# prelogging Documentation

***Release 0.4.3***

**Brian O'Neill**

**Sep 17, 2018**

# Contents

*prelogging* is a pure Python package that provides a simple, consistent, powerful API for configuring logging. The package includes several "batteries":

- consistent methods to add handlers both mundane and exotic
- multiprocessing-safe handlers which output to the console, to files and rotating files, and to the system log
- formatter presets — a fund of useful shorthands for formatters, which you can extend and modify

CHAPTER 1

# Chapters

## 1.1 Introduction and Setup

*prelogging* is a package for setting up, or *configuring*, the logging facility of the Python standard library. To *configure logging* is to specify the logging entities you wish to create — formatters, handlers, optional filters, and loggers — as well as which of them use which others.

Once configured, logging messages with the *logging* facility is simple and powerful; configuration presents the only challenge. *logging* provides a couple of approaches to configuration — static, using a dict or an analogous YAML text file; and dynamic, using the *logging* API — both of which have their shortcomings.

*prelogging* offers a hybrid approach: a streamlined, consistent API for incrementally constructing a dict used to configure logging statically. As you build the configuration dict, by default *prelogging* checks for possible mistakes and issues warnings on encountering them. *prelogging* also supplies missing functionality: it provides multiprocessing-safe logging to the console, to files and rotating files, and to *syslog*.

### 1.1.1 Requirements

The *prelogging* package requires only Python 3.4+ or 2.7. It has no external dependencies.

Very little of *prelogging*'s code is sensitive to Python 3 vs 2. To achieve backwards compatibility with 2.7 we sacrificed, with some reluctance, type annotations and keyword-only parameters. To address the few remaining differences, we've used *six* sparingly (one decorator, one function, and one constant). The *prelogging* package includes a copy of the `six.py` module (v1.10.0, for what it's worth), so no separate installation is required.

The *prelogging* distribution contains an `examples/` subdirectory. A few examples (`mproc_deco*.py`) use the deco package, which provides a "simplified parallel computing model for Python". However, the examples are just for illustration (and code coverage), and aren't installed with the *prelogging* package.

The distribution also contains subdirectories `tests/` and `docs/`, which similarly are not installed.

### 1.1.2 Installation

You can install *prelogging* from PyPI (the Python Package Index) using `pip`:

```
$ pip install prelogging
```

(Here and elsewhere, `$` at the beginning of a line indicates your command prompt, whatever that may be.)

Alternately, you can

- clone the github repo, or
- download a `.zip` or `.tar.gz` archive of the repository from github or PyPI, uncompress it to a fresh directory, change to that directory, and run the command:

  ```
  $ python setup.py install
  ```

On *nix systems, including macOS, `setup.py` is executable and has a proper shebang, so on those platforms you can just say:

```
$ ./setup.py install
```

Downloading and uncompressing the archive lets you review, run and/or copy the tests and examples, which aren't installed by `pip` or `setup.py`. Whichever method you choose to install *prelogging*, ideally you'll do it in a virtual environment.

### 1.1.3 Running tests and examples

The top-level directory of the *prelogging* distribution (where `setup.py` resides) has subdirectories `tests/` and `examples/`, which contain just what their names suggest.

In the top-level directory are three scripts — `run_tests.py`, `run_examples.py`, and `run_all.py`, all executable on *nix platforms — which respectively run all tests, all examples, or both.

#### Running tests

You can run all the tests before installing *prelogging* by running the script `run_tests.py` in the top level directory of the repository:

```
$ python run_tests.py
```

Alternately, you can run

```
$ python setup.py test
```

#### Coverage from tests

*prelogging* contains a small amount of Python-2-only code (workarounds for Py2 shortcomings), and supports a few Python-3-only logging features. In addition, several methods in `lcdict.py` add various exotic handlers, which are easy to write examples for but difficult to test (coverage for this module increases to 98%/96% when examples are included — see the following section).

| Module | Py 3 | Py 2 |
|---|---|---|
| lcdictbasic.py | 99% | 99% |
| lcdict.py | 88% | 88% |
| locking_handlers.py | 89% | 89% |
| lcdict_builder_abc.py | 100% | 100% |

### Running examples

Examples are not installed; they're in the `examples/` subdirectory of the distribution. You can run all the examples by running the script `run_examples.py` in the top-level directory:

```
$ python run_examples.py
```

From the same directory, you can run all tests and examples with the script `run_all.py`:

```
$ python run_all.py
```

Note: the examples that use `deco` of course require that package to be installed; the SMTP examples require that you edit `examples/_smtp_credentials.py` to contain valid email credentials.

The section *Guide to Examples* catalogs all the examples and briefly describes each one.

### Coverage from tests + examples

A few short passages, mostly Python-major-version-specific code, keep *prelogging* shy of 100% coverage when both tests and examples are run:

| Module | Py 3 | Py 2 |
|---|---|---|
| lcdictbasic.py | 99% | 100% |
| lcdict.py | 98% | 96% |
| locking_handlers.py | 100% | 100% |
| lcdict_builder_abc.py | 100% | 100% |
| formatter_presets.py | 100% | 100% |

## 1.2 Overview of Logging

Logging is an important part of a program's internal operations, an essential tool for development, debugging, troubleshooting, performance-tuning and general maintenance. A program *logs messages* in order to record its successive states, and to report any anomalies, unexpected situations or errors, together with enough context to aid diagnosis.

Messages can be logged to multiple destinations at once — `stderr` in a terminal, a local file, the system log, email, or a Unix log server over TCP, to cite common choices.

At the end of this chapter we provide several *logging documentation links*, for reference and general culture. It's not our purpose to rehash or repeat the extensive (and generally quite good) documentation for Python's *logging* package; in fact, we presuppose that you're familiar with basic concepts and standard use cases. Nevertheless, it will be helpful to review several topics, and in the process clarify some obscure features of *logging*.

### 1.2.1 Using *logging*

A program logs messages using the `log` method of objects called *loggers*, which are implemented in *logging* by the `Logger` class. You can think of the `log` method as a pumped-up `print` statement. It writes a message, tagged with a level of severity, to zero or more destinations. In *logging*, a *handler* — a `Handler` object — represents a single destination, together with a specified output format. A handler implements abstract methods which format message data into structured text and write or transmit that text to the output. A logger contains zero or more handlers. When a program logs a message by calling a logger's `log` method (or a shorthand method such as `debug` or `warning`), the logger dispatches the message data to its handlers.

All messages have a *logging level*, or *loglevel*, indicating their severity or importance. The predefined levels in `logging` are `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`, listed in order of increasing severity. Both loggers and handlers have an associated *loglevel*, indicating a severity threshold: a logger or a handler will filter out any message whose loglevel is less than its own. In order for a message to actually be sent to a particular destination, its loglevel must equal or exceed the loglevels of both the logger and the handler representing the destination.

**Note**: This last statement is *basically* true, but glosses over details. We'll sharpen it below, in the subsection *How a message is logged*.

**Sensible choices for dedicated loggers**

The logger named `'__name__'` is the standard choice for a module's dedicated logger; the logger named `'__package__'` is a great choice for a package. Without any configuration, these will just write message text to `stderr`.

This elegant system allows developers to easily dial in different amounts of logging verbosity. When developing a module or package, you can use a dedicated logger to log internal messages at thoughtfully chosen loglevels. In development, set the logger's loglevel to `DEBUG` or `INFO` as needed; once the module/package is in good condition, raise that to `WARNING`; in production, use `ERROR`. There's no need to delete or comment out the lines of code that log messages, or to precede each such block with a conditional guard. The logging facility is a very sophisticated version of using the `print` statement for debugging.

### 1.2.2 *logging* classes that can be configured

*logging* defines a few types of entities, culminating in the `Logger` class. Typically, a program or library will set up, or *configure*, logging only once, at startup. This entails specifying message formats, destinations, loggers, and containment relations between those things. Once a program has configured logging as desired, use of loggers is very straightforward. Configuration, then, is the only barrier to entry.

The following diagram displays the types that can be configured statically, and their dependencies:

In words:

- a `Logger` can have one or more `Handlers`, and a `Handler` can be used by multiple `Loggers`;
- a `Handler` has at most one `Formatter`, but a `Formatter` can be shared by multiple `Handlers`;

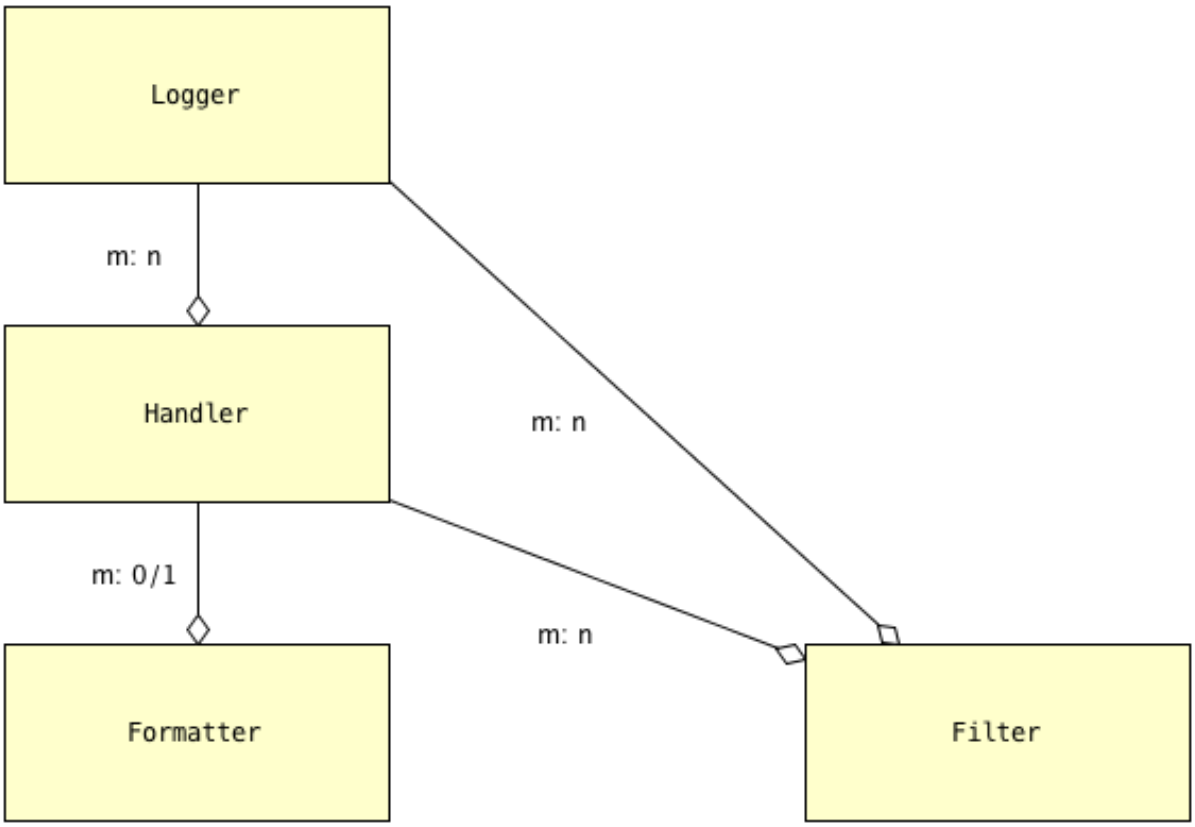



Fig. 1: The objects of *logging* configuration

| Symbol | Meaning |
|---|---|
|  | has zero or more |
| m: 0/1 | many-to-(zero-or-one) |
| m: n | many-to-many |

- Handlers and Loggers can each have zero or more Filters; a Filter can be used by multiple Handlers and/or Loggers.

#### What these objects do

A Formatter is basically just a format string that uses keywords defined by the *logging* module — for example, '%(message)s' and '%(name)-20s: %(levelname)-8s: %(message)s'.

A Handler formats and writes logged messages to a particular destination — a stream (e.g. sys.stderr, sys.stdout, or an in-memory stream such as an io.StringIO), a file, a rotating set of files, a socket, etc.

A Logger sends logged messages to its associated handlers. Various criteria filter out which messages are actually written, notably loglevel thresholding as described above and in greater detail *below*.

Filters provide still more fine-grained control over which messages are written. They can also be used to modify messages or supplement them with additional context.

### 1.2.3 Loggers are identified by name

A logger is uniquely identified by name (except for the name 'root': see the Warning below). For example, the expression logging.getLogger('mylogger') always denotes the same object, no matter where in a program it occurs or when it's evaluated. The *logging* package always creates a special logger, the *root logger*, which *we*, as users of *logging*, identify by the name '' (the empty string); it's accessed by the expression logging.getLogger(''), or equivalently by logging.getLogger().

**Warning:**

**The root logger's name is set to, and reported as, 'root':**

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

**Confusingly, however, you cannot access the root logger by that name:**

```
>>> logging.getLogger('') is logging.getLogger('root')
False
```

**It's most unfortunate that these two *distinct* loggers share the same name:**

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

**Do not use the logger name 'root'.**

Logger names are *dotted names*, and behave in a way that's analogous to package and module names. The analogy is intentional, to facilitate a style of logging in which each package, and/or each module, uses its own logger, with names __package__ and __name__ respectively. The basic idioms are, for example:

```
logging.getLogger(__name__).debug("About to do that thing")
```

and:

```
logging.getLogger(__package__).warning("dict of defaults is empty")
```

Broadly speaking, a logger corresponds to an "area" of your program; you're free to construe that in whatever way suits your needs and situation.

#### The parent-child and ancestor relationships between loggers

A parent-child relation obtains among loggers: the parent of a logger `'a.b.c'` is the logger `'a.b'`, whose parent is `'a'`; the parent of logger `'a'` is the root logger, identified by `''`. The logger `'a'` is an *ancestor* of both `'a.b'` and `'a.b.c'`; `'a.b'` is an ancestor of `'a.b.c'`; the root logger is an ancestor of every other logger. (Note, though, that `aa` is *not* a parent or ancestor of `a`, nor is `a.b` a parent or ancestor of `a.bxyz`: the relation isn't just "startswith" between strings.)

### 1.2.4 How a message is logged

In order to explain what happens when a logger logs a message,

```
logging.getLogger('L').log(level, message)
```

we first have to introduce a few more concepts:

- the 'NOTSET' loglevel
- the "effective level" of a logger
- the `propagate` flag of a logger.

