filename** Write log messages to the given file, rather than stderr.

**filemode** Set to "a" to append to the log file (the default), or "w" to overwrite.

**format** The format of log records.

**level** The log level threshold, described above.

By default, log messages are written to standard error. You can also write them to a file, one per line, to easily read later. Do this by setting filename to the log file path. By default it appends log messages, meaning that it will only add to the end of the file if it isn't empty. If you'd rather the file be emptied before the first log message, set filemode to "w". Be careful about doing that, of course, because you can easily lose old log messages if the application restarts:

```
# Wipes out previous log entries when program restarts
logging.basicConfig(filename="log.txt", filemode="w")
logging.error("oops")
```

[2] These restrictions aren't in place for logger objects, described later.

The other valid value is "a", for append - that's the default, and probably will serve you better in production. "w" can be useful during development, though.

`format` defines what chunks of information the final log record will include, and how they are laid out. These chunks are called "attributes" in the `logging` module docs. One of these attributes is the actual log message - the string you pass when you call `logging.warning()`, and so on. Often you will want to include other attributes as well. Consider the kind of log record we saw above:

```
WARNING:root:Collision imminent
```

This record has three attributes, separated by colons. First is the log level name; last is the actual string message you pass when you call `logging.warning()`. (In the middle is the name of the underlying logger object. `basicConfig` uses a logger called "root"; we'll learn more about that later.)

You specify the layout you want by setting `format` to a string that defines certain named fields, according to percent-style formatting. Three of them are `levelname`, the log level; `message`, the message string passed to the logging function; and `name`, the name of the underlying logger. Here's an example:

```
logging.basicConfig(
    format="Log level: %(levelname)s, msg: %(message)s")
logging.warning("Collision imminent")
```

If you run this as a program, you get the following output:

```
Log level: WARNING, msg: Collision imminent
```

It turns out the default formatting string is

```
%(levelname)s:%(name)s:%(message)s
```

You indicate named fields in percent-formatting by `%(FIELDNAME)X`, where "X" is a type code: `s` for string, `d` for integer (decimal), and `f` for floating-point.

Many other attributes are provided, if you want to include them. Here's a select few from the full list:[3]

[3] https://docs.python.org/3/library/logging.html#logrecord-attributes

| Attribute | Format | Description |
|---|---|---|
| asctime | `%(asctime)s` | Human-readable date/time |
| funcName | `%(funcName)s` | Name of function containing the logging call |
| lineno | `%(lineno)d` | The line number of the logging call |
| message | `%(message)s` | The log message |
| pathname | `%(pathname)s` | Full pathname of the source file of the logging call |
| levelname | `%(levelname)s` | Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL') |
| name | `%(name)s` | The logger's name |

You might be wondering why log record format strings use Python 2's percent-formatting style, when everything else in Python 3 uses the newer, brace-style string formatting. As it turns out, the conversion was attempted, but backwards-compatibility reasons made percent-formatting the only practical choice for the logging module, even after the Python 3 reboot.

If you want to use the newer string formatting badly enough, there are things you can do - there's even a standard recipe.[4] But doing so is complicated enough that it may not be worth the effort, and it won't help with legacy code. I recommend you simply cooperate with the situation, and use percent formatting with your Python logging.

## 1.3 Passing Arguments

You often want to include some kind of runtime data in the logged message. When you construct the message to log, specify the final log message like this:

```
num_fruits = 14
fruit_name = "oranges"
logging.info(
    "We ate %d of your %s. Thanks!",
    num_fruits, fruit_name)
```

The output:

[4] https://docs.python.org/3/howto/logging-cookbook.html#use-of-alternative-formatting-styles

```
INFO:root:We ate 14 of your oranges. Thanks!
```

We call `info` with three parameters. First is the format string; the second and third are arguments. The general form is

```
logging.info(format, *args)
```

You can pass zero or more arguments, so long as each has a field in the format string:

```
logging.info("%s, %s, %s, %s, %s, %s and %s",
    "Doc", "Happy", "Sneezy", "Bashful",
    "Dopey", "Sleepy", "Grumpy")
```

You *must* resist the obvious temptation to format the string fully, and pass that to the logging function:

```
num_fruits = 14
fruit_name = "oranges"
logging.warning(
    "Don't do this: %d %s" % (num_fruits, fruit_name))
logging.warning(
    "Or even this: {:d} {:s}".format(
    num_fruits, fruit_name))
```

This works, of course, in the sense that you will get correct log messages. However, it's wasteful, and surrenders important benefits `logging` normally provides. Remember: when the line of code with the log message is executed, it may not actually trigger a log event. If the log level threshold is higher than the message itself, the line does nothing. In that case, there is no reason to format the string.

