log_calls Documentation

Release 0.3.2

Brian O'Neill

Dec 26, 2020

CONTENTS

CHAPTER

ONE

TABLE OF CONTENTS

1.1 *log_calls* — A Decorator for Debugging and Profiling

log_calls is a Python 3.3+ decorator that can print a lot of useful information about calls to decorated functions, methods and properties. The decorator can write to `stdout`, to another stream or file, or to a logger. *log_calls* provides methods for printing your own debug messages to its output stream, and for easily "dumping" variables and expressions paired with their values. It can decorate individual functions, methods and properties; but it can also programmatically decorate callable members of entire classes and class hierarchies, even of entire modules, with just a single line — which can greatly expedite learning a new codebase.

In short, *log_calls* can save you from writing, rewriting, copying, pasting and tweaking a lot of *ad hoc*, debug-only, boilerplate code — and it can keep *your* codebase free of that clutter.

For each call to a decorated function or method, *log_calls* can show you:

- the caller (in fact, the complete call chain back to another *log_calls*-decorated caller, so there are no gaps in chains displayed)
- the arguments passed to the function or method, and any default values used
- nesting of calls, using indentation
- the number of the call (whether it's the 1st call, the 2nd, the 103rd, . . .)
- the return value
- the time it took to execute
- and more!

These and other features are optional and configurable settings, which can be specified for each decorated callable via keyword parameters, as well as *en masse* for a group of callables all sharing the same settings. You can examine and change these settings on the fly using attributes with the same names as the keywords, or using a dict-like interface whose keys are the keywords.

log_calls can also collect profiling data and statistics, accessible at runtime, such as:

- the number of calls to a function
- total time taken by the function
- the function's entire call history (arguments, time elapsed, return values, callers, and more), available as text in CSV format and, if Pandas is installed, as a DataFrame.

The package contains two other decorators:

- *record_history*, a stripped-down version of *log_calls*, only collects call history and statistics, and outputs no messages;

- *used_unused_keywords* lets a function or method easily determine, per-call, which of its keyword parameters were actually supplied by the caller, and which received their default values.

This document describes the decorators' features and their use. Th e``tests/`` subdirectory of the distribution archive contains many test suites. These contain many additional examples, with commentary. As tests, they provide 96+% coverage.

1.2 What's New (releases 0.3.2, 0.3.1 and 0.3.0)

This release, 0.3.2, updates *log_calls* for Python 3.6. There are *no* changes to package code, only minor changes to documentation and to a single test. However, the package itself is reorganized: the `docs/` and `tests/` subdirectories have been moved to the top level of the distribution archive, at the same level as `log_calls/` rather than within the package. They are no longer installed by `setup.py` or `pip`, so you'll need the distribution to access their files.

This chapter catalogs additions and changes in the current release and its predecessor. Those in 0.3.2/0.3.1 are few; those in 0.3.0, several. *Appendix II* contains the complete list of what has been new in earlier versions.

1.2.1 Version 0.3.2 / 0.3.1

What's New

- This version simplifies writing debugging messages and dumping expressions to the *log_calls* output stream in a *log_calls*-aware way. There are now two methods for these purposes:
  - `log_calls.print()`
  - `log_calls.print_exprs()`

  which can be called from within any decorated callable. These supercede the now deprecated `log_message` and `log_exprs` attributes on each decorated callable. The new methods are notably easier to use from within classes.

  Like their predecessors, these methods are indent-aware, and, unlike the global `print` function, they produce no output unless called from within an enabled decorated function.

  You can now simply call `log_calls.print('Starting timer.')` or `log_calls.print_exprs('x', 'y', '(x+y)/2'`, without having to first obtain a reference to a "wrapper" and then calling the `log_*` methods on *that*.

  Version 0.3.0 provided one-stop shopping for obtaining wrappers; in earlier versions of *log_calls* you had to navigate to it yourself, with different expressions for instance methods, classmethods, staticmethods and properties.

  By default, if you call `log_calls.print*` from within a method or function that isn't decorated, it does nothing. You can comment out the `@log_calls` decorator, or use the `NO_DECO` parameter to achieve the same end, and the `.print*` method calls will play nicely: they won't output anything, **and** the calls won't raise `AttributeError` as they would formerly when calling the methods on a wrapper that `is None`. In short, leaving the `log_calls.print*` lines uncommented is as benign as it can be.