#### The special loglevel NOTSET

There's actually a sixth predefined loglevel, `NOTSET`, whose numeric value is 0, lower than the "real" loglevels (`DEBUG` = 10, ..., `CRITICAL` = 50), which are all non-zero. The root logger by default has loglevel `WARNING`, but all created loggers and handlers have default loglevel `NOTSET`.

`NOTSET` is useless as a loglevel of individual messages. You can't successfully log a message at level `NOTSET` — nothing happens (unless you do something unusual. If you call `logging.disable(neg)` with some negative integer `neg`, you can get `logger.log(0, message)` to emit `message`; but ordinarily, you wouldn't, and it won't.)

A handler with loglevel `NOTSET` rejects no messages; it's the most inclusive level.

When a logger has loglevel `NOTSET`, the loglevels of its ancestors are examined to compute its *effective level* — the level that *logging* uses to determine whether a message that the logger logs will be sent to handlers or not.

#### The "effective level" of a logger

The *effective level* of a logger is its own level if that is non-zero; otherwise, it's the level of its nearest ancestor whose level is non-zero; otherwise, if there's no such ancestor, it's `NOTSET` (0). The `Logger` method `getEffectiveLevel()` returns this value. Its docstring explains that it "[loops] through [the] logger and its parents in the logger hierarchy, looking for a non-zero logging level[, returning] the first one found." (`getEffectiveLevel()` is in the `__init__.py` module of *logging*.)

Now we can make good on an earlier promise – the following statement isn't just "basically true" but really is the case:

> In order for a message to actually be written to a particular destination,
> its level must equal or exceed the *effective level* of the logger that
> logged it, as well as the level of the handler representing the destination.

In the next subsection we'll explain just which handlers a message is sent to when its level clears the effective level threshold.

#### Propagation

Loggers have a `propagate` attribute, a flag, which by default is True.

`propagate` determines which handlers a `message` is sent to when a `logger` logs it at a particular `level` via a call such as `logger.log(level, message)`.

If `logger` has handlers, the `message` is sent to them. If `logger` isn't the root and `logger.propagate` is True, the message is *also* sent to any handlers of the logger's parent; if the parent isn't the root and its `propagate` flag is True, the message is sent to the handlers of the parent's parent; and so on, until this process reaches either the root or an ancestor whose `propagate` flag is False. The loglevels of ancestor loggers are **not** consulted when they are ascended through; the message is sent directly to their handlers.

If no handlers are encountered in this procedure, in Python 3.2+ the message is sent to the "handler of last resort", `logging.LastResort`, whose loglevel is 'WARNING', and which simply writes the message to `stderr`. (In earlier versions of Python, or if you set `logging.LastResort = None` in 3.2+, an error message is written to `stderr` that no handlers could be found for the logger.)

In many cases, to configure logging it's sufficient just to add a handler or few and attach them to the root.

---

**Note:** The *logging* documentation contains a pair of flowcharts, "Logging flow" and "Handler flow", which summarize what this section, *How a message is logged*, has described; however, they seem to predate Python 3.2, so "Handler flow" doesn't mention the "last resort" handler.

---

### 1.2.5 *logging* defaults

*logging* supplies reasonable out-of-the-box defaults and shorthands so that you can easily start to use its capabilities.

When accessed for the first time, the `Logger` named `'mylogger'` is created "just in time" if it hasn't been explicitly configured. You don't *have* to attach handlers to `'mylogger'`; logging a message with that logger will "just work". Suppose this is a complete program:

```
import logging
logging.getLogger('mylogger').warning("Uh oh.")
```

When run, it writes `Uh oh.` to `stderr`. In light of the last section, we can now understand why. The effective level of `'mylogger'` is the level of its parent, the root logger, which is `WARNING`, and the level of the message clears that threshold. Thus, the message is sent to `'mylogger'`'s handlers (none). Because `'mylogger'` has `propagate` set to True, the message is also sent to the handlers of the root. The root has no handlers, so the message is sent to the last resort handler, whose loglevel is `WARNING`, which lets the message through, writing it to `stderr`.

The `warning(...)` logger method shown above is a shorthand for `log(logging.WARNING, ...)`. Similarly, there are convenience methods `debug`, `info`, `error` and `critical`.

#### The *logging* convenience functions `log()`, `debug()`,... `critical()` have a side-effect

*logging* provides six functions, `logging.log()`, `logging.debug()`,... `logging.critical()`, which let you instantly use logging out of the box, with no configuration or even any calls to `getLogger`. You can just call:

```
logging.error("Something went wrong")
```

and something plausible will happen. This works because `logging.error(...)` is (almost always) a shorthand for `logging.getLogger().error(...)`.

However, in one circumstance these six functions all have a side effect which can make them **not** mere shorthands for expressions that explicitly access the root logger with `getLogger`.

Specifically, if the root logger has no handlers when any of them is called, these functions call `logging.basicConfig()` (with no arguments), which creates a `stderr` stream handler that has a formatter with format string

```
BASIC_FORMAT = "%(levelname)s:%(name)s:%(message)s"
```

and attaches it to the root. One might expect that these functions under such circumstances would use the LastResort handler, as described above; but they don't.

Consider this complete program:

```
import logging
from importlib import reload
import sys

# logging.error installs a root handler because none exist,
# so order of these three calls matters.
logging.getLogger('').error('Trouble!')
logging.getLogger('newlogger').critical('Big trouble!')
logging.error('Trouble!')

print('-----------------', file=sys.stderr)
reload(logging)

# Clear everything, and do the three calls again, with logging.error first.
logging.error('Trouble!')
logging.getLogger('newlogger').critical('Big trouble!')
logging.getLogger('').error('Trouble!')
```

When run, it prints these messages to `stderr`:

```
Trouble!
Big trouble!
ERROR:root:Trouble!
-----------------
ERROR:root:Trouble!
CRITICAL:newlogger:Big trouble!
ERROR:root:Trouble!
```

The call to `logging.error` attaches a new handler to the root. Subsequently, all loggers that propagate to the root have the format of their messages changed (albeit for the better).

**logging.basicConfig()**

The logging.basicConfig() function lets you configure the root logger (up to a point), using a monolithic function that's somewhat complex yet of limited capabilities. When used to quickly configure logging with a single call, the function can create a stream handler, or a file handler (but not both!), and attaches it to the root.

---

In the next chapter, we'll examine the approaches to configuration offered by *logging*, and then see how *prelogging* simplifies the process.

### 1.2.6 *logging* documentation links

See the logging docs for the official explanation of how Python logging works.

For the definitive account of static configuration, see the documentation of logging.config, in particular the documentation for the format of a logging configuration dictionary.

Here's a useful reference: the complete list of keywords that can appear in formatters.

The logging HOWTO contains tutorials that show typical setups and uses of logging, configured in code at runtime. The logging Cookbook contains a wealth of techniques, several of which exceed the scope of *prelogging* because they involve *logging* capabilities that can't be configured statically (e.g. the use of LoggerAdapters, or QueueListeners ). A few of the examples contained in the *prelogging* distribution are examples from the Cookbook and HOWTO, reworked to use *prelogging*.

The *logging* package supports multithreaded operation, but does **not** directly support logging to a single file from multiple processes. Happily, *prelogging* does, in a couple of ways, both illustrated by examples.

One additional resource merits mention: the documentation for logging in Django provides another, excellent overview of logging and configuration, with examples. Its first few sections aren't at all Django-specific.

## 1.3 Configuration — with *logging*, and with *prelogging*

We'll use a simple example to discuss and compare various approaches to logging configuration — using the facilities provided by the *logging* package, and then using *prelogging*.

### 1.3.1 Logging configuration requirements — example

Suppose we want the following configuration:

> **Configuration requirements**
>
> Messages should be logged to both `stderr` and a file. Only messages with loglevel `INFO` or higher should appear on-screen, but all messages should be logged to the file. Messages to `stderr` should consist of just the message, but messages written to the file should also contain the logger name and the message's loglevel.
>
> The logfile contents should persist: the file handler should **append** to the logfile, rather than overwriting it each time the program using these loggers is run.

This suggests two handlers, each with an appropriate formatter — a `stderr` stream handler with level `INFO`, and a file handler with level `DEBUG` or `NOTSET`. (`NOTSET` is the default loglevel for handlers. Numerically less than `DEBUG`, all loglevels are greater than or equal to it.) Both handlers should be attached to the root logger, which should have level `DEBUG` to allow all messages through. The file handler should be created with `mode='a'` (for append, not `'w'` for overwrite) so that existing logfile contents persist.

#### Using the example configuration

Once this configuration is established, these logging calls:

```
import logging
root_logger = logging.getLogger()
# ...
root_logger.debug("1. 0 = 0")
root_logger.info("2. Couldn't create new Foo object")
```

(continues on next page)

(continued from previous page)

```
root_logger.debug("3. 0 != 1")
root_logger.warning("4. Foo factory raised IndexError")
```

should produce the following `stderr` output:

```
2. Couldn't create new Foo object
4. Foo factory raised IndexError
```

and the logfile should contain (something much like) these lines:

```
root                : DEBUG   : 1. 0 = 0
root                : INFO    : 2. Couldn't create new Foo object
root                : DEBUG   : 3. 0 != 1
root                : WARNING : 4. Foo factory raised IndexError
```

### 1.3.2 Meeting the configuration requirements with *logging*

The *logging* package offers two approaches to configuration:

- dynamic, using code;
- static (and then, there are two variations).

These can be thought of as *imperative* and *declarative*, respectively. The following subsections show how each of these approaches can be used to meet the requirements stated above.

#### Using dynamic configuration

Here's how to dynamically configure logging to satisfy the given requirements:

```
import logging
import sys

root = logging.getLogger()
root.setLevel(logging.DEBUG)

# Create stderr handler,
#   level = INFO, formatter = default i.e. '%(message)s';
# attach it to root
h_stderr = logging.StreamHandler(stream=sys.stderr)
h_stderr.setLevel(logging.INFO)
root.addHandler(h_stderr)

# Create file handler, level = NOTSET (default),
#   filename='blather_dyn_cfg.log', formatter = logger:level:msg, mode = 'a'
# attach it to root
logger_level_msg_fmtr = logging.Formatter('%(name)-20s: %(levelname)-8s: %(message)s')
h_file = logging.FileHandler(filename='blather_dyn_cfg.log')
h_file.setFormatter(logger_level_msg_fmtr)
root.addHandler(h_file)
```

We've used a number of defaults. It was unnecessary to add:

```
msg_fmtr = logging.Formatter('%(message)s')
h_stderr.setFormatter(msg_fmtr)
```

because the same effect is achieved without them. The default `mode` of a `FileHandler` is `a`, which opens the logfile for appending, as per our requirements; thus it wasn't necessary to pass `mode='a'` to the `FileHandler` constructor. (We omitted other arguments to this constructor, e.g. `delay`, whose default values are suitable.) Similarly, it wasn't necessary to set the level of the file handler, as the default level `NOTSET` is just what we want.

### Advantages of dynamic configuration

- *Hierarchy of logging entities respected*

  Formatters must be created before the handlers that use them; handlers must be created before the loggers to which they're attached.

  You can configure the entities of logging (formatters, optional filters, handlers, loggers) one by one, in order, starting with those that don't depend on other entities, and proceeding to those that use entities already defined.

- *Methods of the 'logging' API provide reasonable defaults*

  With static configuration, certain fussy defaults must be specified explicitly.

- *Error prevention*

  For instance, there's no way to attach things that simply don't exist.

- *Fine-grained error detection*

  If you use a nonexistent keyword argument, for example, the line in which it occurs gives an error; you don't have to wait until issuing a final `dictConfig` call to learn that something was amiss.

  Thus it's easier to debug: each step taken is rather small, and you can fail faster than when configuring from an entire dictionary.

### Disadvantages of dynamic configuration

- *Low-level methods, inconsistent API*

  The `Handler` base class takes a keyword argument `level`; however, its subclass `StreamHandler` takes a keyword argument `stream`, but doesn't recognize `level`. Thus we couldn't concisely say:

```
h_stderr = logging.StreamHandler(level=logging.INFO, stream=sys.stderr)
```

  but had to call `h_stderr.setLevel` after constructing the handler.

- *In 'logging', only loggers have names; formatters, handlers and filters don't*

  Thus we have to use Python variables to reference the various logging entities which we create and connect. If another part of the program later wanted to access, say, the file handler attached to the root logger, the only way it could do so would be by iterating through the `handlers` collection of the root and examining the type of each:

```
root = logging.getLogger()
fh = next(h for h in root.handlers if isinstance(h, logging.FileHandler))
```

- *Somehow it winds up more even verbose than static dictionaries*

  The methods are low-level, and many boilerplate passages recur in dynamic configuration code.

### Using static configuration

The *logging.config* submodule offers two equivalent ways to specify configuration statically:

- with a dictionary meeting various requirements (mandatory and optional keys, and their values), which is passed to `logging.config.dictConfig()`;
- with a text file written in YAML, meeting analogous requirements, and passed to `logging.config.fileConfig()`.

We'll call a dictionary that can be passed to `dictConfig` a *logging config dict*. The schema for configuration dictionaries documents the format of such dictionaries. (Amusingly, it uses YAML to do so!, to cut down on the clutter of quotation marks. colons and curly braces.)

We'll deal only with logging config dicts, ignoring the YAML-based approach. The Web frameworks Django and Flask configure logging with dictionaries. (Django can accomodate YAML-based configuration, but its path of least resistance is certainly the dict-based approach.) Dictionaries are native Python; YAML isn't. YAML may be more readable than dictionary specifications, but *prelogging* offers another, pure-Python solution to that problem.

### Configuring our requirements statically

Here's how to do so:

```
import logging
from logging import config

config_dict = {
     'formatters': {'logger_level_msg': {'format': '%(name)-20s: %(levelname)-8s: '
                                                   '%(message)s'}},
     'handlers': {'h_stderr': {'class': 'logging.StreamHandler',
                               'level': 'INFO',
                               'stream': 'ext://sys.stderr'},
                  'h_file': {'class': 'logging.FileHandler',
                             'filename': 'blather_stat_cfg.log',
                             'formatter': 'logger_level_msg'}},
     'root': {'handlers': ['h_stderr', 'h_file'], 'level': 'DEBUG'},
     'version': 1
}
logging.config.dictConfig(config_dict)
```

As with dynamic configuration, most keys have default values, and in the interest of brevity we've omitted those that already suit our needs. We didn't specify a formatter for the stream handler, nor the file handler's mode or loglevel, and so on.