In the first form, the string is formatted if and only if a log event actually happens, so that's fine. But if you format the string yourself, it's *always* formatted. That takes up system memory and CPU cycles even if no logging takes place. If the code path with the logging call is only executed occasionally, that's not a big deal. But it impairs the program when a debug message is logged in the middle of a tight loop. When you originally code the line, you never really know where it might migrate in the future, or who will call your function in ways you never imagined.

So just use the supported form, where the first argument is the format string, and subsequent arguments are the parameters for it. You can also use named fields, by passing a dictionary as the second argument:

```
fruit_info = {"count": 14, "name": "oranges"}
logging.info(
    "We ate %(count)d of your %(name)s. Thanks!",
    fruit_info)
```

## 1.4 Beyond Basic: Loggers

The basic interface is simple and easy to set up. It works well in single-file scripts. Larger Python applications tend to have different logging needs, however. `logging` meets these needs through a richer interface, called *logger objects* - or simply, *loggers*.

Actually, you have been using a logger object all along: when you call `logging.warning()` (or the other log functions), they convey messages through what is called the *root logger* - the primary, default logger object. This is why the word "root" shows in some example output.

`logger.basicConfig` operates on this root logger. You can fetch the actual root logger object by calling `logging.getLogger`:

```
>>> logger = logging.getLogger()
>>> logger.name
'root'
```

As you can see, it knows its name is "root". Logger objects have all the same functions (methods, actually) the `logging` module itself has:

```
import logging
logger = logging.getLogger()
logger.debug("Small detail. Useful for troubleshooting.")
logger.info("This is informative.")
logger.warning("This is a warning message.")
logger.error("Uh oh. Something went wrong.")
logger.critical("We have a big problem!")
```

Save this in a file and run it, and you'll see the following output:

```
This is a warning message.
Uh oh. Something went wrong.
We have a big problem!
```

This is different from what we saw with `basicConfig`, which printed out this instead:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

At this point, we've taken steps backward compared to `basicConfig`. Not only is the log message unadorned by the log level, or anything else useful. The log level threshold is hard-coded to `logging.WARNING`, with no way to change it. The logging output will be written to standard error, and no where else, regardless of where you actually need it to go.

Let's take inventory of what we want to control here. We want to choose our log record format. And further, we want to be able to control the log level threshold, and write messages to different streams and destinations. You do this with a tool called *handlers*.

## 1.5 Log Destinations: Handlers and Streams

By default, loggers write to standard error. You can select a different destination - or even *several* destinations - for each log record:

- You can write log records to a file. Very common.
- You can, while writing records to that file, *also* parrot it to stderr.
- Or to stdout. Or both.
- You can simultaneously log messages to two different files.
- In fact, you can log (say) `INFO` and higher messages to one file, and `ERROR` and higher to another.
- You can write log records to a remote log server, accessed via a REST HTTP API.
- Mix and match all the above, and more.
- And you can set a different, custom log format for each destination.

This is all managed through what are called *handlers*. In Python logging, a handler's job is to take a log record, and make sure it gets recorded in the appropriate destination. That destination can be a file; a stream like stderr or stdout; or something more abstract, like inserting into a queue, or transmitting via an RPC or HTTP call.

By default, logger objects don't have any handlers. You can verify this using the `hasHandlers` method:

```
>>> logger = logging.getLogger()
>>> logger.hasHandlers()
False
```

With no handler, a logger has the following behavior:

- Messages are written to stderr.
- Only the message is written, nothing else. There's no way to add fields or otherwise modify it.
- The log level threshold is `logging.WARNING`. There is no way to change that.

To change this, your first step is to create a handler. Nearly all logger objects you ever use will have custom handlers. Let's see how to create a simple handler that writes messages to a file, called "log.txt".

```
import logging
logger = logging.getLogger()
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

The `logging` module provides a class called `FileHandler`. It takes a file path argument, and will write log records into that file, one per line. When you run this code, `log.txt` will be created (if it doesn't already exist), and will contain the string "Boo!" followed by a newline. (If `log.txt` did exist already, the logged message would be *appended* to the end of the file.)

But "A little detail" is not written, because it's below the default logger threshold of `WARNING`. We change that by calling a method named `setLevel` on the logger object:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

This writes the following in "log.txt":

```
A little detail
Boo!
```

Confusingly, you can call `setLevel` on a logger with no handlers, *but it has no effect*:

```
# Doing it wrong:
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) # No effect.
logger.debug("This won't work :(")
```

To change the threshold from the default of `logging.WARNING`, you must both add a handler, *and* change the logger's level.

What if you want to log to stdout? Do that with a `StreamHandler`:

```
import logging
import sys
logger = logging.getLogger()
out_handler = logging.StreamHandler(sys.stdout)
logger.addHandler(out_handler)
logger.warning("Boo!")
```

If you save this in a file and run it, you'll get "Boo!" on standard output. Notice that `logging.StreamHandler` takes `sys.stdout` as its argument. You can create a `StreamHandler` without an argument too, in which case it will write its records to standard error:

```
import logging
logger = logging.getLogger()
# Same as StreamHandler(sys.stderr)
stderr_handler = logging.StreamHandler()
logger.addHandler(stderr_handler)
logger.warning("This goes to standard error")
```

In fact, you can pass any file-like object; The object just needs to define compatible `write` and `flush` methods. Theoretically, you could even log to a file by creating a handler like `StreamHandler(open("log.txt", "a"))` - but in that case, it's better to use a `FileHandler`, so it can manage opening and closing the file.

When creating a handler, your needs are nearly always met by either `StreamHandler` or `FileHandler`. There are other predefined handlers, too, useful when logging to certain specialized destinations:

- `WatchedFileHandler` and `RotatingFileHandler`, for logging to rotated log files
- `SocketHandler` and `DatagramHandler` for logging over network sockets
- `HTTPHandler` for logging over an HTTP REST interface
- `QueueHandler` and `QueueListener` for queuing log records across thread and process boundaries

See the official docs[5] for more details.

## 1.6 Logging to Multiple Destinations

Suppose you want your long-running application to log all messages to a file, including debug-level records. At the same time, you want warnings, errors, and criticals logged to the console. How do you do this?

We've given you part of the answer already. A single logger object can have multiple handlers: all you have to do is call `addHandler` multiple times, passing a different handler object for each. For example, here is how you parrot all log messages to the console (via standard error) and also to a file:

```
import logging
logger = logging.getLogger()
# Remember, StreamHandler defaults to using sys.stderr
console_handler = logging.StreamHandler()
logger.addHandler(console_handler)
# Now the file handler:
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)
logger.warning(
    "This goes to both the console, AND log.txt.")
```

[5] https://docs.python.org/3/library/logging.handlers.html

This is combining what we learned above. We create two handlers - a `StreamHandler` named `console_handler`, and a `FileHandler` named `logfile_handler` - and add both to the same logger (via `addHandler`). That's all you need to log to multiple destinations in parallel. Sure enough, if you save the above in a script and run it, you'll find the messages are both written into "log.txt", as well as printed on the console (through standard error).

We aren't done, though. How do we make it so every record is written in the log file, but only those of `logging.WARNING` or higher get sent to the console screen? Do this by setting log level thresholds for both the logger object and the individual handlers. Both logger objects and handlers have a method called `setLevel`, taking a log level threshold as an argument:

```
my_logger.setLevel(logging.DEBUG)
my_handler.setLevel(logging.INFO)
```

If you set the level for a logger, but not its handlers, the handlers inherit from the logger:

```
my_logger.setLevel(logging.ERROR)
my_logger.addHandler(my_handler)
my_logger.error("This message is emitted by my_handler.")
my_logger.debug("But this message will not.")
```

And you can override that at the handler level. Here, I create two handlers. One handler inherits its threshold from the logger, while the other does its own thing:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

verbose_handler = logging.FileHandler("verbose.txt")
logger.addHandler(verbose_handler)

terse_handler = logging.FileHandler("terse.txt")
terse_handler.setLevel(logging.WARNING)
logger.addHandler(terse_handler)

logger.debug("This message appears in verbose.txt ONLY.")
logger.warning("And this message appears in both files.")
```

There's a caveat, though: a handler can only make itself *more* selective than its logger, not less. If the logger chooses a threshold of `logger.DEBUG`, its handler can choose a threshold of `logger.INFO`, or `logger.ERROR`, and so on. But if the logger defines a strict threshold -

say, `logger.INFO` - an individual handler cannot choose a lower one, like `logger.DEBUG`. So something like this won't work:

```
# This doesn't quite work...
import logging
my_logger = logging.getLogger()
my_logger.setLevel(logging.INFO)
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
my_logger.debug("No one will ever see this message :(")
```

There's a subtle corollary of this. By default, a logger object's threshold is set to `logger.WARNING`. So if you don't set the logger object's log level at all, it implicitly censors all handlers:

```
import logging
my_logger = logging.getLogger()
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
# No one will see this message either.
my_logger.debug(
    "Because anything under WARNING gets filtered.")
```

The logger object's default log level is not always permissive enough for all handlers you might want to define. So you will generally want to start by setting the logger object to the lowest threshold needed by any log-record destination, and tighten that threshold for each handler as needed.