  But probably at some point you *do* want to know when you have lingering code that's supposedly development-only. *log_calls* will inform you of that if you set the following new global flag to `True` (or to something truthy):

- `log_calls.print_methods_raise_if_no_deco` (`bool`; default: `False`)

When this flag is true, calls to `log_calls.print` and `log_calls.print_exprs` from within an undecorated function or method will raise an appropriate exception. This compels you to comment out or delete any calls to `log_calls.print*` from within undecorated functions or methods.

The chapter *Writing log_calls-Aware Debug Messages* documents the new methods and global flag.

- The new methods and global exist on `record_history` too:
  - `record_history.print()`
  - `record_history.print_exprs()`
  - `record_history.print_methods_raise_if_no_deco`

  all exist and behave analogously to the *log_calls* attributes.

What's Changed

- A callable's display name now has its `__name__` in parentheses following its `__qualname__` if (and only if)
  - the `name` parameter was not provided to *log_calls* for the callable, and
  - the callable's `__name__` is not a substring of its `__qualname__`.

  See the section section *Decorating "external" code* in the Quick Start chapter for a motivating example.
- `log_exprs` now a `suffix` keyword parameter (as does `log_calls.print_exprs()`).
- Fixed: `log_calls.decorate_module()` wouldn't decorate *only* classes, or *only* functions; it would decorate nothing instead.

Deprecations

- *wrapper*`.log_message()` and *wrapper*`.log_exprs()`.

  Use `log_calls.print()` and `log_calls.print_exprs()` instead.

1.2.2 Version 0.3.0

What Was New in 0.3.0

- *log_calls* and *record_history* can decorate classes – all, or some, of the methods and properties within a class – and their inner classes.
  - The decorators properly decorate instance methods, classmethods, staticmethods and properties (whether defined with the `@property` decorator or the `property` function).
  - Settings provided in the class-level decorator apply to all decorated members and inner classes. Members and inner classes can also be individually decorated, and (by default) their explicitly given settings supplement and override those given at outer levels.
  - `omit` and `only` keyword parameters to a class decorator let you concisely specify which callables to decorate. Each is a sequence of strings specifying methods and/or properties — by name, with optional class prefixes, with optional suffixes for selecting specific property methods, as well as with wildcards and character-range inclusion and exclusion using "glob" syntax.

  - A decorated class has methods `get_log_calls_wrapper(methodname)` and `get_own_log_calls_wrapper()`, the latter for use by methods and properties of the decorated class. These provide easy and uniform ways to obtain the wrapper of a decorated method, without the special-case handling otherwise (and formerly) required for classmethods and properties.

    record_history provides the analogous methods `get_record_history_wrapper(methodname)` and `get_own_record_history_wrapper()`.

These capabilities are documented in *Decorating Classes*.

- *log_calls* and *record_history* have classmethods to programmatically decorate functions, classes and class hierarchies, even modules, for situations where altering source code is impractical (too many things to decorate) or inadvisable (third-party packages and modules). These methods can expedite learning a new codebase:
  - `decorate_class(baseclass, decorate_subclasses=False, **setting_kwds)` decorates a class and optionally all of its subclasses
  - `decorate_hierarchy(baseclass, **setting_kwds)` decorates a class and all of its subclasses
  - `decorate_function(f, **setting_kwds)` decorates a function defined in or imported into the module from which you call this method
  - `decorate_package_function(f, **setting_kwds)` decorates a function in an imported package
  - `decorate_module_function(f, **setting_kwds)` decorates a function in an imported package or module
  - `decorate_module(mod: 'module', functions=True, classes=True, **setting_kwds)` decorates all functions and classes in a module.

These are documented in *Bulk (Re)Decoration, (Re)Decorating Imports*.

- *log_calls* has classmethods to globally set and reset default values for settings, program-wide:
  - `set_defaults(new_default_settings=None, **more_defaults)`
  - `reset_defaults()`

  as well as classmethods to retrieve the current defaults and the "factory defaults", each as an `OrderedDict`:
  - `get_defaults_OD()`
  - `get_factory_defaults_OD()`

These are documented in *Retrieving and Changing the Defaults*.

- The `log_exprs()` method, added as an attribute to decorated callables, allows a wrapped callable to easily "dump" values of variables and expressions. Simply pass it one or more expressions, as strings; it prints the expressions together with their current values. See *Writing expressions and their values with log_calls.print_exprs()*.