### Advantages of static configuration

- *Logging entities are referenced by name*

  You give a name to every logging entity you specify, and then refer to it by that name when attaching it to higher-level entities. (It's true that after the call to `dictConfig`, only the names of loggers endure [*as per the documentation! but see :ref:'Note <HANDLER_NAMES_TOO>' below*]; however, that's a separate issue — a deficiency of *logging*, not of static configuration.)

- *It's arguably more natural to specify configuration in a declarative way*, especially for the typical application which will "set it and forget it".

#### Disadvantages of static configuration

- *Not very good error detection* (none until the `dictConfig` call)
- *Some boilerplate key/value pairs*
- *Lots of noise* — a thicket of nested curly braces, quotes, colons, etc.

  Triply-nested dicts are hard to read.
- *Logging config dicts seem complex*

  At least on first exposure to static configuration, it's not easy to comprehend a medium- to large-sized dict of dicts, in which many values are lists of keys occurring elsewhere in the structure.

#### Assessment

As we've seen, both approaches to configuration offered by the *logging* package have virtues, but both have shortcomings:

- Its API, mostly dedicated to dynamic configuration, is at once complex and limited.
- With static configuration, no warnings are issued and no error checking occurs until `dictConfig` (or `fileConfig`) is called.
- Of the various kinds of entities that *logging* constructs, only loggers have (documented) names, which, as seen above, can lead to various conundrums and contortions.

  Said another way, once logging is configured, only the names of `Loggers` endure. *logging* retains *no associations* between the names you used to specify `Formatters`, `Handlers` and `Filters`, and the objects constructed to your specifications; you can't access those objects by any name.

To this list, we might add the general observation that the entire library is written in thoroughgoing camelCase (except for inconsistencies, such as `levelname` in format strings).

---

### 1.3.3 Configuration with *prelogging*

*prelogging* provides a hybrid approach to configuration that offers the best of both the static and dynamic worlds. The package provides a simple but powerful API for building a logging config dict incrementally, and makes it easy to use advanced features such as rotating log files and email handlers. As you add and attach items, by default *prelogging* issues warnings when it encounters possible mistakes such as referencing nonexistent entities or redefining entities.

*prelogging* defines two classes which represent logging config dicts: a `dict` subclass `LCDictBasic`, and *its* subclass `LCDict`. (The *diagram of classes* shows all the classes in the *prelogging* package and their interrelations.) `LCDictBasic` provides the basic model of building a logging config dict; `LCDict` supplies additional conveniences — for example, formatter presets (predefined formatters), and easy access to advanced features such as filter creation and multiprocessing-safe rotating file handlers. The centerpiece of *prelogging* is the `LCDict` class.

You use the methods of these classes to add specifications of named `Formatters`, `Handlers`, `Loggers`, and optional `Filters`, together with containment relations between them. Once you've done so, calling the `config()` method of an `LCDictBasic` configures logging by passing itself, as a `dict`, to `logging.config.dictConfig()`. This call creates all the objects and linkages specified by the underlying dictionary.

Let's see this in action, applied to our use case, and then further discuss how the *prelogging* classes operate.

### Configuring our requirements using `LCDict`

Here's how we might use `LCDict` to configure logging to satisfy our *Configuration requirements*:

```
from prelogging import LCDict

lcd = LCDict(root_level='DEBUG',
             attach_handlers_to_root=True)
lcd.add_stderr_handler(
                'h_stderr',
                formatter='msg',    # actually not needed
                level='INFO'
).add_file_handler('h_file',
                   formatter='logger_level_msg',
                   filename='blather.log',
)
lcd.config()
```

First we create an `LCDict`, which we call `lcd` — a logging config dict with root loglevel `'DEBUG'`. An `LCDict` has a few attributes that aren't part of the underlying dict, including the `attach_handlers_to_root` flag, which we set to `True`. The `add_*_handler` methods do just what you'd expect: each adds a subdictionary to `lcd['handlers']` with the respective keys `'h_stderr'` and `'h_file'`, and with key/value pairs given by the keyword parameters.

We've used a couple of *prelogging*'s formatter presets — `'msg'` and `'logger_level_msg'`. Because we pass the flag `attach_handlers_to_root=True` when creating `lcd`, every handler we add to `lcd` is (by default) automatically attached to the root logger. (You can override this default by passing `add_to_root=False` to any `add_*_handler` call.)

**Notes**

- To allow chaining, as in the above example, the methods of `LCDictBasic` and `LCDict` generally return `self`.
- Here's the *complete table of prelogging's formatter presets*.

### Configuring our requirements using `LCDictBasic`

It's instructive to see how to achieve *the example configuration* using only `LCDictBasic`, foregoing the conveniences of `LCDict`. The code becomes just a little less terse. Now we have to add two formatters, and we must explicitly attach the two handlers to the root logger. We've commented those passages with `# NEW`:

```
from prelogging import LCDictBasic

lcd = LCDictBasic(root_level='DEBUG')

# NEW
lcd.add_formatter('msg',
                  format='%(message)s'
).add_formatter('logger_level_msg',
                format='%(name)-20s: %(levelname)-8s: %(message)s'
)

lcd.add_handler('h_stderr',
                formatter='msg',
                level='INFO',
                class_='logging.StreamHandler',
```

(continues on next page)

(continued from previous page)

```
).add_file_handler('h_file',
                   formatter='logger_level_msg',
                   level='DEBUG',
                   filename='blather.log',
)

# NEW
lcd.attach_root_handlers('h_stderr', 'h_file')

lcd.config()
```

#### Summary

As the preceding example hopefully shows, *prelogging* offers an attractive way to configure logging, one that's more straightforward, concise and easier on the eyes than the facilities provided by the *logging* package itself. The following chapters discuss basic organization and usage of `LCDictBasic` and `LCDict`. Later chapters present techniques and recipes showing how to use these classes to get more out of logging.

## 1.4 *LCDictBasic* Organization and Basic Usage

`LCDictBasic` provides an API for building dictionaries that specify Python logging configurations — *logging config dicts*. The class is fully documented in *LCDictBasic*; this chapter discusses its organization and use. Everything said here about `LCDictBasic` will also be true of its subclass `LCDict`, whose unique features we'll discuss in the next chapter.

### 1.4.1 Configuration with `LCDictBasic`

Logging configuration involves *a small hierarchy* of only four kinds of entities, which can be specified in a layered way. `LCDictBasic` lets you build a logging config dict modularly and incrementally. You add each logging entity and its attached entities one by one, instead of entering a single large thicket of triply-nested dicts.

An `LCDictBasic` instance *is* a logging config dict. It inherits from `dict`, and its methods —`add_formatter`, `add_handler`, `add_logger`, `attach_logger_handlers` and so on — operate on the underlying dictionary, breaking down the process of creating a logging config dict into basic steps.

While configuring logging, you give a name to each of the entities that you add. (Strictly speaking, you're adding *specifications* of logging objects.) When adding a higher-level entity, you identify its constituent lower-level entities by name.

Once you've built an `LCDictBasic` meeting your requirements, you configure logging by calling that object's `config` method, which passes it (`self`, a dict) to logging.config.dictConfig().

#### Specification order

- `Formatters` and `Filters` (if any) don't depend on any other logging entities, so they should be specified first.
- Next, specify `Handlers`, referencing any `Formatters` and `Filters` that the handlers use.
- Finally, specify `Loggers`, referencing any `Handlers` (and possibly `Filters`) that they use.

**Note**: `LCDictBasic` has dedicated methods for configuring the root logger (setting its level, attaching handlers and filters to it), but you can also use the class's general-purpose handler methods for this, identifying the root logger by its name, `''`.

Typically, `Filters` aren't required, and then, setting up logging involves just these steps:

1. specify `Formatters`
2. specify `Handlers` that use the `Formatters`
3. specify `Loggers` that use the `Handlers`.

**Note**: In common cases, such as the *configuration requirements* example in the previous chapter and *its solution*, `LCDict` eliminates the first step, and makes the last step trivial when only the root logger will have handlers.

### 1.4.2 Methods and properties

The `add_*` methods of `LCDictBasic` let you specify new, named logging entities. Each call to one of the `add_*` methods adds an item to one of the subdictionaries `'formatters'`, `'filters'`, `'handlers'` or `'loggers'`. In each such call, you can specify all of the data for the entity that the item describes — its loglevel, the other entities it will use, and any type-specific information, such as the stream that a `StreamHandler` will write to.

You can specify all of an item's dependencies in an `add_*` call, using names of previously added items, or you can add dependencies subsequently with the `attach_*` methods. In either case, you assign a list of values to a key of the item: for example, the value of the `handlers` key for a logger is a list of zero or more names of handler items.

The `set_*` methods let you set single-valued fields (loglevels; the formatter, if any, of a handler).

In addition to the `config` method, which we've already seen, `LCDictBasic` has methods `check` and `dump`. The properties of `LCDictBasic` correspond to the top-level subdictionaries of the underlying dict. See *LCDictBasic* for details.

#### Keyword parameters

Keyword parameters of the `add_*` methods are consistently snake_case versions of the corresponding keys that occur in statically declared logging config dicts; their default values are the same as those of *logging*. (There are just a few — rare, documented — exceptions to these sweeping statements. One noteworthy exception: `class_` is used instead of `class`, as the latter is a Python reserved word and can't be a parameter.)

For example, the keyword parameters of `add_file_handler` are keys that can appear in a (sub-sub-)dictionary of configuration settings for a file handler; the keyword parameters of `add_logger` are keys that can appear in the (sub-sub-)dicts that configure loggers. In any case, all receive sensible default values consistent with *logging*.

#### Items of a logging config dict

Here's what a minimal, "blank" logging config dict looks like:

```
>>> from prelogging import LCDictBasic
>>> d = LCDictBasic()
>>> d.dump()         # prettyprint the underlying dict
{'filters': {},
 'formatters': {},
 'handlers': {},
 'incremental': False,
 'loggers': {},
 'root': {'handlers': [], 'level': 'WARNING'},
 'version': 1}
```

Every logging config dict built by *prelogging* has the five subdictionaries and two non-dict items shown; no *prelogging* methods remove any of these items or add further items. The `LCDictBasic` class exposes the subdictionaries as properties: `formatters`, `filters`, `handlers`, `loggers`, `root`. The last, `root`, is a dict containing settings for that special logger. Every other subdict contains keys that are names of entities of the appropriate kind; the value of each such key is a dict containing configuration settings for the entity. In an alternate universe, `'root'` and its value (the `root` subdict) could be just a special item in the `loggers` subdict; but logging config dicts aren't defined that way.

### Properties

An `LCDictBasic` makes its top-level subdictionaries available as properties with the same names as the keys: `d.formatters is d['formatters']` is true, so is `d.handlers is d['handlers']`, and likewise for `d.filters`, `d.loggers`, `d.root`. For example, adding a formatter `'simple'` to `d`:

```
>>> d.add_formatter('simple')
```

changes the `formatters` collection to:

```
>>> d.formatters                # ignoring whitespace
{'simple': {'class': 'logging.Formatter',
            'format': None}
}
```

### Methods, terminology

#### The `add_*` methods

The basic `add_*` methods are these four:

```
add_formatter(self, name, format='', ... )
add_filter(self, name, ... )
add_handler(self, name, level='NOTSET', formatter=None, filters=None, ... )
add_logger(self, name, level='NOTSET', handlers=None, filters=None, ...  )
```

`LCDictBasic` also defines three special cases of `add_handler`:

```
add_stream_handler
add_file_handler
add_null_handler
```

which correspond to all the handler classes defined in the core module of `logging`. (*LCDict* defines methods for many of the handler classes defined in `logging.handlers` – see the later section, *Handler classes encapsulated by LCDict*.)

Each `add_*` method adds an item to (or replaces an item in) the corresponding subdictionary. For example, when you add a formatter:

```
>>> _ = d.add_formatter('fmtr', format="%(name)s %(message)s")
```

you add an item to `d.formatters` whose key is `'fmtr'` and whose value is a dict with the given settings:

```
>>> d.dump()
{'filters': {},
 'formatters': {'fmtr': {'format': '%(name)s %(message)s'}},
```

(continues on next page)

(continued from previous page)

```
 'handlers': {},
 'incremental': False,
 'loggers': {},
 'root': {'handlers': [], 'level': 'WARNING'},
 'version': 1}
```

The result is as if you had executed:

```
d.formatters['fmtr'] = {'class': 'logging.Formatter',
                        'format': '%(name)s %(message)s'}
```

Now, when you add a handler, you can assign this formatter to it by name:

```
>>> _ = d.add_file_handler('fh', filename='logfile.log', formatter='fmtr')
```

This `add_*_handler` method added an item to `d.handlers` — a specification for a new handler `'fh'`:

```
>>> d.dump()
{'filters': {},
 'formatters': {'fmtr': {'format': '%(name)s %(message)s'}},
 'handlers': {'fh': {'class': 'logging.FileHandler',
                     'delay': False,
                     'filename': 'logfile.log',
                     'formatter': 'fmtr',
                     'level': 'NOTSET',
                     'mode': 'a'}},
 'incremental': False,
 'loggers': {},
 'root': {'handlers': [], 'level': 'WARNING'},
 'version': 1}
```

Similarly, `add_filter` and `add_logger` add items to the `filters` and `loggers` dictionaries respectively.

### The `attach_*_*` methods

The configuring dict of a handler has an optional `'filters'` list; the configuring dict of a logger can have a `'filters'` list and/or a `'handlers'` list. The `attach_`*entity*`_`*entities* methods extend these lists of filters and handlers:

```
attach_handler_filters(self, handler_name, * filter_names)

attach_logger_handlers(self, logger_name, * handler_names)
attach_logger_filters(self, logger_name, * filter_names)

attach_root_handlers(self, * handler_names)
attach_root_filters(self, * filter_names)
```

**Note**: All these methods attach *entities* to an *entity*. Each takes a variable number of *entities* as their final parameters, and attach them to *entity*, which precedes them in the parameter list. The method names reflect the parameter order.