Bringing this all together, we can now accomplish what we originally wanted - to verbosely log everything into a log file, while duplicating only the more interesting messages onto the console:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
# Warnings and higher only on the console.
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)
logger.addHandler(console_handler)
# But allow everything to into the log file.
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)

logger.warning(
    "This goes to both the console, AND into log.txt.")
logger.debug("While this only goes to the file.")
```

Add as many handlers as you want. Each can have different log levels. You can log to many different destinations, using the different built-in handler types mentioned above. If those don't do what you need, implement your own subclass of `logging.Handler` and use that.

## 1.7 Record Layout with Formatters

We haven't covered one important detail. So far, we've only shown you how to create logger objects that will write just the log message and nothing else. At the very least, you probably want to annotate that with the log level. You may also want to insert the time, or some other information. How do you do that?

The answer is to use a *formatter*. A formatter converts a log record into something that is recorded in the handler's destination. That's an abstract way of saying it; more simply, a typical formatter just converts the record into a usefully-formatted string. That string contains the actual log message, as well as the other fields you care about.

The procedure is to create a `Formatter` object, then associate with a handler (using the latter's `setHandler` method). Creating a formatter is easy - it normally takes just one argument, the format string:

```
import logging
my_handler = logging.StreamHandler()
fmt = logging.Formatter("My message is: %(message)s")
my_handler.setFormatter(fmt)
my_logger = logging.getLogger()
my_logger.addHandler(my_handler)
my_logger.warning("WAKE UP!!")
```

If you run this in a script, the output will be:

```
My message is this: WAKE UP!!
```

Notice the attribute for the message, `%(message)s`, included in the string. This is just a normal formatting string, in the older, percent-formatting style. It's exactly equivalent to using the `format` argument when you call `basicConfig`. For this reason, you can use the same attributes, arranged however you like - here's the attribute table again, distilled from the full official list:[6]

| Attribute | Format | Description |
|---|---|---|
| asctime | `%(asctime)s` | Human-readable date/time |
| funcName | `%(funcName)s` | Name of function containing the logging call |
| lineno | `%(lineno)d` | The line number of the logging call |
| message | `%(message)s` | The log message |
| pathname | `%(pathname)s` | Full pathname of the source file of the logging call |
| levelname | `%(levelname)s` | Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL') |
| name | `%(name)s` | The logger's name |

[6] https://docs.python.org/3/library/logging.html#logrecord-attributes

# Chapter 2

# String Formatting

The situation with string formatting is complicated.

Once upon a time, Python introduced *percent formatting*. It uses `"%"` as a binary operator to render strings:

```
>>> drink = "coffee"
>>> price = 2.5
>>> message = "This %s costs $%.2f." % (drink, price)
>>> print(message)
This coffee costs $2.50.
```

Later in Python 2's history, a different style was introduced, called simply *string formatting* (yes, that's the official name). Its very different syntax makes any Python string a potential template, inserting values through the `str.format()` method.

```
>>> template = "This {} costs ${:.2f}."
>>> print(template.format(drink, price))
This coffee costs $2.50.
```

Python 3.6 introduces a third option, called *f-strings*. This lets you write literal strings, prefixed with an "f" character, interpolating values from the immediate context:

```
>>> message = f"This {drink} costs ${price:.02f}."
>>> print(message)
This coffee costs $2.50.
```

So… which do you use? Here's my guidance in a nutshell:

- Go ahead and master `str.format()` now. Everything you learn transfers entirely to f-strings, and you'll sometimes want to use `str.format()` even in cutting-edge versions of Python.
- Prefer f-strings when working in a codebase that supports it - meaning, all developers and end-users of the code base are certain to have Python 3.6 or later.
- Until then, prefer `str.format()`.
- Exception: for the `logging` module, use percent-formatting, even if you're otherwise using f-strings.
- Aside from `logging`, don't use percent-formatting unless legacy reasons force you to.

"Which should I use?" is a separate question from "which should a Python book use for its code examples?" As you've noticed, I am using `str.format()` throughout this book. That's because all modern Python versions support it, so I know everyone reading this book can use it.

Someday, when Python versions before 3.6 are a distant memory, there will be no reason not to use f-strings. But when that happens, `str.format()` will still be important. There are string formatting situations where f-strings are awkward at best, and `str.format()` is well suited. In the meantime, there is a lot of Python code out there using `str.format()`, which you'll need to be able to read and understand. Hence, I'm choosing to focus on `str.format()` in this chapter. Conveniently, this also teaches you much about f-strings; they are more similar than different, because the formatting codes are nearly identical. `str.format()` is also the only practical choice for most people reading this, and will be for years still.

You might wonder if the old percent-formatting has any place in modern Python. In fact, it does, due to the `logging` module. As you'll read in its chapter, this important module is built on percent-formatting in a deep way. It's possible to use `str.format()` in new logging code, but requires special steps; and legacy logging code cannot be safely converted in an automated way. I recommend you just cooperate with the situation, and use percent-formatting for your log messages.

For those interested, this chapter ends with sections briefly explaining f-strings and percent-formatting. For now, we'll focus on `str.format()`. While reading, I highly recommend you have a Python interpreter prompt open, typing in the examples as you go along. The goal is to make its expressive power automatic and easy for you to use, so that it's *mentally* available to you... giving you the easy ability to use it in the future, without digging into the reference docs. Most people never master it to this threshold, effectively denying them most of the benefits of this rich tool. You won't have that problem.

## 2.1 Replacing Fields

`str.format()` lets you start simple, leveraging more complex extensions as needed. You start by creating a format string. This is just a regular string, and acts as a kind of template. It contains, among other text, one or more *replacement fields*. These are simply pairs of opening and closing curly braces: `"Good {}, my friend"`. You then invoke the `format` method on that string, passing in one argument for each replacement field:

```
>>> "Good {}, my friend".format("morning")
'Good morning, my friend'
>>> "Good {}, my friend".format("afternoon")
'Good afternoon, my friend'
>>> offer = "Give me {} dollars and I'll give you a {}."
>>> offer.format(2, "cheeseburger")
"Give me 2 dollars and I'll give you a cheeseburger."
>>> offer.format(7, "nice shoulder rub")
"Give me 7 dollars and I'll give you a nice shoulder rub."
```

(If possible, type these examples in an interpreter as you go along, so you learn them deeply.) `.format()` is a method returning a new string; the format string itself is not modified.

Notice how fields line up by position, and no type information is needed. The integer 2 and the string "cheeseburger" are both inserted without complaint. We'll see how to specify more precise types for the fields later.

Within the curly braces of the replacement field, you can specify numbers starting at 0. These reference the positions of the arguments passed to `format`, and allow you to repeat fields:

```
>>> "{0} is {1}; {1}, {0}".format("truth", "beauty")
'truth is beauty; beauty, truth'
```

You can also reference fields by a name, and pass the fields as key-value pairs to `format`:

```
>>> "Good {when}, {user}!".format(
        when="morning", user="John")
'Good morning, John!'
```

The arguments to `format()` don't actually have to be strings: they can be objects or lists. Reference within them as you normally would, within the curly braces:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> point = Point(3, 7)
>>> 'The coordinates are {point.x}, {point.y}'.format(
        point=point)
'The coordinates are 3, 7'
>>> 'The coordinates are {0.x}, {0.y}'.format(point)
'The coordinates are 3, 7'
```

Notice the difference in how you use the point with a named field (the first format call), versus a numbered field (the second). Here's how it looks with a list:

```
>>> params = ["morning", "user"]
>>> "Good {params[0]}, {params[1]}!".format(params=params)
'Good morning, user!'
>>> "Good {0[0]}, {0[1]}!".format(params)
'Good morning, user!'
```

You can do the same thing with dictionaries too, though there is one subtle quirk - can you spot it?

```
>>> params = {"when": "morning", "user":"John"}
>>> "Good {0[when]}, {0[user]}!".format(params)
'Good morning, John!'
```

See it? The key `when` in `{0[when]}` does not have quotation marks around it! In fact, if you do put them in, you get an error. This quirk was intentionally put in, to make it easier to reference keys within a string that is bounded by quotes already.

## 2.2 Number Formats (and "Format Specs")

Now that you know how to substitute values into different replacement fields (i.e., pairs of curly braces), you may next want to format a field as a number. Do this by inserting a colon between the curly braces, followed by one or more descriptive characters. For example, use ":d" to format as an integer, and ":f" to format as a floating-point number. Here's how it works:

```
>>> 'The magic number is {:d}'.format(42)
'The magic number is 42'
>>> 'The magic number is actually {:f}'.format(42)
'The magic number is actually 42.000000'
>>> 'But this will cause an error: {:d}'.format("foo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'd' for object of type 'str'
```

The number 42 is rendered as either `42` or `42.000000`, depending on whether the replacement field is `{:d}` or `{:f}`. And when we try to stuff something that isn't a number in `{:d}`, it triggers a fatal error.

You can combine this with field numbering and naming. Just put that label before the colon:

```
>>> "The time is {hour:d} o' clock.".format(hour=11)
"The time is 11 o' clock."
>>> "The answer is {0:f}, not {1:f}. But you can round it to {0:d}."
    .format(12, 14)
'The answer is 12.000000, not 14.000000. But you can round it to 12.
    '
```

The rule is easy: if you label a field - whether it's a string name, or a number - always put that label first within the curly brackets.