- New keyword parameters:
  - `NO_DECO`, a "kill switch". When true, the decorator does nothing, returning the decorated callable or class itself, unwrapped and unaltered. Using this parameter in a settings file or dictionary lets you toggle "true bypass" with a single switch, e.g. for production, without having to comment out every decoration.
  - `name`, a literal string or a format string, lets you specify a custom name for a decorated callable.
  - `override`, a boolean, intended mainly for use with *log_calls* as a functional and with the `decorate_*` methods, allows updating the explicit settings of already decorated classes and callables.
  - `mute`, a three-valued setting:
    - mute nothing (default)
    - mute output about calls but allow `log_message()` and `log_exprs()` output
    - mute everything.
- Global mute, `log_calls.mute`, which can assume the same values as the new `mute` setting.
- Classmethods `log_calls.version()` and `record_history.version()` return the version string.

What Changed in 0.3.0

- The `indent` setting is now by default `True`.
- By default, the display name for a function or method is now its `__qualname__`, which in the case of methods includes class name. This makes unnecessary what was probably the main use case of `prefix`.
- *record_history* can now use `log_message()` and `log_exprs()`. Output is always via `print`.
- Fixed: `log_message()` formerly would blow up if called on a function or method for which logging was disabled. It now produces no output in that situation.
- `prefix` is mutable in *log_calls* and *record_history*.
- Fixed, addressed: double-decoration no longer raises an exception. Doing so doesn't wrap another decorator around an already wrapped function or method, but merely adjusts the settings of the decorated callable.
- Change to `__repr__` handling in the `arguments` section of output: use `object.__repr__` for objects still in construction (i.e. whose `__init__` methods are still active), otherwise use `repr`.
- *log_calls* won't itself decorate `__repr__` methods (it will decorate them instead with `reprlib.recursive_repr()`); *record_history* can decorate `__repr__`.
- Removed the deprecated `settings_path` keyword parameter.
- Officially, explicitly requires Python 3.3+. The package won't install on earlier versions.
- For consistency with the `get*_defaults_OD()` methods, the `as_OrderedDict()` method of the "settings" objects (e.g. `log_calls_settings`) has been renamed `as_OD()`. Note, `as_OrderedDict()` is still supported but is now deprecated. You'll have to run the Python interpreter with the `-Wd` flag to see the deprecation warning(s), which include the file names and line numbers where `as_OrderedDict()` occurs. (Since Python 3.2, DeprecationWarnings are by default not displayed.)

1.3 Installation

1.3.1 Dependencies and requirements

The *log_calls* package has no dependencies — it requires no other packages. All it requires is a standard distribution of Python 3.3 or higher (Python 3.4+ recommended).

1.3.2 Installing *log_calls*

You have two simple options:

1. Run:

   ```
   $ pip install log_calls
   ```

   to install *log_calls* from PyPI (the Python Package Index), or

   > *Here and elsewhere,* `$` *at the* beginning *of a line indicates your command prompt, whatever it may be.*

2. download the compressed distribution file (a `.tar.gz` or a `.zip`), uncompress it into a directory, and run:

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

   in that directory.

The complete distribution of *log_calls* (available as a `tar.gz` or a `zip` from PyPI or github) contains three subdirectories: `log_calls`, the package proper; `docs`, the documentation source files; and `tests`, mentioned above. These last two subdirectories are *not* installed by pip, so to obtain those files you'll have to download an archive, and then, you may as well install *log_calls* using method 2.

Whichever you choose, ideally you'll do it in a virtual environment (a *virtualenv*). In Python 3.3+, it's easy to set up a virtual environment using the pyvenv tool included in the standard distribution.

1.3.3 Running the tests

Each `*.py` file in the `log_calls/` directory has at least one corresponding test file `test_*.py` in the `log_calls/tests/` directory. The tests provide 96+% coverage. All tests have passed on every tested platform + Python version (3.3.x through 3.6.0); however, that's a sparse matrix :) If you encounter any turbulence, do let us know.

You can run the tests for *log_calls* after downloading it but before installing it, by running the following command in the directory into which you uncompressed the download:

```
$ ./run_tests.py [-q | -v | -h]
```

which takes switches `-q` for "quiet" (the default), `-v` for "verbose", and `-h` for "help".