To illustrate, Let's add another handler, attach both handlers to the root, and examine the underlying dict:

```
>>> _ = d.add_handler('console',
...                   formatter='fmtr',
...                   level='INFO',
```

(continues on next page)

(continued from previous page)

```
...                     class_='logging.StreamHandler'
... ).attach_root_handlers('fh', 'console')
>>> d.dump()
{'filters': {},
 'formatters': {'fmtr': {'format': '%(name)s %(message)s'}},
 'handlers': {'console': {'class': 'logging.StreamHandler',
                          'formatter': 'fmtr',
                          'level': 'INFO'},
              'fh': {'class': 'logging.FileHandler',
                     'delay': False,
                     'filename': 'logfile.log',
                     'formatter': 'fmtr',
                     'level': 'NOTSET',
                     'mode': 'a'}},
 'incremental': False,
 'loggers': {},
 'root': {'handlers': ['fh', 'console'], 'level': 'WARNING'},
 'version': 1}
```

#### The `set_*_*` methods

These methods modify a single value — a loglevel, or a formatter:

```
set_handler_level(self, handler_name, level)
set_root_level(self, root_level)
set_logger_level(self, logger_name, level)
set_handler_formatter(self, handler_name, formatter_name)
```

**Note**: We might have named the last method "attach_handler_formatter", as the handler-uses-formatter relation is another example of an association between two different kinds of logging entities. However, further reflection reveals that a formatter is not "attached" in the sense of all the other `attach_*_*` methods. A handler has at most one formatter, and "setting" a handler's formatter replaces any formatter previously set; in contrast, the `attach_*_*` methods only append to and extend collections of filters and handlers, and never delete or replace items. Hence "set_handler_formatter".

---

### 1.4.3 *prelogging* warnings and consistency checking

Here's another benefit provided by *prelogging* that you don't enjoy by handing a (possibly large) dict to *logging.config.dictConfig()*': *prelogging* detects certain dubious practices and probable mistakes, and optionally prints warnings about them. In any case it automatically prevents some of those detected problems, such as attempting to attach a handler to a logger multiple times, or referencing an entity that doesn't exist (because you haven't added it yet, or mistyped its name).

#### The inner class `LCDictBasic.Warnings`

`LCDictBasic` has an inner class `Warnings` that defines bit-field "constants", or flags, which indicate the different kinds of anomalies that *prelogging* checks for, corrects when that's sensible, and optionally reports on with warning messages.

| `Warnings` "constant" | Issue a warning when. . . |
|---|---|
| `REATTACH` | attaching an entity {formatter/filter/handler} to another entity that it's already attached to |
| `REDEFINE` | overwriting an existing definition of an entity |
| `ATTACH_UNDEFINED` | attaching an entity that hasn't yet been added ("defined") |
| `REPLACE_FORMATTER` | changing a handler's formatter |

The class also defines a couple of shorthand "constants":

```
DEFAULT = REATTACH + REDEFINE + ATTACH_UNDEFINED
ALL     = REATTACH + REDEFINE + ATTACH_UNDEFINED + REPLACE_FORMATTER
```

#### `warnings` — property, parameter of `__init__`

The value of the `warnings` parameter of the `LCDictBasic` constructor can be any combination of the "constants" in the above table. Its default value is, naturally, `Warnings.DEFAULT`. The value of this parameter is saved as an `LCDictBasic` instance attribute, which is exposed by the read-write `warnings` property.

When one of these flags is "on" in the `warnings` property and the corresponding kind of offense occurs, *prelogging* prints a warning message to stderr, indicating the source file and line number of the offending method call.

#### REATTACH (default: reported)

*prelogging* detects and eliminates duplicates in lists of handlers or filters that are to be attached to higher-level entities. If `REATTACH` is "on" in `warnings`, *prelogging* will report duplicates.

#### REDEFINE (default: reported)

If this flag is "on" in `warnings`, *prelogging* warns when an existing definition of an entity is replaced, for example by calling `add_handler('h', ...)` twice.

#### `ATTACH_UNDEFINED` (default: reported)

If this flag is "on" in `warnings`, *prelogging* detects when an as-yet undefined entity is associated with another entity that uses it:

- undefined formatter assigned to a handler
- undefined filter attached to a handler
- undefined filter attached to a logger
- undefined handler attached to a logger

#### REPLACE_FORMATTER (default: not reported)

If this flag is "on" in `warnings`, *prelogging* warns when a handler that already has a formatter is given a new formatter.

#### Consistency checking — the `check` method

This method checks for references to "undefined" entities, as described above for *ATTACH_UNDEFINED*. If any exist, `check` reports that, and raises `KeyError`; otherwise, it returns `self`.

If the `Warnings.REATTACH` flag of the `warnings` property is "off", `config()` calls `check()` automatically before calling `logging.config.config()`.

## 1.5 *LCDict* Features and Usage

`LCDict` subclasses `LCDictBasic` to contribute additional conveniences. The class is fully documented in *LCDict*. In this chapter we describe the features it adds:

- *using formatter presets*
- *add_*_handler methods for several classes in logging.handlers*
- *optional automatic attaching of handlers to the root logger as they're added*
- *easy multiprocessing-safe logging*
- *simplified creation and use of filters*.

### 1.5.1 Using formatter presets

We've already seen simple examples of adding new formatters using `add_formatter`. The documentation of that method in *LCDictBasic* details its parameters and their possible values.

As our *first example* indicated, often it's not necessary to specify formatters from scratch, because *prelogging* provides an extensible, modifiable collection of *formatter presets* — predefined formatter specifications which cover many needs. You can use the name of any of these presets as the `formatter` argument to `add_*_handler` methods and to `set_handler_formatter`. *prelogging* ships with about a dozen of them, shown in *this table*.

Formatter presets are added to an `LCDict` "just in time", when they're used:

```
>>> lcd = LCDict()
>>> # The underlying dict of a "blank" LCDict
>>> #   is the same as that of a blank LCDictBasic --
>>> #   lcd.formatters is empty:
>>> lcd.dump()
{'disable_existing_loggers': False,
 'filters': {},
 'formatters': {},
 'handlers': {},
 'incremental': False,
 'loggers': {},
 'root': {'handlers': [], 'level': 'WARNING'},
 'version': 1}

>>> # Using the 'level_msg' preset adds it to lcd.formatters:
```

(continues on next page)

(continued from previous page)

```
>>> _ = lcd.add_stderr_handler('console', formatter='level_msg')
>>> lcd.dump()
{'disable_existing_loggers': False,
 'filters': {},
 'formatters': {'level_msg': {'format': '%(levelname)-8s: %(message)s'}},
 'handlers': {'console': {'class': 'logging.StreamHandler',
                          'formatter': 'level_msg',
                          'level': 'NOTSET',
                          'stream': 'ext://sys.stderr'}},
 'incremental': False,
 'loggers': {},
 'root': {'handlers': [], 'level': 'WARNING'},
 'version': 1}
```

Only `'level_msg'` has been added to `lcd.formatters`.

Of course, the dozen or so formatter presets that *prelogging* contains, aren't a comprehensive collection, and probably won't meet everyone's needs or suit everyone's tastes. Therefore *prelogging* provides two functions that let you add your own presets, and/or modify existing ones:

- `update_formatter_presets_from_file(filename)`, and
- `update_formatter_presets(multiline_str)`.

These functions, and the formats of their arguments, are described in the chapter *Formatter Presets* following this one.

---

### 1.5.2 Handler classes encapsulated by `LCDict`

*logging* defines more than a dozen handler classes — subclasses of `logging.Handler` — in the modules `logging` and `logging.handlers`. The package defines the basic stream, file and null handler classes, for which `LCDictBasic` supplies `add_*_handler` methods. Its `handlers` module defines more specialized handler classes, for about half of which (presently) `LCDict` provides corresponding `add_*_handler` methods.

#### Handler classes that LCDict configures

LCDict provides methods for configuring these *logging* handler classes, with optional "locking" support in most cases:

| method | creates | optional locking? |
|---|---|---|
| add_stream_handler | StreamHandler | yes |
| add_stderr_handler | stderr StreamHandler | yes |
| add_stdout_handler | stdout StreamHandler | yes |
| add_file_handler | FileHandler | yes |
| add_rotating_file_handler | RotatingFileHandler | yes |
| add_syslog_handler | SyslogHandler | yes |
| add_email_handler | SMTPHandler | |
| add_queue_handler | QueueHandler | |
| add_null_handler | NullHandler | |

#### Adding other kinds of handlers

The following *logging* handler classes presently have no corresponding `add_*_handler` methods:

- logging.handlers.WatchedFileHandler
- logging.handlers.TimedRotatingFileHandler
- logging.handlers.SocketHandler
- logging.handlers.DatagramHandler
- logging.handlers.MemoryHandler
- logging.handlers.NTEventLogHandler
- logging.handlers.HTTPHandler

Future versions of *prelogging* may supply methods for at least some of these. In any case, all can be configured using *prelogging*. It's straightforward to write `add_*_handler` methods for any or all of these classes, on the model of the existing methods: call `add_handler` with the appropriate handler class as value of the `class_` keyword, and pass any other class-specific key/value pairs as keyword arguments.

---

### 1.5.3 Automatically attaching handlers to the root logger

Because handlers are so commonly attached to the root logger, `LCDict` makes it easy to do that. Two parameters and their defaults govern this:

- The initializer method `LCDict.__init__` has a boolean parameter `attach_handlers_to_root` [default: `False`].

  Each instance saves the value passed to its constructor, and exposes it as the read-only property `attach_handlers_to_root`. When `attach_handlers_to_root` is true, by default the handler-adding methods of this class automatically attach handlers to the root logger after adding them to the `handlers` subdictionary.

- All `add_*_handler` methods **called on an** `LCDict`, as well as the `clone_handler` method, have an `attach_to_root` parameter [type: `bool` or `None`; default: `None`]. The `attach_to_root` parameter allows overriding of the value `attach_handlers_to_root` passed to the constructor.

  The default value of `attach_to_root` is `None`, which is interpreted to mean: use the value of `attach_handlers_to_root` passed to the constructor. If `attach_to_root` has any value other than `None`, the handler will be attached *iff* `attach_to_root` is true/truthy.

Thus, if `lcd` is an `LCDict` created with `attach_handlers_to_root=True`,

```
lcd = LCDict(attach_handlers_to_root=True, ...)
```

you can still add a handler to `lcd` without attaching it to the root:

```
lcd.add_stdout_handler('stdout', attach_to_root=False, ...)
```

Similarly, if lcd‘‘ is created with `attach_handlers_to_root=False` (the default),

```
lcd = LCDict(...)
```

you can attach a handler to the root as soon as you add it to `lcd`:

```
lcd.add_file_handler('fh', filename='myfile.log', attach_to_root=True, ...)
```

without having to subsequently call `lcd.attach_root_handlers('fh', ...)`.

---

### 1.5.4 Easy multiprocessing-safe logging

As we've mentioned, most recently in the this chapter's earlier section *Handler classes that LCDict configures*, *prelogging* provides multiprocessing-safe ("locking") versions of the essential handler classes that write to the console, streams, files, rotating files, and syslog. These subclasses of handler classes defined by *logging* are documented in *Locking Handlers*. The following `LCDict` methods:

```
add_stream_handler
add_stderr_handler
add_stdout_handler
add_file_handler
add_rotating_file_handler
add_syslog_handler
```

can create either a standard, *logging* handler or a locking version thereof. Two keyword parameters and their defaults govern which type of handler will be created:

- The initializer method `LCDict.__init__` has a boolean parameter `locking` [default: `False`].

  Each `LCDict` instance saves the value passed to its constructor, and exposes it as the read-only property `locking`. When `locking` is true, by default the `add_*_handler` methods listed above will create locking handlers.

- The `add_*_handler` methods listed above have a `locking` parameter [type: `bool` or `None`; default: `None`], which allows overriding of the value `locking` passed to the constructor.

  The default value of the `add_*_handler` parameter `locking` is `None`, which is interpreted to mean: use the value of `locking` passed to the constructor. If the `add_*_handler` parameter `locking` has any value other than `None`, a locking handler will be created *iff* the parameter's value is true/truthy.

---

### 1.5.5 Simplified creation and use of filters

Filters allow finer control than mere loglevel comparison over which messages actually get logged.

There are two kinds of filters: class filters and callable filters. `LCDict` provides a pair of convenience methods, `add_class_filter` and `add_callable_filter`, which are easier to use than the lower-level `LCDictBasic` method `add_filter`.

In Python 2, the *logging* module imposes a fussy requirement on callables that can be used as filters, which the Python 3 implementation of *logging* removes. The `add_callable_filter` method provides a single interface for adding callable filters that works in both Python versions.

#### Defining filters

Here are a couple of examples of filters, both of which suppress certain kinds of messages. Each has the side effect of incrementing a distinct global variable.

#### Class filters

Classic filters are instances of any class that implement a `filter` method with the following signature:

```
filter(self, record: logging.LogRecord) -> int
```

where `int` is treated like `bool` — nonzero means true (log the record), zero means false (don't). These include subclasses of `logging.Filter`, but a filter class doesn't have to inherit from that *logging* class.

#### Class filter example

```
_info_count = 0     # incremented by the following class filter

class CountInfoSquelchOdd():
    def filter(self, record):
        """Suppress odd-numbered messages (records) whose level == INFO,
        where the "first" message is the 0-th hence is even-numbered.

        :param self: unused
        :param record: logging.LogRecord
        :return: int -- true (nonzero) ==> let record through,
                        false (0) ==> squelch
        """
        global _info_count
        if record.levelno == logging.INFO:
            _info_count += 1
            return _info_count % 2
        else:
            return True
```

#### Callable filters

A filter can also be a callable, of signature `logging.LogRecord -> int`. (In fact, *prelogging* lets you use callables of signature `(logging.LogRecord, **kwargs) -> int`; see the section below on *providing extra, static data to callable filters* for discussion and an example.)

### Callable filter example

```
_debug_count = 0        # incremented by the following callable filter

def count_debug_allow_2(record):
    """
    Allow at most 2 messages with loglevel ``DEBUG``.

    :param record: ``logging.LogRecord``
    :return: ``bool`` -- True ==> let record through, False ==> squelch
    """
    global _debug_count
    if record.levelno == logging.DEBUG:
        _debug_count += 1
        return _debug_count <= 2
    else:
        return True
```

### Filters on the root logger

Let's configure the root logger to use both filters shown above:

```
lcd = LCDict(
    attach_handlers_to_root=True,
    root_level='DEBUG')

lcd.add_stdout_handler(
    'console',
    level='DEBUG',
    formatter='level_msg')

lcd.add_callable_filter('count_d', count_debug_allow_2)
lcd.add_class_filter('count_i', CountInfoSquelchOdd)

lcd.attach_root_filters('count_d', 'count_i')

lcd.config()
```

Now use the root logger:

```
import logging
root = logging.getLogger()

for i in range(5):
    root.debug(str(i))
    root.info(str(i))

print("_debug_count:", _debug_count)
print("_info_count:", _info_count)
```

This passage writes the following to `stdout`:

```
DEBUG   : 0
INFO    : 0
DEBUG   : 1
```

(continues on next page)

(continued from previous page)

```
INFO     : 2
INFO     : 4
_debug_count: 5
_info_count: 5
```

**Note:** This example **is** the test `test_add_xxx_filter.py`, with little modification.

### Filters on a non-root logger

Attaching the example filters to a non-root logger `'mylogger'` requires just one change: instead of using `attach_root_filters` to attach the filters to the root logger, now we have to attach them to an arbitrary logger. This can be accomplished in either of two ways:

- Attach the filters when calling `add_logger` for `'mylogger'`, using the `filters` keyword parameter:

  ```
  lcd.add_logger('mylogger',
                 filters=['count_d', 'count_i'],
                 ...
                )
  ```

  The value of the `filters` parameter can be either the name of a single filter (a `str`) or a sequence (list, tuple, etc.) of names of filters.