The part after after the colon is called a *format spec*. There's actually many options you can stuff in there, but let's focus on those related to formatting numbers first. As you saw, the code for an integer is `d`, converting the argument to an integer, if possible. If not, it throws a `ValueError`. (By the way, "d" stands for "decimal number", as in a base-ten number.)

Then there's floating point numbers. With the `f` code, we get six decimal places of precision by default - which are filled with zeros if necessary:

```
>>> from math import pi
>>> 'The ratio is about {:f}'.format(pi)
'The ratio is about 3.141593'
>>> '{:f} is NOT a good approximation.'.format(3)
'3.000000 is NOT a good approximation.'
```

"f" actually stands for "fixed-point number", not "floating point" - we'll see some variations later. We can change the number of fixed points: writing a period followed by a number means to use that many decimal places. We put it between the colon and the letter "f", like so:

```
>>> 'The ratio is about {:.3f}'.format(pi)
'The ratio is about 3.142'
>>> '{:.1f} is NOT a good approximation.'.format(3)
'3.0 is NOT a good approximation.'
```

It's easier for humans to read numbers with many digits if they have commas. You can tell the formatter to do this by putting a comma after the colon:

```
>>> "Billions and {:,d}'s".format(10**9)
"Billions and 1,000,000,000's"
>>> "It works with floating point too: {:,f}".format(10**9)
'It works with floating point too: 1,000,000,000.000000'
```

For large numbers, sometimes we want scientific notation, also called exponent notation. We can use the code "E" for that instead:

```
>>> "Billions and {:E}'s".format(10**9)
"Billions and 1.000000E+09's"
>>> "Precision works the same: {:.2E}".format(10**9)
'Precision works the same: 1.00E+09'
```

So far, we have seen codes for three presentation types: d (decimal) for integers; and f (fixed-point) and E (exponential) for floating-point numbers. We actually have many other choices for both: read the format-specification mini-language section[1] in the Python docs.

## 2.3 Width, Alignment, and Fill

In the examples above, the substituted values will take only as much space as they need, but no more. So `"a{}b".format(n)` will render as `a7b` if n is 7 - but not `a07b`, for example. But if n is 77, it will expand to take up four characters instead of three: `a77b`.

We can change this default behavior, placing the value in a field of a certain number of characters. If it's small enough to fit in there (i.e. not too many digits or chars), then it will be right-justified. We specify the width by putting the number of characters between the colon and the type code:

```
>>> "foo{:7d}bar".format(753)
'foo    753bar'
```

[1] https://docs.python.org/3/library/string.html#format-specification-mini-language

Let's count the character columns here:

```
foo    753bar
0123456789012
```

Positions 3 through 9 are taken up by the replacement field value. That value only has three characters (753), so the others are *filled* by the space character. We say that the space is the *fill* character here.

By default, the value is right-justified in the field for numbers. But for strings, it's left justified:

```
>>> "foo{:7s}bar".format("blah")
'fooblah   bar'
```

Generally speaking, it will default to right-justifying for any kind of number, and left-justify for everything else. We can override the default, or even just be explicit about what we want, by inserting an *alignment* right before the field width. For right-justifying, this is the greater-than sign:

```
>>> "foo{:>7d}bar".format(753)
'foo    753bar'
>>> "foo{:>7s}bar".format("blah")
'foo   blahbar'
```

To left-justify, use a less-than sign:

```
>>> "foo{:<7d}bar".format(753)
'foo753    bar'
>>> "foo{:<7s}bar".format("blah")
'fooblah   bar'
```

Or we can center it, with a caret:

```
>>> "foo{:^7d}bar".format(753)
'foo  753  bar'
>>> "foo{:^7s}bar".format("blah")
'foo blah  bar'
```

So far, the extra characters in the field have been spaces. That extra character is called the *fill character*, or the *fill*. We can specify a different fill character by placing it just before the alignment character (<, > or ^):

```
>>> "foo{:_>7d}bar".format(753)
'foo____753bar'
>>> "foo{:+<7d}bar".format(753)
'foo753++++bar'
>>> "foo{:X^7d}bar".format(753)
'fooXX753XXbar'
```

This is a good time to remember you can combine field names or indices with these format annotations - just put the name or index to the left side of the colon:

```
>>> "alpha{x:_<6d}beta{y:+^7d}gamma".format(x=42, y=17)
'alpha42____beta++17+++gamma'
>>> "{0:_>6d}{1:-^7d}{0:_<6d}".format(11, 333)
'____11--333--11____'
```