What to expect

Both of the above commands run all tests in the `tests/` subdirectory. If you run either of them, the output you see should end like so:

```
Ran 112 tests in 2.235s
OK
```

indicating that all went well. (**Depending upon which Python version you're using and on what packages you have installed, you may see fewer tests reported.**) If any test fails, it will tell you.

Note: This package *probably* requires the CPython implementation, as it uses internals of stack frames which may well differ in other interpreters. It's not guaranteed to fail without CPython, it's just untested. (*If you're able and willing to run the tests under another interpreter or compiler, please tell us what you find.*)

PyPy is not (yet?) compatible with `log_calls`. Finally PyPy3 supports Python 3.3: as of Winter 2017, the PyPy3 project has reached Python 3.3.5 with their version PyPy3.3, with 3.5 support in development (PyPy3.5, by name). So finally we can test its compatibility with *log_calls*: 8 of 110 tests fail. Some of the failures appear to be because PyPy3.3 has incorrect values for ``__qualname__``s of inner classes, and we'd expect these tests to pass in PyPy3.5. Others may be more fundamental — watch this space for at least a fuller assessment, if not a change in compatibility status.

1.4 Quick Start

1.4.1 Basic usage

First, let's import the *log_calls* decorator from the package of the same name:

```
>>> from log_calls import log_calls
```

In code, `log_calls` now refers to the decorator, a class (an object of type `type`), and not to the module:

```
>>> type(log_calls)
type
```

The decorator has many options, and thus can take many parameters, but let's first see the simplest examples possible, using no parameters at all.

Decorating functions

If you decorate a function with *log_calls*, each call to the function is generally preceded and followed by some reportage. The decorator first writes messages announcing entry to the function and what arguments it has received; the decorator calls the function with those arguments, and the function executes; upon its return, the decorator finishes up and announces the return of the function:

```
>>> @log_calls()
... def f(a, b, c):
...     print("--- Hi from f")
>>> f(1, 2, 3)
f <== called by <module>
    arguments: a=1, b=2, c=3
```

(continues on next page)

(continued from previous page)

```
--- Hi from f
f ==> returning to <module>
```

Adding another decorated function to the call chain presents useful information too. Here, `g` calls the decorated `f` above. Observe that (by default) the *log_calls* output for the nested call to `f` is indented to align with the inner lines of the *log_calls* output for `g`:

```
>>> @log_calls()
... def g(n):
...     print("*** Hi from g")
...     f(n, 2*n, 3*n)
...     print("*** Bye from g")
>>> g(3)
g <== called by <module>
    arguments: n=3
*** Hi from g
    f <== called by g
        arguments: a=3, b=6, c=9
--- Hi from f
    f ==> returning to g
*** Bye from g
g ==> returning to <module>
```

log_calls gives informative output even when call chains include undecorated functions. In the next example, a decorated function `h` calls an undecorated `g2`, which calls an undecorated `g1`, which, finally, calls our original decorated `f`:

```
>>> def g1(n): f(n, 2*n, 3*n)
>>> def g2(n): g1(n)
>>> @log_calls()
... def h(x, y): g2(x+y)
```

Now let's call `h`:

```
>>> h(2, 3)
h <== called by <module>
    arguments: x=2, y=3
    f <== called by g1 <== g2 <== h
        arguments: a=5, b=10, c=15
--- Hi from f
    f ==> returning to g1 ==> g2 ==> h
h ==> returning to <module>
```

Notice that when writing entry and exit messages for `f`, *log_calls* displays the entire active call chain *back to the nearest decorated function*, so that there aren't "gaps" in the chain of functions it reports on. If it didn't do this, we'd see only `f <== called by g1`, and then `f ==> returning to g1` followed by `h ==> returning to <module>`, which wouldn't tell us the whole story about how control reached `g1` from `h`.

See the *Call Chains* chapter for more examples and finer points.