- Add the logger with `add_logger`, without using the `filters` parameter:

  ```
  lcd.add_logger('mylogger', ... )
  ```

  and then attach filters to it with `attach_logger_filters`:

  ```
  lcd.attach_logger_filters('mylogger',
                            'count_d', 'count_i')
  ```

### Filters on a handler

There are two ways to attach filters to a handler:

- Attach the filters in the same method call that adds the handler. Every `add_*_handler` method takes a `filters` keyword parameter — all those methods funnel through `LCDictBasic.add_handler`. As with the `add_logger` method, the value of the `filters` parameter can be either the name of a single filter (a `str`) or a sequence (list, tuple, etc.) of names of filters.

  For example, each of the following method calls adds a handler with only the `'count_d'` filter attached:

  ```
  lcd.add_stderr_handler('con-err',
                         filters='count_d'
  ).add_file_handler('fh',
                     filename='some-logfile.log',
                     filters=['count_d'])
  ```

  For another example, the following statement adds a rotating file handler with both the `'count_i'` and `'count_d'` filters attached:

```
lcd.add_rotating_file_handler('rfh',
                              filename='some-rotating-logfile.log',
                              max_bytes=1024,
                              backup_count=5,
                              filters=['count_i', 'count_d'])
```

- Add the handler using any `add_*_handler` method, then use `add_handler_filters` to attach filters to the handler. For example:

```
lcd.add_file_handler('myhandler',
                     filename='mylogfile.log'
).attach_handler_filters('myhandler',
                         'count_d', 'count_i')
```

In *a later chapter* we'll discuss providing extra data to filters, in addition to the `LogRecord`s they're called with.

## 1.6 Formatter Presets

*prelogging* provides an extensible, modifiable collection of *formatter presets* — predefined formatter specifications which you can reference by name as the `formatter` argument to `add_*_handler` and `set_handler_formatter` methods of `LCDict`, without having to first call `add_formatter`. We've already seen them in use, in the *first example of using prelogging* and in the previous chapter's section on *using formatter presets*.

When first loaded, *prelogging* provides these presets:

| Formatter name | Format string |
|---|---|
| 'msg' | '%(message)s' |
| 'level_msg' | '%(levelname)-8s:  %(message)s' |
| 'process_msg' | '%(processName)-10s:  %(message)s' |
| 'logger_process_msg' | '%(name)-20s:  %(processName)-10s: %(message)s' |
| 'logger_level_msg' | '%(name)-20s:  %(levelname)-8s: %(message)s' |
| 'logger_msg' | '%(name)-20s:  %(message)s' |
| 'process_level_msg' | '%(processName)-10s: %(levelname)-8s:  %(message)s' |
| 'process_time_level_msg' | '%(processName)-10s:  %(asctime)s: %(levelname)-8s:  %(message)s' |
| 'process_logger_level_msg' | '%(processName)-10s:  %(name)-20s: %(levelname)-8s:  %(message)s' |
| 'process_time_logger_level_msg' | '%(processName)-10s:  %(asctime)s: %(name)-20s:  %(levelname)-8s: %(message)s' |
| 'time_logger_level_msg' | '%(asctime)s:  %(name)-20s: %(levelname)-8s:  %(message)s' |

This collection is by no means comprehensive, nor could it be. (*logging* recognizes about 20 keywords in format

strings; you can even use your own keywords, as shown in *Adding custom fields and data to messages*; these can all be combined in infinitely many format strings.) The names of these presets probably won't be to everyone's liking either (`level` not `levelname`; `msg` and `process`, which are themselves recognized keywords, rather than `message` and `processName`). Nevertheless, formatter presets are a useful facility, especially across multiple projects. Therefore, *prelogging* lets you add your own formatter presets, and/or modify existing ones. Two functions make that possible:

- `update_formatter_presets(multiline_str)` reads descriptions of formatters in a multiline string;
- `update_formatter_presets_from_file(filename)` reads descriptions of formatters from a text file;

Both functions update the collection of formatter presets.

**Note**: The changes and additions made by these functions do **not** persist after your program exits.

Generally, you call one of these functions, once, after importing *prelogging* or things from it, and before creating an `LCDict` and populating it using your new or improved formatter presets.

The following subsections describe these functions and the expected formats of their arguments. It's convenient to present the file-based function first.

---

### 1.6.1 The `update_formatter_presets_from_file` function

This function basically passes the contents of the file to `update_formatter_presets`, described below.

#### File format

This functions expects a text file consisting of:

- zero or more blank lines, followed by
- zero or *formatter descriptions*, all separated by one or more blank lines.

A blank line consists only of whitespace. A *formatter description* is a group of lines consisting of a *name*, beginning in column 1 on a line by itself, followed by one or more indented lines each containing a *key* `:` *value* pair, and all subject to the following conditions:

- Each *key* must be one of `format`, `dateformat`, `style`. `format` is required; the others are optional.
- If a *value* contains spaces then it should be enclosed in quotes (single or double); otherwise, enclosing quotes are optional (any outermost matching quotes are removed).
- A *name* can contain spaces, and does not have to be quoted unless you want it to have initial or trailing whitespace.
- In a *key* `:` *value* pair, zero or more spaces may precede and follow the colon.
- The *value* given for `style` should be one of `%` `{` `$`; if `style` is omitted then it defaults to `%`. (Under Python 2, only `%` is allowed, so if you're still using that then you should omit `style`.)

These keys and values are as in the LCDictBasic.add_formatter method.

#### Example 1 – basic and corner cases

Here's an example of a valid/well-formed file (assume the names begin in column 1):

```
name_level_message
    format: '%(name)s - %(levelname)s - %(message)s'

name level message
    format: '%(name)s - %(levelname)s - %(message)s'

datetime_name_level_message
    format    : '{asctime}: {name:15s} - {levelname:8s} - {message}'
    dateformat: '%Y.%m.%d %I:%M:%S %p'
    style: {

'    his formatter    '
    format:%(message)s
```

If the file passed to `update_formatter_presets_from_file` has ill-formed contents, the function writes an appropriate error message to `stderr`, citing the file name and offending line number, and the collection of formatter presets remains unchanged.

#### Example 2 – `formatter_presets.txt` declares *prelogging*'s formatter presets

Another example of a valid file containing formatter presets is the text file `formatter_presets.txt` in the *prelogging* directory. *prelogging* creates its stock of formatter presets by calling

`update_formatter_presets_from_file(`*path/to/*`'formatter_presets.txt')`

when the `lcdict` module is loaded.

---

### 1.6.2 The `update_formatter_presets` function

For example, all of these are equivalent well-formed possible arguments:

```
# <-- assume that's column 1

    s1 = '''\
myformatter
    format: '%(message)s'
    style: '%'
'''
    s2 = '''\
    myformatter
        format: '%(message)s'
            style: '%'
    '''
    s2 = '''\
        myformatter
          format: '%(message)s'
          style: '%'
    '''
```

Note that each triple-quote beginning a multiline string is followed by `\`, so that the logical line *n* is not actually line *n*+1.

If the string passed to `update_formatter_presets` is ill-formed, the function writes an appropriate error message to `stderr`, citing the offending line number, and the collection of formatter presets remains unchanged.

## 1.7 Configuring Loggers

We have already seen examples of how easy it can be to configure the root logger — for example, with both a console handler and a file handler, as in the *overview*.

This chapter is mainly concerned with configuring non-root loggers. We'll begin by considering the special case of "configuring" non-root loggers by not configuring them at all, so that the root does all the work via propagation.

For simplicity the examples in this chapter use the root logger and non-root loggers, but they can be adapted to the more general situation of a non-propagating logger with handlers, and its descendants.

- **Configuring non-root loggers by propagation to the root**
  - *Using non-root loggers without configuring them*
- **Configuring non-root loggers; using root and non-root loggers together**
  - *Example: A "discrete" non-root logger*
  - *Best practices for propagation and handler placement*

---

### 1.7.1 Using non-root loggers without configuring them

A common, useful approach is to attach handlers only to the root logger, and then have each module log messages using `logging.getLogger(__name__)`. These "child" loggers require no configuration; they use the handlers of the root because, by default, loggers are created with `propagate=True` (and with `level='NOTSET'`).

If the formatters of the handlers include the logger name — as the formatter preset `logger_level_msg` does, for example — each logged message will state which logger wrote it.

The following example illustrates the general technique:

```
>>> from prelogging import LCDict
>>> import logging
>>> lcd = LCDict(attach_handlers_to_root=True)
>>> lcd.add_stdout_handler('con', formatter='logger_level_msg')
>>> lcd.config()
```

```
>>> logging.getLogger().warning("Look out!")
root                : WARNING : Look out!
>>> logging.getLogger('my_submodule').warning("Something wasn't right.")
my_submodule        : WARNING : Something's wasn't right.
>>> logging.getLogger('your_submodule').error("Uh oh, there was an error.")
your_submodule      : ERROR   : Uh oh, there was an error.
```

---

### 1.7.2 Configuring and using non-root loggers

In the previous section we saw one common configuration of non-root loggers. Other configurations are possible and sometimes desirable:

- you want the logger for a module or package to have a different loglevel from that of the root, but to use the same handlers as the root (thus, it will write to the same destination(s));
- you want to write to destinations other than those of the root, either instead of or in addition to those.

The first case is easily achieved simply by setting the loglevel of the non-root logger as desired, giving it no handlers; propagation takes care of the rest (a logger's `propagate` property is, by default, true).

The second case has many variations, depending upon whether the non-root logger propagates or not. We consider a non-propagating example, where the non-root logger is totally "walled off" from the root logger. Variations will be easy to devise and configure.

### Example: A "discrete" non-root logger

In this example we use two loggers: the root, and another logger that's "discrete" from the root, and indeed from any ancestor logger, in the sense that:

- it doesn't share any handlers with any ancestor, and
- it doesn't propagate to any ancestor.

As the root is an ancestor of every logger, in particular we'll require that the added logger should *not* attach its handlers to the root, and that it should not "propagate" to its parent (the root, in this example).

### Requirements

- root logger with a `stderr` console handler at loglevel `WARNING`, and a file handler at default loglevel `NOTSET`;
- a discrete logger, named let's say `'extra'`, with loglevel `DEBUG`, which will write to a different file using a handler at default loglevel `NOTSET`;
- logfiles should be in the `_log/` subdirectory of the current directory.

### How-to

Start with an `LCDict` that uses standard (non-locking) stream and file handlers; use root loglevel `'DEBUG'`; set `log_path` as required:

```
import logging
from prelogging import LCDict


lcd = LCDict(log_path='_log/',
             root_level='DEBUG',
             attach_handlers_to_root=True)
```

Set up the root logger with a `stderr` console handler at loglevel 'WARNING', and a file handler at its default loglevel 'NOTSET':

```
lcd.add_stderr_handler('console',
                       level='WARNING',
                       formatter='msg'
).add_file_handler('root_fh',
                   filename='root.log',
                   formatter='logger_level_msg')
```

Add an `'extra'` logger, with loglevel `DEBUG`, which will write to a different file using a handler at default loglevel `NOTSET`. Note the use of parameters `attach_to_root` and `propagate`:

- in the `add_file_handler` call, passing `attach_to_root=False` ensures that this handler *won't* be attached to the root logger, overriding the `lcd` default value established by `attach_handlers_to_root=True` above;
- in the `add_logger` call, `propagate=False` ensures that messages logged by `'extra'` don't also bubble up to the root and its handlers:

```
lcd.add_file_handler('extra_fh',
                     filename='extra.log',
                     formatter='logger_level_msg',
                     attach_to_root=False
).add_logger('extra',
             handlers=['extra_fh'],
             level='DEBUG',
             propagate=False)
```

Finally, call `config()` to create actual objects of *logging* types — `logging.Formatter`, `logging.Logger`, etc.

```
lcd.config()
```

Now `lcd` is actually no longer needed (we don't do 'incremental' configuration, but then, arguably, neither does *logging*).

To use the loggers, access them by name:

```
# 'extra' writes "Hi there" to file `_log/extra.log`:
logging.getLogger('extra').warning("Hi there.")

# Root writes "UH OH" to `stderr` and to `_log/root.log`:
logging.getLogger().error("UH OH")

# Root writes "ho hum" to `_log/root.log` only:
logging.getLogger().debug("ho hum")
```

**Exercise**: Verify the claimed effects of the `attach_to_root` and `propagate` parameters in the two calls that configure the `'extra_fh'` handler and the `'extra'` logger.

1. Comment out `attach_to_root=False` from the `add_file_handler` call for `'extra_fh'`.

   Now, `'extra_fh'` is a handler of the root logger *too*, so it logs its messages `"UH OH"` and `"ho hum"` to `_log/extra.log`, as well as to `root.log` and `stderr` as before.

   `_log/root.log` contains:

   ```
   root                : ERROR   : UH OH
   root                : DEBUG   : ho hum
   ```

   `_LOG/extra.log` contains:

   ```
   extra               : WARNING : Hi there.
   root                : ERROR   : UH OH
   root                : DEBUG   : ho hum
   ```

   `stderr` output:

   ```
   UH OH
   ```

2. Uncomment `attach_to_root=False` in the `add_file_handler` call, and comment out `propagate=False` from the `add_logger` call.

Now, `'extra'` writes to the root's handlers as well as its own, so it logs a warning `"Hi there."` to both `stderr` and `_log/root.log`.

`_log/root.log` contains:

```
extra               : WARNING : Hi there.
root                : ERROR   : UH OH
root                : DEBUG   : ho hum
```

`_log/extra.log` contains:

```
extra               : WARNING : Hi there.
```

`stderr` output:

```
Hi there.
UH OH
```

---

### 1.7.3 Best practices for propagation and handler placement

The examples in this chapter, and the preceding Exercise, have hopefully conveyed the significance of propagation and the importance of "right" handler placement. Now is a good time to reflect further on these matters.