Historically, over the long and ongoing lifetime of `printf`, it's been common to use zero as a fill character for right-justified integer fields. So if the number 42 is put in a five-character-wide field, it shows up at "00042". Since people often want to do this, a shorthand evolved. You can omit the alignment character if the fill is "0" (zero), and the type is decimal:

```
>>> "foo{:0>7d}bar".format(753)
'foo0000753bar'
>>> "foo{:07d}bar".format(753)
'foo0000753bar'
```

That doesn't generally work for other fill values, though:

```
>>> "foo{:_7d}bar".format(753)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Invalid format specifier
```

What happens if the value is too big to fit in a tiny field? The width is actually a *minimum* width. So it will expand as needed to fill it out:

```
>>> "red{:4d}green".format(123456789)
'red123456789green'
```

There isn't any way to specify a maximum width. If you need that, you can convert it to a string, implement your own trimming logic, then inject that trimmed string:

```
>>> value = 123456789 # Or some other number.
>>> trimmed_value = str(value)[:4] # Or last 4, etc.
>>> "red{:>4s}green".format(trimmed_value)
'red1234green'
```

## 2.4 F-Strings

Python 3.6 introduced an alternative to `str.format()`, called **f-strings**. The formal name is "formatted string literal". Instead of a `.format()` method, you prefix the string with the letters "f" or "F", putting variable names directly inside the replacement field:

```
>>> time_of_day = "afternoon"
>>> f"Good {time_of_day}, my friend"
'Good afternoon, my friend'
>>> F"Good {time_of_day}, my friend"
'Good afternoon, my friend'
```

It's exactly equivalent to this:

```
>>> # This...
... "Good {time_of_day}, my friend".format(
...     time_of_day=time_of_day)
'Good afternoon, my friend'
>>> # Or this:
... "Good {}, my friend".format(time_of_day)
'Good afternoon, my friend'
```

If your values are already stored in locally-readable variables, using f-strings is more succinct. But you can do more than that. In fact, the replacement field (i.e. the curly braces) can contain not only a variable name, but a full Python expression!

```
>>> f"Good {time_of_day.upper()}, my friend"
'Good AFTERNOON, my friend'

>>> def reverse(string):
...     return string[::-1]
>>> f"Good {reverse(time_of_day)}, my friend"
'Good noonretfa, my friend'

>>> groceries = ["milk", "bread", "broccoli"]
>>> f"I need to get some {groceries[2]}."
'I need to get some broccoli.'
```

You can do this with `str.format()`, too, but f-strings express it a bit more naturally. Aside from that, you can use the normal number formatting codes. After the expression name, simply write a colon, and the same code you would use for `str.format()`:

```
>>> from math import pi
>>> f"The ratio is about {pi:f}"
'The ratio is about 3.141593'
>>> f"Which is roughly {pi:0.2f}"
'Which is roughly 3.14'
>>> f"{pi:.0f} is NOT a good approximation."
'3 is NOT a good approximation.'

>>> number = 10**9
>>> f"Billions and {number:,d}'s"
"Billions and 1,000,000,000's"
>>> f"Billions and {number:E}'s"
"Billions and 1.000000E+09's"
```

As well as the width, alignment, and fill:

```
>>> num = 753
>>> word = "WOW"
>>> f"foo{num:7d}bar"
'foo    753bar'
>>> f"foo{word:7s}bar"
'fooWOW    bar'
>>> f"foo{word:>7s}bar"
'foo    WOWbar'
>>> f"foo{num:<7d}bar"
'foo753    bar'
>>> f"foo{num:^7d}bar"
'foo  753  bar'
>>> f"foo{num:X^7d}bar"
'fooXX753XXbar'
```

Now it's clear why I emphasize `str.format()` in this chapter, and this book. Practically speaking, all Python programmers need to know `str.format()` anyway; and once you've learned it, you are fluent with f-strings almost immediately.

The main downside to f-strings will become less important over time. It only works with Python 3.6 and later. That means in order to use f-strings in your code, you must be working on a codebase which will only ever be executed on those versions. This applies not only to your fellow developers, but - more problematically - all end users. If a customer has installed Python 3.5 on a server, and your program uses f-strings, they won't be able to run it unless you can convince them to upgrade. Which, unfortunately, probably isn't as high a priority for them as it is for you.

As I write this, f-strings have been out a short while. But many Python developers seem to have already fallen in love with them. The next edition of this book may emphasize them more, depending how quickly the Python community moves to versions of Python which support f-strings, and how popular they become. Fortunately, it's quite easy to switch back and forth between `str.format()` and f-strings; there seems to be very little mental energy needed to switch. So if you want to use f-strings, you can do so when it's practical, and then easily switch to `str.format()` when needed.