According to the documentation for Logger.propagate,

> if [a logger's `propagate` property] evaluates to true [the default], events logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handlers attached to this logger. Messages are passed directly to the ancestor loggers' handlers - neither the level nor filters of the ancestor loggers in question are considered.

This suggests that truly intricate, and no doubt surprising, configurations can be achieved using propagation and fussy placements of handlers on loggers. The **Note** at the end of the above link clearly states best practice:

> If you attach a handler to a logger and one or more of its ancestors, it may emit the same record multiple times. In general, you should not need to attach a handler to more than one logger - if you just attach it to the appropriate logger which is highest in the logger hierarchy, then it will see all events logged by all descendant loggers, provided that their propagate setting is left set to True. A common scenario is to attach handlers only to the root logger, and to let propagation take care of the rest.

## 1.8 Further Topics and Recipes

- **Configuration distributed across multiple modules or packages**
  - *Using LCDictBuilderABC*
  - *Migrating a project that uses dynamic configuration to prelogging*
  - *Migrating a project that uses static dict-based configuration to prelogging*

### 1.8.1 Using `LCDictBuilderABC`

One way for a larger program to configure logging is to pass around an `LCDict` to the different "areas" of the program, each area contributing specifications of the logging entities it will use. The `LCDictBuilderABC` class provides a mini-microframework that automates this approach: each area of a program need only define an `LCDictBuilderABC` subclass and override its method `add_to_lcdict(lcd)`, where it contributes its specifications by calling methods on `lcd`.

The *LCDictBuilderABC* documentation describes how that class and its two methods operate. The test `tests/test_lcdict_builder.py` illustrates using the class to configure logging across multiple modules.

#### Migrating a project that uses dynamic configuration to *prelogging*

First a caveat: If your program uses the *logging* API throughout the course of its execution to create or (re)configure logging entities, then migration to *prelogging* may offer little gain: many of the runtime calls to *logging* methods probably can't be replaced. In particular, obviously *prelogging* provides no means to delete or detach logging entities.

However, if your program uses the *logging* API to configure logging only at startup, in a "set it and forget it" way, then it's probably easy to migrate it to *prelogging*. Benefits of doing so include clearer, more concise code, and access to the various amenities of *prelogging*.

#### Migrating a project that uses static dict-based configuration to *prelogging*

A common pattern for a large program that uses static dict-based configuration is to pass around a single (logging config) dict to each "area" of the program; each "area" adds its own required entities and possibly modifies those already added; finally a top-level routine passes the dict to `logging.config.dictConfig`.

Let's suppose that each program "area" modifies the logging config dict in a function called `add_to_config_dict(d: dict)`. These `add_to_config_dict` functions performs dict operations on the parameter `d` such as

```
d['handlers']['another_formatter'] = { ... }
```

and

`d.update( ... )`.

*Assuming your* `add_to_config_dict` *functions use "duck typing" and work on any parameter* `d` *such that* `isinstance(d, dict)` *is true, they should continue to work properly if you pass them an* LCDict.

Thus, the `add_to_config_dict` function specific to each program area can easily be converted to an `add_to_lcdict(cls, lcd: LCDict)` classmethod of an *LCDictBuilderABC* subclass specific to that program area.

---

### 1.8.2 Multiprocessing — two approaches

The section of the *logging* Cookbook entitled Logging to a single file from multiple processes begins by admitting that "logging to a single file from multiple processes is not supported". It goes on to discuss three approaches to providing this capability:

1. using `SocketHandler`
2. developing locking versions of handlers, the approach taken by *prelogging* with its "locking handlers"
3. (*Python 3 only*) using a `QueueHandler` in each process, all writing to a common Queue, and then using either a `QueueListener` or a dedicated thread in another process (e.g. the main one) to extract `LogRecord`s from the queue and log them.

**Note**: the third approach is unavailable in Python 2 because the class `QueueHandler` is Python 3 only.

The `examples/` top-level directory of the *prelogging* distribution contains several multiprocessing examples. See the *Filter examples* section of the *Guide to Examples* for a list of them with descriptions of what each one does.

In this section we'll discuss the second and third approaches.

**Basic situation and challenge**

Suppose we have some significant amount of computational work to do, and the code that performs it uses logging. Let's say there are $L$ many loggers used:

$$logger_1, \cdots, logger_i, \cdots, logger_L,$$

Each logger $logger_i$ is denoted by some name $name_i$, and has some intended handlers:

$$handler_{i,j} \quad (j < n_i).$$

Later, we notice that the work can be parallelized: we can partition it into chunks which can be worked on simultaneously and the results recombined. We put the code that performs the work into a function, and spawn $N$ worker processes

$$P_1, ..., P_k, ...P_N,$$

each of which runs that function on a discrete chunk of the data. The worker processes are basically homogeneous, except for their distinct PIDs, names, and the ranges of data they operate on. Now, each worker process $P_k$ uses all the loggers $logger_i, i < L$. The loggers and handlers all have different instances in different processes; however, all the handler destinations remain unique. Somehow, we have to serialize writing to single destinations from multiple concurrent processes.

**Two solutions**

In the approach provided natively by *prelogging*, serialization occurs at the ultimate outputting handlers, using the package's simple "locking handler" classes. Before an instance of a locking handler writes to its destination, it acquires a lock (*shared by all instances* of the handler), which it releases when done; attempts by other instances to write concurrently will block until the lock is released by the handler that "got there first".

The queue-based approach is an important and sometimes more performant alternative. Using an explicit shared queue and a layer of indirection, this approach serializes messages early in their lifecycle. Each process merely enqueues logged messages to the shared queue, in the form of `LogRecord`s. The actual writing of messages to their intended destinations occurs later, in a dedicated *logging thread* of a non-worker process. That thread pulls logging records off the queue and *handles* them, so that messages are finally dispatched to their intended handlers and destinations. The *logging* package's `QueueHandler` class makes all this possible.

---

**Note:** The *prelogging* examples contain a pair of programs that are "the same" except that each takes a different approach to multiprocessing:

- `mproc_approach__locking_handlers.py` uses locking handlers,
- `mproc_approach__queue_handler_logging_thread.py` uses a queue and logging thread (the only example that does so).

In these examples, the handlers only write to local files, and performance of the two approaches is about the same, with the queue-based approach slightly faster.

---

### Using locking handlers

*prelogging* provides multiprocessing-safe logging natively by using locking handlers — subclasses of certain *logging* handler classes which use locks to serialize their output. As only Python 3 implements `QueueHandler`s, this is the only option easily available under Python 2 for multiprocessing-safe logging with *prelogging*.

All but one of the multiprocessing examples use locking handlers — see *Filter examples* in the *Guide to Examples* for an overview. Those examples illustrate the use of every locking handler. The section *Easy multiprocessing-safe logging* in the chapter *LCDict Features and Usage* explains how to use the Boolean `locking` parameter to enable locking. These resources more than suffice to explain how to take advantage of the simple interface that *prelogging* provides to its locking handlers.

### Using QueueHandlers (*Python 3 only*)

The queue-based approach serializes logged messages as soon as possible, moving the actual writing of messages out of the worker processes. Worker processes merely enqueue messages, with context, onto a common queue. The real handlers don't run in the worker processes: they run in a dedicated thread of the main process, where records are dequeued from that common queue and handled in the ways you intend.

When a worker process $P_k$ logs a message using one of the loggers $logger_i$, none of the "real", intended handlers of that logger executes in $P_k$. Instead, the message, in the form of a `logging.LogRecord`, is put on a `Queue` object which all the processes share. The enqueued record contains all information required to write it later, even in another process. This is all achieved by a simple logging configuration that uses *logging*'s `QueueHandler` class.

In a dedicated thread in another process — the main process, let's assume — a tight loop polls the shared queue and pulls records from it. Each record contains context information from the originating process $P_k$, including the logger's name, the message's loglevel, the process name of $P_k$ — values for the keys that can occur in format strings. The

thread uses this information to dispatch the record via the originating logger, and finally the intended handlers execute. This setup too is easily achieved with an appropriate logging configuration.

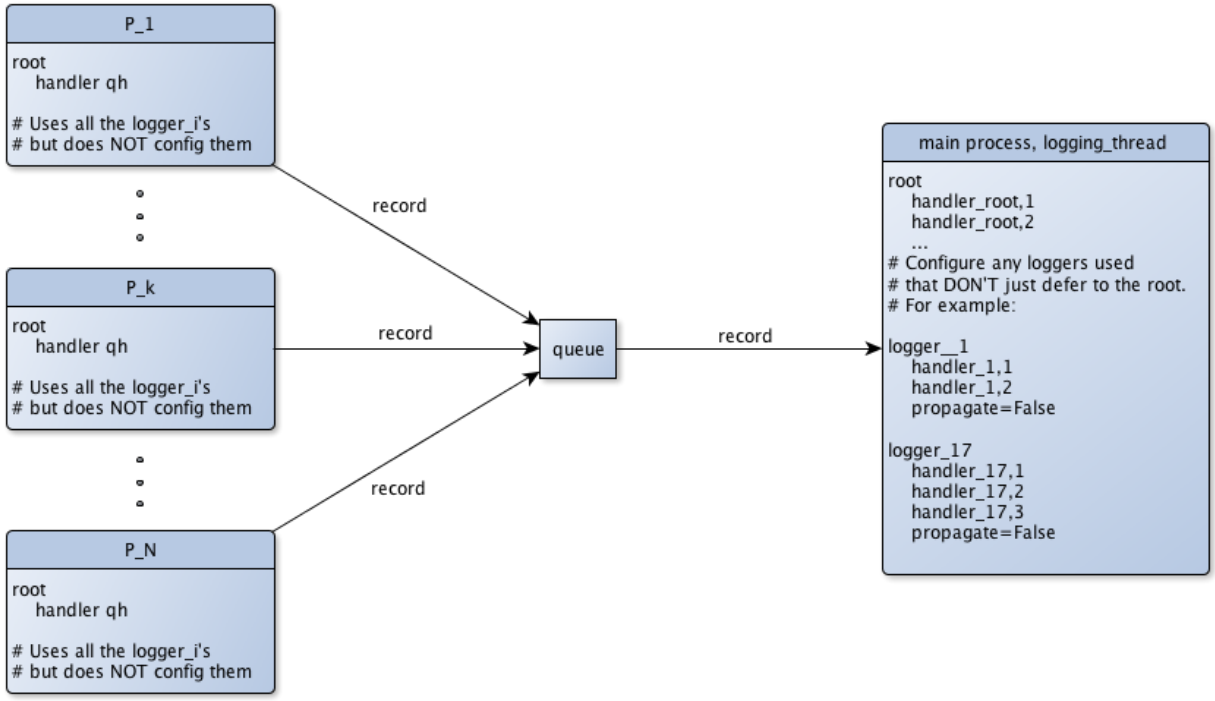



Fig. 2: Multiprocess logging with a queue and a logging thread

This design gives better performance, especially for blocking, slow handlers (SMTP, for example). Generally, the worker processes have better things to do than wait for emails to be successfully sent, so we relieve them of such extraneous burdens.

Handling all logged messages in a dedicated thread (of a non-worker process) confers additional benefits:

- the UI won't stutter or temporarily freeze whenever a slow (and blocking) handler runs;
- the main thread can do other useful things.

The queue-based approach confers these same benefits even in single-processing situations. The example `queue_handler_listener.py` illustrates this, using the logging package's `QueueListener` instead of a logging-thread. `QueueListeners` encapsulate setup and teardown of a logging thread, and the proper handling of queued messages. It's unfortunate that they're an awkward fit for static configuration.

**Aside: `QueueListeners` and static configuration**

It's awkward to use a `QueueListener` with static configuration. Once it has been created, a `QueueListener` has to be stopped and started, using its `stop` and `start` methods. If we could statically specify a `QueueListener`, somehow we have to obtain a reference it after configuring logging, in order to call these methods.

Furthermore, a `QueueListener` must be initialized/constructed with one or more `QueueHandlers` – actual handler objects. Of course, these don't exist before configuration, and then the names we gave them in configuration have disappeared. As we've noted elsewhere, handler objects are anonymous, so the only way to obtain references

to the `QueueHandlers` is a bit disappointing (filter the handlers of some logger with `isinstance(handler, QueueHandler)`). The example `queue_handler_listener.py` demonstrates this in action.

#### Worker process configuration

The main process creates a common queue, then spawns the worker processes $P_k$, passing the queue to each one. The worker processes *use, but do not configure* the intended loggers $logger_i$. In the logging configuration of the worker processes, these loggers have *no handlers*. Thus, because of inheritance, all messages are actually logged by their common ancestor, the root logger. The root is equipped with a single handler: a `QueueHandler` (`qh` in the diagram above), which puts messages on the queue it's initialized with.

At startup, every worker process configures logging in this simple way:

```
def worker_config_logging(q: Queue):
    d = LCDict(root_level='DEBUG')
    d.add_queue_handler('qhandler', attach_to_root=True, queue=q)
    d.config()
```

#### *logging thread*/main process configuration

The logging thread does one thing: dispatch logged messages to their ultimate destinations as they arrive. Before the main process creates the logging thread, it configures logging as you really intend. The configuration used here is essentially what you would use in the locking handlers approach (but with `locking=False`). The logging configuration specifies all the intended loggers $logger_i$, after specifying, for each logger, all of its intended handlers $handler_{i,j}, j < n_i$ and any formatters they use. As a result, the "real" handlers finally execute.

Here's what the logging thread does:

```
def logging_thread(q):
    while True:
        record = q.get()
        if record is None:
            break
        logger = logging.getLogger(record.name)
        logger.handle(record)
```

### 1.8.3 Using *prelogging* in libraries: using a null handler

The `add_null_handler` method configures a handler of class `logging.NullHandler`, a do-nothing, placeholder handler that's useful in writing libraries (packages).

If you want your library to write logging messages *only* if its user has configured logging, the *logging* docs section Configuring Logging for a Library recommends adding a `NullHandler`, only, to the library's top-level logger.

The example `use_library.py` and the `library` package it uses illustrate how to use *prelogging* to follow that recommendation and achieve such a setup. It's essential that both the library and its user set the logging configuration flag `disable_existing_loggers` to `False`. This is actually *prelogging*'s default — one of the few instances where *prelogging* changes the default used by *logging* (the *logging* package defaults `disable_existing_loggers` to `True`).

In this section we'll further discuss the configurations and interaction of the example library and library user.

### `library` use of *prelogging* and *logging*

The package contains just two modules: `__init__.py` and `module.py`.

`__init__.py` configures logging with *prelogging*, adding a null handler and attaching it to the library's "top-level logger", `'library'`:

```
lcd = LCDict()                  # default: disable_existing_loggers=False
lcd.add_null_handler('library-nullhandler')    # default: level='NOTSET'
lcd.add_logger('library', handlers='library-nullhandler', level='INFO')
lcd.config()
```

`module.py` contains two functions, which use logging:

```
def do_something():
    logger = logging.getLogger(__name__)
    logger.debug("DEBUG msg")
    logger.info("INFO msg")
    print("Did something.")

def do_something_else():
    logging.getLogger(__name__ + '.other').warning("WARNING msg")
    print("Did something else.")
```

If a user of *library* configures logging, the messages logged by these functions *will* actually be written; if it doesn't, those messages *won't* appear.

### `use_library.py` use of *prelogging* and *logging*

The example `use_library.py` makes it easy to explore the various possibilities. It contains a simple `main()` function, which the program calls when run as `__main__`:

```
def main():
    # Exercise:
    #   Comment out and uncomment the following two lines, independently;
    #   observe the console output in each case.
    logging_config()
    logging.getLogger().warning("I must caution you about that.")

    library.do_something()
    library.do_something_else()
```

and a simple `logging_config` function:

```
def logging_config():
    d = LCDict(attach_handlers_to_root=True)
    # defaults: disable_existing_loggers=False, root_level='WARNING'
    d.add_stdout_handler('stdout', formatter='logger_level_msg', level='DEBUG')
    d.config()
```

### Results (4 cases)

1. With both lines uncommented, the program writes the following to stdout:

```
root                 : WARNING : I must caution you about that.
library.module       : INFO    : INFO msg
Did something.
library.module.other: WARNING : WARNING msg
Did something else.
```

**Note**: The loglevel of the root logger, configured in the library's user, is `'WARNING'`, whereas the loglevel of the `'library.module'` logger is `'INFO'`. Although `'WARNING'` is more restrictive than `'INFO'`, propagated messages

> are passed directly to the ancestor loggers' handlers –
> neither the level nor filters of the ancestor loggers in question
> are considered.
>
> (*from the* 'propagate' *documentation*)

In our example, messages of *library* propagate to the root, and those of level `` `INFO` `` and up (not just `` `WARNING` `` and up) *are logged*.

2. With just `logging_config()` commented out, the library prints these to stdout:

```
Did something.
Did something else.
```

and `use_library.py` logs this line to stderr (possibly between or after those printed to stdout):

```
I must caution you about that.
```

Observe that the library's logged messages are **not** written, even though the library's user *uses* logging (with the default configuration).

3. With `logging_config()` uncommented but the line following it commented out, the program writes the following to stdout:

```
library.module       : INFO    : INFO msg
Did something.
library.module.other: WARNING : WARNING msg
Did something else.
```

4. With both lines commented out, the program writes the following to stdout:

```
Did something.
Did something else.
```

### 1.8.4 Using *prelogging* with *Django*

Django uses Python logging for its logging needs, and supplies several classes that build on the facilities of the *logging* package. However, none of its additions address configuration. Fortunately, it's quite easy to use *prelogging* in conjunction with Django.

#### Setting the `LOGGING` variable in `settings.py`

Django uses logging config dicts: the easiest way to configure logging in Django is to provide a logging config dict as the value of the `LOGGING` variable in `settings.py`. Of course, you can use *prelogging* to build an `LCDict`; just refrain from calling its `config` method, as Django will pass the `LOGGING` dict to `dictConfig`.

The general approach:

- Write a function that builds and returns an `LCDict`, perhaps by using the *LCDictBuilderABC class*. For the sake of example, say the function is `build_settings_lcdict`, in module `mystuff`.
- Add the following two lines to your Django project's `settings.py`, either contiguous or not:

```
from mystuff import build_settings_lcdict
LOGGING = dict(build_settings_lcdict())
```

*build_settings_lcdict* builds a logging config dict but doesn't call its `config` method. Django will add its logging specifications to the `LOGGING` dict and then pass that to `logging.config.dictConfig`.

---

### 1.8.5 Providing extra data to a filter

Often you'll want the behavior of a filter to depend on more than just the `LogRecord` that's passed to it. In the first subsection of this topic, we'll see how to provide a filter with extra data that doesn't change. In *the second subsection*, we'll discuss how to provide a filter with dynamic data, whose value may be different each time the filter is called.

#### Providing extra, static data to a filter

It's simple to provide a filter with extra, unchanging data, and in this section we'll see how to do so.

#### Class filter

The `add_class_filter` method has the following signature:

```
def add_class_filter(self, filter_name, filter_class, **filter_init_kwargs):
    """
    filter_init_kwargs: any other parameters to be passed to `add_filter`.
                        These will be passed to the `filter_class` constructor.
                        See the documentation for `LCDictBasic.add_filter`.
    Return: self
    """
```

When logging is configured, the class `filter_class` is instantiated, and its `__init__` method is called. If the signature of `__init__` includes `**kwargs`, that dict will contain all the keyword parameters in `filter_init_kwargs`. Thus, the filter class's `__init__` can save values in `kwargs` as instance attributes, for later use by the `filter` method.

The following example (basically `examples/filter-class-extra-static-data.py`) illustrates this scenario:

```
import logging
from prelogging import LCDict

class CountAndSquelchOdd():
    def __init__(self, *args, **kwargs):
        self.level_count = 0

        print(kwargs)
        self.filtername = kwargs.get('filtername', '')
```

(continues on next page)

(continued from previous page)

```
        self.loglevel_to_count = kwargs.get('loglevel_to_count', 0)

    def filter(self, record):
        """Suppress odd-numbered messages (records)
        whose level == self.loglevel_to_count,
        where the "first" message is 0-th hence even-numbered.

        Returns int or bool -- not great practice, but just to distinguish
        which branch of if-then-else was taken.
        """
        if record.levelno == self.loglevel_to_count:
            self.level_count += 1
            ret = self.level_count % 2          # int
        else:
            ret = True                          # bool

        print("{:11s}: record levelname = {}, self.level_count = {}; returning {}".
              format(self.filtername, record.levelname,
                     self.level_count, ret))
        return ret
```

Now configure logging:

```
lcd = LCDict(attach_handlers_to_root=True,
             root_level='DEBUG')
lcd.add_stdout_handler('console-out',
                       level='DEBUG',
                       formatter='level_msg')
lcd.add_class_filter('count_debug', CountAndSquelchOdd,
                     # extra, static data
                     filtername='count_debug',
                     loglevel_to_count=logging.DEBUG)
lcd.add_class_filter('count_info', CountAndSquelchOdd,
                     # extra, static data
                     filtername='count_info',
                     loglevel_to_count=logging.INFO)
lcd.attach_root_filters('count_debug', 'count_info')

lcd.config()
```

The call to `lcd.config()` creates two instances of `CountAndSquelchOdd`, which print their `kwargs` to `stdout` in `__init__`. Here's what they print:

```
{'filtername': 'count_info', 'loglevel_to_count': 20}
{'filtername': 'count_debug', 'loglevel_to_count': 10}
```

Finally, let's use the root logger:

```
for i in range(2):
    print("\ni ==", i)
    logging.debug(str(i))    # root has a handler, so no format surprises
    print("---")
    logging.info(str(i))     # no format surprises
```

This loop prints the following to stdout:

```
i == 0
count_debug: record levelname = DEBUG, self.level_count = 1; returning 1
count_info : record levelname = DEBUG, self.level_count = 0; returning True
DEBUG    : 0
---
count_debug: record levelname = INFO, self.level_count = 1; returning True
count_info : record levelname = INFO, self.level_count = 1; returning 1
INFO     : 0

i == 1
count_debug: record levelname = DEBUG, self.level_count = 2; returning 0
---
count_debug: record levelname = INFO, self.level_count = 2; returning True
count_info : record levelname = INFO, self.level_count = 2; returning 0
```

When `logging.debug(str(1))` is called, only one line is printed. The `'count_debug'` filter returns 0, which suppresses not only the logger's message, but also any calls to the logger's other filters – `count_info`, in this case.

When `logging.info(str(1))` is called, two lines are printed. `'count_debug'` returns `True`, so `count_info` is called; it returns 0, suppressing the logger's message.

### Callable filter

You can also pass extra, static data to a callable filter by passing additional keyword arguments and their values to `add_callable_filter`. Here's the signature of that method, and part of its docstring:

```
def add_callable_filter(self, filter_name, filter_fn, **filter_init_kwargs):
    """
    filter_fn: a callable, of signature
            (logging.LogRecord, **kwargs) -> bool.
        A record is logged iff this callable returns true.
    filter_init_kwargs: Keyword arguments that will be passed,
        with these same values, to the filter_fn **each time it is called**.
        (So, this method is something like "partial" -- it provides
         a kind of Currying.)
    return: self
    """
```

The example `filter-callable-extra-static-data.py`) illustrates using a callable filter. As it's quite similar to the class filter example above, there's no need to walk through the code here.

### Providing extra, dynamic data to a filter

Sometimes you may want a filter to access dynamic data, whose value may be different from one filter call to the next. Python doesn't provide references or pointers to immutable types, so the usual workaround would be to pass a list or dict containing the value. The value of the item in the wrapping collection can be changed dynamically, and any object that retained a reference to the containing collection would see those changes reflected. The following code illustrates this idiom, using a list to wrap an integer:

```
>>> class A():
...     def __init__(self, list1=None):
...         self.list1 = list1
...
...     def method(self):
...         print(self.list1[0])
```

```
>>> data_wrapper = [17]
>>> a = A(list1=data_wrapper)
>>> a.method()
17
>>> data_wrapper[0] = 101
>>> a.method()
101
```

This approach won't work with logging configuration.

### Configuring logging "freezes" lists and dicts in the logging config dict

While you're still building a logging config dict, the subdict for an added filter will reflect changes to any data that's accessible through dict or list references you've passed as keyword arguments. For example,

```
>>> def my_filter_fn(record, list1=None):
...     assert list1
...     print(list1[0])
...     return list1[0] > 100
```

```
>>> data_wrapper = [17]
>>> lcd = LCDict(attach_handlers_to_root=True, root_level='DEBUG')
>>> lcd.add_stdout_handler('con', formatter='msg', level='DEBUG')
>>> lcd.add_callable_filter('callable-filter',
...                         my_filter_fn,
...                         list1=data_wrapper)
>>> lcd.attach_root_filters('callable-filter')
>>> lcd.filters['callable-filter']['list1']
[17]
>>> data_wrapper[0] = 21
>>> lcd.filters['callable-filter']['list1']
[21]
```

However, once you configure logging, any such live references are broken, because the values in the dict are copied. Let's confirm this. First, configure logging with the dict we've built:

```
>>> lcd.config()
```

Now log something. The filter prints the value of `list1[0]`, which is `21`; thus it returns `False`, so no message is logged:

```
>>> logging.debug("data_wrapper = %r" % data_wrapper)
21
```

Now change the value of `data_wrapper[0]`:

```
>>> data_wrapper[0] = 101
```

Prior to configuration, the filter's `list1` referred to `data_wrapper`; but that's no longer true: `list1[0]` is still `21`, not *101*, so the filter still returns `False`:

```
>>> logging.debug("data_wrapper = %r" % data_wrapper)
21
```

### Successfully passing dynamic data

The moral of the story: if you want to pass dynamic data to a filter, you can't use a list or dict as a container (nor, of course, a tuple). The following example shows a successful strategy, using a simple ad-hoc class as a container:

```
>>> class DataWrapper():
...     def __init__(self, data=None):  self.data = data
...     def __str__(self):              return "%r" % self.data
```

```
>>> def my_filter_fn(record, data_wrapper=None):
...     assert data_wrapper
...     print(data_wrapper)
...     return isinstance(data_wrapper.data, int) and data_wrapper.data > 100
```

```
>>> dw = DataWrapper(17)
```

```
>>> lcd = LCDict(attach_handlers_to_root=True, root_level='DEBUG')
>>> lcd.add_stdout_handler('con', formatter='msg', level='DEBUG')
>>> lcd.add_callable_filter('callable-filter',
...                         my_filter_fn,
...                         data_wrapper=dw)
>>> lcd.attach_root_filters('callable-filter')
>>> lcd.filters['callable-filter']['data_wrapper'])
17
>>> dw.data = 21
>>> lcd.filters['callable-filter']['data_wrapper'])
21
```

```
>>> lcd.config()
```

```
>>> # filter prints 21 and returns False:
>>> # in the filter, data_wrapper.data == 21
>>> logging.debug("dw = %s" % dw)
21
```

```
>>> dw.data = 101
>>> # In the filter, data_wrapper.data == 101,
>>> #  so message is logged:
>>> logging.debug("dw =", dw)
101
dw = 101
```

Of course, this has become complicated, even kludgy. Instead, you can pass a data-returning callable rather than a container. That's the approach taken in the next topic.

---

## 1.8.6 Adding custom fields and data to messages

This example demonstrates adding custom fields and data to logged messages. It uses a custom formatter with two new keywords, `user` and `ip`, and a class filter created with a callable data source – static initializing data for the filter, but a source of dynamic data. The filter's `filter` method adds attributes of the same names as the keywords to each `LogRecord` passed to it, calling the data source to obtain current values for these attributes.