## 2.5 Percent Formatting

Modern Python still needs percent formatting in a few places, mainly when you work with the `logging` module. Thankfully, you don't need to know all its details. Learning just a few parts of percent formatting will cover 95% of what you're likely to need. I'll focus on that high-impact portion here; for more detail, consult the official Python reference.[2]

Percent formatting is officially called "printf-style string formatting", and - like the string formatting in *many* languages - has its roots in C's `printf()`. So if you have experience with that, you'll skim through quickly - though there *are* a few differences.

It uses the percent character in two ways. Here's a simple example:

```
>>> "Hello, %s, today is %s." % ("Aaron", "Tuesday")
'Hello, Aaron, today is Tuesday.'
```

Notice percent is used as a binary operator. On its left is a string; on its right, a tuple of two strings. That string on the left is called the *format*; you can see it has two percent characters inside. Specifically, it has the sequence `%s` twice. These `%s` sequences are called *conversion specifiers*. There's two of them, and two values in the tuple on the right; they map to each other. Each value is substituted for its corresponding `%s`.

The "s" means that the inserted value is converted to a string; these are already strings, though, so they are just placed in. You can also use %d for an integer:

```
>>> "Here's %d dollars and %d cents." % (14, 25)
"Here's 14 dollars and 25 cents."
```

Sometimes it doesn't matter much which specifier you use. If that last format string was `"Here's %s dollars and %s cents."`, it would render the same. But I recommend you choose the strictest type that will work for your data; if you expect the value to be an integer, use %d, so that if it's *not* an integer, you'll get a clear stack trace telling you what's wrong:

```
>>> "Here's %d dollars and %d cents." % (14, "a quarter")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
```

(I'm assuming you'd rather discover a lurking bug now, during development, instead of through an angry customer's bug report later.)

[2] https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

Sometimes you'll have just one value to interject. A tuple of one value must be written with an extra comma, like `(foo,)` - because `(foo)` becomes simply `foo` in the normal meaning of parentheses for grouping. So to format a string of one variable, you can type this:

```
>>> "High %d!" % (5,)
'High 5!'
```

Early in Python's history, people decided typing those extra characters was a bit annoying, especially given how common it is to format a string with a single value. So as a special case, when you have a single specifier, you can pass in a single value instead of a tuple:

```
>>> "High %d!" % 5
'High 5!'
```

In addition to `%d` and `%s`, you can use `%f` for a floating-point number:

```
>>> "I owe you $%f." % 7.05
'I owe you $7.050000.'
```

You'll often want to specify its precision - the number of digits to the right of the decimal. Do this by inserting a *precision code* between `%` and `f` - which will be a `.` (dot) followed by an integer:

```
>>> "I owe you $%.2f." % 5.05
'I owe you $5.05.'
```

You can also use `%r` (which formats the `repr()` of the object). It's especially useful for logging and troubleshooting:

```
>>> class Money:
...     def __init__(self, dollars, cents):
...         self.dollars = dollars
...         self.cents = cents
...     def __repr__(self):
...         return 'Money({},{})'.format(
...             self.dollars, self.cents)
>>> cash = Money(127, 82)
>>> "Cash on hand: %r" % cash
'Cash on hand: Money(127,82)'
```

There is much more to percent formatting than this, but what we've covered lets you read and write most of what you need.

Now, as mentioned, in modern Python you'll mainly need to use percent formatting with the `logging` module. However, in that case, you use it a bit differently. As described in its chapter, the `logging` module includes functions for log events at different levels of urgency - whether that's an error, a warning, or even a non-urgent informational message:

```
logging.info("So far, so good!")
```

You will *very* often want to inject run-time values into the message. For example, if a customer spends a certain amount of money:

```
logging.info("User %s spent $%0.2f", username, amount)
```

Notice there's no binary percent operator! That's deliberate. The logging functions are designed to take a format string as the first argument, and the values to substitute as subsequent arguments. That's because not every log message needs to be executed. You can - and often will - configured your logger to filter out all those boring `info` messages, for example, or omit the overly detailed debug messages. In other words, don't do this:

```
# NO! Bad code!
logging.info("User %s spent $%0.2f" % (username, amount))
```

Suppose you are filtering out `info` messages right now, so the `info` message doesn't need to actually log. In the first, recommended form, that `logging.info` line in your code is cheap; it's essentially treated by Python as a no-op. In the second form, it will still be translated as a no-op, *but only after that string is rendered.* So you unnecessarily incur the cost of rendering the string, just to throw it away. This is all explained in more detail in the logging chapter; for now, just be aware of the idea.

# Index