Here's the class filter and the data source:

```
import logging
from prelogging import LCDict
from random import choice

USER = 0
IP = 1

class FilterThatAddsFields():
    def __init__(self, *args, **kwargs):
        self.datasource = kwargs.get('datasource', None)    # callable

    def filter(self, record):
        """
        Add attributes to `record`.
        Their names must be the same as the keywords in format string (below).
        """
        record.user = self.datasource(USER)
        record.ip = self.datasource(IP)
        return True

def get_data(keyword):
    """ Source of dynamic data, passed to filter via `add_class_filter`. """
    IPS = ['192.0.0.1', '254.15.16.17']
    USERS = ['John', 'Mary', 'Arachnid']

    if keyword == IP:
        return choice(IPS)
    elif keyword == USER:
        return choice(USERS)
    return None
```

Configure logging:

```
lcd = LCDict(attach_handlers_to_root=True,
             root_level='DEBUG')
lcd.add_formatter('user_ip_level_msg',
                  format='User: %(user)-10s  IP: %(ip)-15s  %(levelname)-8s
↪%(message)s')
lcd.add_stdout_handler('console-out',
                       level='DEBUG',
                       formatter='user_ip_level_msg')
lcd.add_class_filter('field-adding_filter', FilterThatAddsFields,
                     # extra, static data
                     datasource=get_data)
lcd.attach_root_filters('field-adding_filter')

lcd.config()
```

Finally, log some messages, using the root logger:

```
LEVELS = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.
↪CRITICAL)
for i in range(10):
    logging.log(choice(LEVELS), "Msg %d", i)
```

The loop prints something like this:

```
User: Arachnid    IP: 254.15.16.17     CRITICAL  Msg 0
User: John        IP: 192.0.0.1        INFO      Msg 1
User: Mary        IP: 192.0.0.1        DEBUG     Msg 2
User: John        IP: 192.0.0.1        CRITICAL  Msg 3
User: Mary        IP: 254.15.16.17     WARNING   Msg 4
User: John        IP: 254.15.16.17     CRITICAL  Msg 5
User: John        IP: 254.15.16.17     DEBUG     Msg 6
User: John        IP: 254.15.16.17     CRITICAL  Msg 7
User: Arachnid    IP: 192.0.0.1        DEBUG     Msg 8
User: Mary        IP: 254.15.16.17     ERROR     Msg 9
```

This example loosely adapts the code of the section Using Filters to impart contextual information in *The Logging Cookbook*.

---

### 1.8.7 Adding SMTPHandlers with `add_email_handler`

Sending an email can take a comparatively long time, so you'll want to do that "in the background", so that other processes, or the UI, aren't impeded by sending emails. Use the queue handler/queue listener approach (see the example `queue_handler_listener.py`) to send emails from a thread other than the main one (and other than the UI thread).

Two of the examples illustrate relevant techniques.

`examples/SMTP_handler_just_one.py` uses `add_email_handler` to add an `SMTPHandler` with loglevel `ERROR`. The emails sent will have the same Subject and recipients for both `ERROR` and `CRITICAL` logged messages.

`examples/SMTP_handler_two.py` uses `add_email_handler` to add two `SMTPHandlers` – one with loglevel `ERROR`, the other with loglevel `CRITICAL`. The handler with loglevel `ERROR` has a filter to screen out logged messages of loglevel `CRITICAL`. In this way, emails sent for `ERROR` and `CRITICAL` logged messages can have different Subjects and recipients, specific to the triggering loglevel.

## 1.9 Guide to Examples

The *prelogging* distribution contains a number of examples, in the top-level `examples/` directory. You must download the repository in order to run them: they are not installed.

### 1.9.1 Running the examples

Run the programs in the `examples/` directory from that directory. On *nix, all the top-level modules of the examples are marked executable and contain shebangs, so, for example, the following works:

```
$ cd path/to/examples
$ ./mproc.py
```

Of course, `python ./mproc.py` works too. On Windows, use:

```
$ cd path\to\examples
$ python ./mproc.py
```

### 1.9.2 Programs in the `examples/` directory

This section catalogs the example programs by category and briefly describes each one, sometimes in the imperative *a la* docstrings.

#### Simple examples

##### `root_logger.py`

Create a root logger with a stdout console handler at loglevel `INFO`, and a file handler at default loglevel `NOTSET`. Root loglevel is `INFO`.

**Logfile** `examples/_log/root_logger/logfile.log`

##### `child_logger_main.py`

Uses modules `child_logger_sub_prop.py` and `child_logger_sub_noprop.py`.

Create a non-root logger with two child loggers, one propagating and one not. The parent logger has a stderr handler and a file handler, shared by the propagating logger. The non-propagating logger creates its own stderr handler by cloning its parent's stderr handler; however, it uses the same file handler as its parent (and its sibling).

Observe how the loglevels of the handlers and loggers determine what gets written to the two destinations.

**Logfile** `examples/_log/child_loggers/child_loggers.log`

##### `child_logger2_main.py`

Uses `child_logger2_sub_prop.py`, `child_logger2_sub_noprop.py`.

Give the root logger a stderr handler and a file handler. Create two loggers, one propagating and the other not. The non-propagating logger creates its own stderr handler by cloning the root's stderr handler; however, it uses the same file handler used by the root (and its sibling).

Observe how the loglevels of the handlers and loggers determine what gets written to the two destinations.

**Logfile** `examples/_log/child_loggers2/child_loggers2.log`

##### `dateformat.py`

A small example showing two uses of the `dateformat` parameter of `add_formatter`.

#### dictConfig-can-kill-existing-root-configuration.py

This example helps make the case for `LCDictBuilderABC`.

A call to `logging.config.dictConfig(d)` kills existing handlers on any logger that's configured in `d` – even with `'disable_existing_loggers': False`. The root logger *always* get configured if `d['root']` is nonempty. Thus, multiple calls to `logging.config.dictConfig(d)` can leave the root with only the handlers specified for it in the last logging config dict passed, or with no handlers at all.

The same is of course true of `LCDict.config()`.

This program demonstrates the phenomenon, using either *prelogging* or pure *logging* APIs depending on the value of the constant `USE_PRELOGGING`. When the `PRESERVE_ROOT` constant is True, the `'root'` subdict is set to `{}`, preserving the root's configuration, including its handlers.

Thus, it's chancy to do "collaborative configuration" by having separate "areas" of a program build their own `LCDicts` and each call `config()` on them. Not only does that approach sacrifice *prelogging*'s consistency checking, but it also opens the door to hard-to-diagnose logging bugs.

### Handler examples

#### `use_library.py` (use of a null handler)

A main program which configures logging, *and* uses a package (`library`) which *also* configures logging in its `__init__` module. The package sets up logging with a non-root logger at loglevel `INFO` which uses a null handler; package methods log messages with that logger. The program adds a stdout handler to the root, with loglevel `DEBUG`; the root loglevel is the default, `WARNING`.

The package's logger propagates, therefore messages logged by the package with loglevel at least `INFO` are written.

#### `SMTP_handler_just_one.py` and `SMTP_handler_two.py`

These programs use `add_email_handler` to add SMTP handlers. `SMTP_handler_just_one.py` adds a single SMTP handler; `SMTP_handler_two.py` adds two, one with a filter, in order to send different email messages for different loglevels.

> **Attention:** For these examples to work properly, you must edit `examples/_smtp_credentials.py` to contain a valid username, password and SMTP server.

#### `syslog.py`

(*OS X aka macOS only*) Set the root logger level to `DEBUG`, and add handlers:

- add a stdout handler with loglevel `WARNING`, and
- use `add_syslog_handler` to add a syslog handler with default loglevel `NOTSET`.

Also see the example `mproc_deco_syslog.py`, described *below*.

#### `queue_handler_listener.py`

An example that illustrates how to use a QueueListener with *prelogging* so that messages can be logged on a separate thread. In this way, handlers that block and take long to complete (e.g. SMTP handlers, which send emails) won't make other threads (e.g. the UI thread) stall and stutter.

For motivation, see Dealing with handlers that block in the *logging* Cookbook. We've adapted the code in that section to *prelogging*.

Another approach can be found in the example `mproc_approach__queue_handler_logging_thread.py`, described *below*.

### Filter examples

#### filter-class-extra-static-data.py

Passing extra static data to a class filter via keyword arguments to `add_class_filter` to specify how different instances will filter messages.

Described and walked through in the section on *providing extra, static data to a class filter* of the "Further Topics and Recipes" chapter.

#### filter-callable-extra-static-data.py

The analogous construction – passing extra static data to a callable filter via keyword arguments to `add_class_filter` to specify how it will filter messages.

Namedropped but not described in the section on *providing extra, static data to a callable filter* of "Further Topics and Recipes".

#### filter-adding-fields--custom-formatter-keywords-for-fields.py

This example illustrates adding custom fields and data to logged messages. It uses a custom formatter with two new keywords, `user` and `ip`, and a class filter created with a callable data source – static initializing data for the filter, but a source of dynamic data. The filter's `filter` method adds attributes of the same names as the keywords to each `LogRecord` passed to it, calling the data source to obtain current values for these attributes.

Loosely adapts the section Using Filters to impart contextual information of The Logging Cookbook.

### Custom formatter examples

#### custom_class_formatter.py

How to configure and use a subclass of `logging.Formatter` as a formatter.

#### custom_callable_formatter.py

How to configure and use a callable as a formatter.

### Multiprocessing examples

Except for the `mproc_approach_*.py` examples, the programs described in this section all take command line arguments which tell them whether or not to use locking handlers.

Usage for the programs that take command line parameters:

```
./program_name [--LOCKING | --NOLOCKING]
./program_name -h | --help

 Options (case-insensitive, initial letter suffices,
          e.g. "--L" or "--n" or even -L):
```

(continues on next page)

(continued from previous page)

```
-L, --LOCKING      Use locking handlers       [default: True]
-N, --NOLOCKING    Use non-locking handlers   [default: False]
-h, --help         Write this help message and exit.
```

When run without locking, the multiprocessing examples *will* eventually misbehave – `NUL` (0) bytes will appear in the logged output, and messages logged by different processes will barge in on each other. The directory `examples/_log saved` contains subdirectories `_log--2.7-runs`, `_log--3.x-runs (I)` and `_log--3.x-runs (II)` which capture several instances of this misbehavior. Though your mileage may vary, experience has shown that this expected misbehavior is more likely when these examples are run individually than when they're run via `run_examples.py` or `run_all.py`.

After running any of these examples, you can use `check_for_NUL.py` to check whether or not its logfile output is garbled:

`$ ./check_for_NUL.py` *filename*

reports which if any lines of a text file *filename* contain `NUL` bytes. Here, *filename* would be the name of the logfile that the program wrote to.

### mproc.py

A basic multiprocessing example that uses a non-propagating logger with a stdout handler and a file handler. The handlers are locking by default, non-locking if the `-N` command line flag is given.

**Logfiles** `examples/_log/mproc/mproc_LOCKING.log` `examples/_log/mproc/mproc_NONLOCKING.log`

### mproc2.py

Another basic multiprocessing example that adds a stdout handler and a file handler to the root logger. The handlers are locking by default, optionally non-locking as explained above.

**Logfiles** `examples/_log/mproc2/mproc2_LOCKING.log` `examples/_log/mproc2/mproc2_NONLOCKING.log`

### mproc_deco.py

Just like `mproc2.py` but using the *deco* package to set up multiprocessing.

**Logfiles** `examples/_log/mproc_deco/logfile (LOCKING).log examples/_log/mproc_deco/logfile (NONLOCKING).log`

### mproc_deco_rot_fh.py

Adds a stdout handler and a rotating file handler to the root logger. The handlers are locking by default, optionally non-locking as explained above. This example uses *deco* to set up multiprocessing,

**Logfiles** `examples/_log/mproc_deco_rot_fh/LOCKING/rot_fh.log examples/_log/mproc_deco_rot_fh/LOCKING/rot_fh.log.1 . .. examples/_log/mproc_deco_rot_fh/LOCKING/rot_fh.log.10 examples/_log/mproc_deco_rot_fh/NONLOCKING/rot_fh.log examples/_log/mproc_deco_rot_fh/NONLOCKING/rot_fh.log.1 ... examples/_log/mproc_deco_rot_fh/NONLOCKING/rot_fh.log.10`

**mproc_deco_syslog.py**

Adds a stdout handler and a syslog handler to the root logger. The handlers are locking by default, optionally non-locking as explained above. This example uses *deco* to set up multiprocessing,

**mproc_approach__locking_handlers.py and mproc_approach__queue_handler_logging_thread.py**

These two programs illustrate two approaches to logging in the presence of multiprocessing: one uses *prelogging*'s native locking handlers; the other uses a queue handler and a logging thread.

See *Using QueueHandlers (Python 3 only)* in the chapter *Further Topics and Recipes* for (much) more about the latter.

**locking handler logfiles** `examples/_log/mproc_LH/mplog.log`, `examples/_log/mproc_LH/mplog-errors.log`, `examples/_log/mproc_LH/mplog-foo.log`

**queue handler logfiles** `examples/_log/mproc_QHLT/mplog.log`, `examples/_log/mproc_QHLT/mplog-errors.log`, `examples/_log/mproc_QHLT/mplog-foo.log`

## 1.10 Class Reference

*prelogging* isn't a large package: it's a few, mostly small classes in four modules.

### 1.10.1 LCDictBasic

This class resides in `lcdictbasic.py`.

### 1.10.2 LCDict

This class resides in `lcdict.py`.

### 1.10.3 Locking Handlers

The multiprocessing-safe handler classes `LockingStreamHandler`, `LockingFileHandler`, `LockingRotatingFileHandler` and `LockingSyslogHandler` all use the mixin class `MPLock_Mixin` to wrap a lock around calls to `emit`. All these classes reside in `locking_handlers.py`.

The *LCDict* class provides an interface to the locking handlers; in the ordinary course of things it's probably unnecessary to use them directly.

### 1.10.4 LCDictBuilderABC

This class resides in `lcdict_builder_abc.py`.

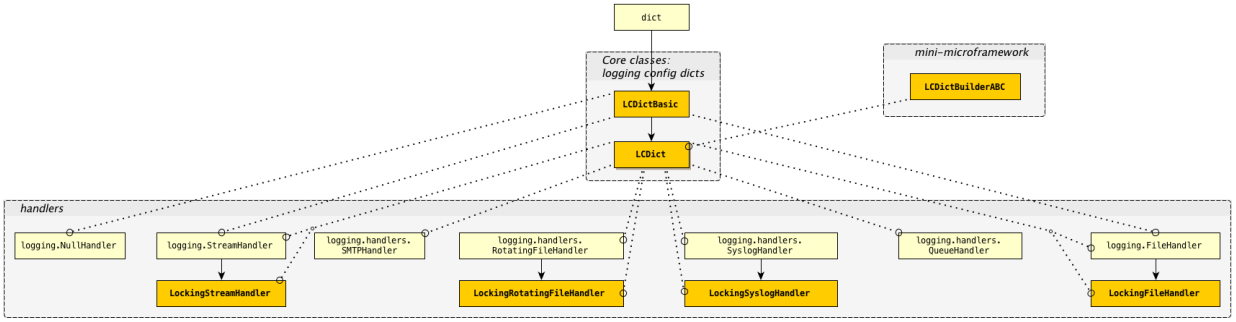
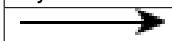



Fig. 3: **prelogging** classes — inheritance, and who uses whom

| Symbol | Meaning |
|---|---|
|  | is a superclass of |
|  | uses (instantiates) |

### 1.10.5 Class diagram

## 1.11 Index

# Index