Decorating methods

Similarly, you can decorate methods (and properties) within a class:

```
>>> class A():
...     def __init__(self, n):
...         self.n = n
...
...     @log_calls()
...     def ntimes(self, m):
...         return self.n * m
```

Only the `ntimes` method is decorated:

```
>>> a = A(3)            # __init__ called
>>> a.ntimes(4)
A.ntimes <== called by <module>
    arguments: self=<__main__.A object at 0x...>, m=4
A.ntimes ==> returning to <module>
12
```

1.4.2 Decorating classes

To decorate all methods of a class, simply decorate the class itself:

```
>>> @log_calls()
... class C():
...     def __init__(self, n):
...         self.n = n if n >= 0 else -n
...
...     @staticmethod
...     def revint(x): return int(str(x)[::-1])
...
...     @property
...     def revn(self): return self.revint(self.n)
```

All methods of `C` are now decorated. Creating an instance logs the call to `__init__`:

```
>>> c = C(123)
C.__init__ <== called by <module>
    arguments: self=<__main__.C object at 0x...>, n=123
C.__init__ ==> returning to <module>
```

Accessing its `revn` property calls the staticmethod `revint`, and both calls are logged:

```
>>> c.revn
C.revn <== called by <module>
    arguments: self=<__main__.C object at 0x...>
    C.revint <== called by C.revn
        arguments: x=123
    C.revint ==> returning to C.revn
C.revn ==> returning to <module>
321
```

If you want to decorate only some of the methods of a class, you *don't* have to individually decorate all and only all the ones you want: the `only` and `omit` keyword parameters to the class decorator let you concisely specify which methods will and won't be decorated. The section on *the omit and only keyword parameters* contains the details.

Decorating *most* methods, overriding the settings of one method

Suppose you have a class `D` that's just like `C` above, but adds a `double()` method. (For the sake of example, never mind that in practice you might subclass `C`.) Suppose you want to decorate all callables in `D` *except* `revint`, and furthermore, you want *log_calls* to report values returned by the property getter `revn`. Here's how to do it:

```
>>> @log_calls(omit='revint')
... class D():
...     def __init__(self, n):
...         self.n = n if n >= 0 else -n
...
...     @staticmethod
...     def revint(x): return int(str(x)[::-1])
...
...     def double(self): return self.n + self.n
...
...     @property
>>>     @log_calls(log_retval=True)
...     def revn(self): return self.revint(self.n)
```

By default, *log_calls* does *not* display return values, and the outer, class-level decorator uses that default. The explicit decorator of `revn` overrides that, specifying the desired setting. Note that `@log_calls` follows `@property`: in general, when decorating a callable in a class, `@log_calls` should come *after* any `@property`, `@classmethod` or `@staticmethod` decorator.

Let's see this class in action:

```
>>> d = D(71)
D.__init__ <== called by <module>
    arguments: self=<__main__.D object at 0x...>, n=71
D.__init__ ==> returning to <module>
```

The return value of `d.double()` is *not* logged:

```
>>> d.double()
D.double <== called by <module>
    arguments: self=<__main__.D object at 0x...>
D.double ==> returning to <module>
```

However, the return value of `revn` *is* logged, and `revint` has *not* been decorated:

```
>>> print('~~~\\nMy favorite number plus 3 is', d.revn + 3)
D.revn <== called by <module>
    arguments: self=<__main__.D object at 0x...>
    D.revn return value: 17
D.revn ==> returning to <module>
~~~
My favorite number plus 3 is 20
```

A *doctest* quirk

The *doctest* examples in this document use `\\n` and `\\t` where in actual code you'd write `\n` and `\t` respectively. All the examples herein work (as tests, they pass), but they would fail if `\n` were used. It would also be possible to use "raw" strings and single escapes, as in `r'Nobody ever expects\nThe Spanish Inquisition!'`.

For more information

The *Decorating Classes* chapter covers that subject thoroughly — basics, details, subtleties and techniques. In particular, the parameters `only` and `omit` are documented there, in the section the omit and only keyword parameters.

1.4.3 Writing *log_calls*-aware debugging messages

Printing statements to an output device or file is one of the oldest forms of debugging. These statements track a program's progress, display the values of variables, announce milestones, report on the consistency of internal state, and so on. Let's call such statements *debugging messages*.

The `@log_calls` decorator automates the boilerplate aspects of this reportage: who calls whom, when, how, and with what result. *log_calls* also provides the methods

- `log_calls.print()` and
- `log_calls.print_exprs()`

as attractive alternatives to the `print` function for writing other debugging messages.

One common kind of debugging message reports the values of variables as a program runs, taking snapshots at strategic places at the top level of the code, or within a loop as an algorithm executes. Writing such statements becomes tedious quickly — they're all alike though in details all different too. The `log_calls.print_exprs` method lets you easily display the values of variables and expressions within a decorated function.

All other debugging messages require a method as general as `print`: the `log_calls.print` method is that counterpart.

Both methods write to the same output destination as the decorator, whether that's the console, a file or a logger, and their output is properly synced and aligned with the decorator's output:

```
>>> @log_calls()
... def gcd(a, b):
...     log_calls.print("At bottom of loop:")
...     while b:
...         a, b = b, (a % b)
...         log_calls.print_exprs('a', 'b', prefix="\\t", suffix= '\\t<--')
...     return a
>>> gcd(48, 246)
gcd <== called by <module>
    arguments: a=48, b=246
    At bottom of loop:
            a = 246, b = 48 <--
            a = 48, b = 6   <--
            a = 6, b = 0    <--
gcd ==> returning to <module>
6
```

If you delete, comment out or otherwise disable the decorator, the `print*` methods will do nothing (except waste a little time). To illustrate this, we could just repeat the above function with the decorator omitted or commented out; but we can also disable the decorator dynamically, and the `print*` methods will be silent too:

```
>>> gcd.log_calls_settings.enabled = False
>>> gcd(48, 246)
6
```

You can pass expressions to `print_exprs`:

```
>>> @log_calls()
... def f():
...     x = 42
...     log_calls.print_exprs('x', 'x//6', 'x/6')
>>> f()
f <== called by <module>
    x = 42, x//6 = 7, x/6 = 7.0
f ==> returning to <module>
```

`print` and `print_exprs` properly indent even multiline messages:

```
>>> @log_calls()
... def f(a):
...     log_calls.print("Even multiline messages\\n"
...                     "are properly indented.")
...     return g(a, 2*a)
>>> @log_calls()
... def g(x, y):
...     retval = x + y + 1
...     log_calls.print_exprs('retval',
...                           prefix="So are multiline\\n"
...                                  "prefixes --\\n",
...                           suffix="\\n-- and suffixes.")
...     return retval
>>> f(2)
f <== called by <module>
    arguments: a=2
    Even multiline messages
    are properly indented.
    g <== called by f
        arguments: x=2, y=4
        So are multiline
        prefixes --
        retval = 7
        -- and suffixes.
    g ==> returning to f
f ==> returning to <module>
7
```

You can specify multiple lines for `print` either with one string that has explicit newlines, as above, or by using the `sep` keyword parameter together with multiple positional string arguments:

```
>>> @log_calls()
... def h():
...     log_calls.print("Line 1 of 3", "line 2 of 3", "line 3 of 3",
...                     sep='\\n')
>>> h()
h <== called by <module>
    Line 1 of 3
    line 2 of 3
    line 3 of 3
h ==> returning to <module>
```

The behavior of the `print*` methods is configurable in a few ways:

- their output can be "allowed through" while muting the output of the decorators;
- their output doesn't *have* to be indented, it can be flush left (`extra_indent_level=-1000`);
- optionally the methods can raise an exception if called from within a function or method that isn't decorated, so that development-only code doesn't sneak into production.

See the chapter *Writing log_calls-Aware Debugging Messages* for details about the `print()` and `print_exprs()` methods. The chapter *Dynamic Control of Settings* documents the `log_calls_settings` attribute of a decorated callable.

1.4.4 Decorating "external" code

Sometimes it's enlightening and instructive to decorate objects in a package or module that you import. It might be in a new codebase you're getting to know, your own nontrivial code from a while ago which you now wish you had documented more, or even a function, class or module in Python's standard library.

We'll illustrate techniques with a simple example: decorating the fractions class `fractions.Fraction` in the standard library, to examine how it works. Along the way we'll illustrate using *log_calls* settings to filter the output, forming hunches about how `Fraction` works based on the information the decorator presents, and consulting the source code to confirm or refute those hunches.

First, let's import the class, decorate it and create an instance:

```
>>> from fractions import Fraction as Frac
>>> log_calls.decorate_class(Frac)
>>> print(Frac(3,4))
Fraction.__new__ <== called by <module>
    arguments: cls=<class 'fractions.Fraction'>, numerator=3, denominator=4
    defaults:  _normalize=True
Fraction.__new__ ==> returning to <module>
Fraction.__str__ <== called by <module>
    arguments: self=Fraction(3, 4)
Fraction.__str__ ==> returning to <module>
3/4
```

(**Note**: *In this section, the expected output shown is from Python 3.6 and 3.5. The output of Python 3.4 differs slightly: in places it's less efficient, and* __new__*, indirectly called below, had no* _normalize *parameter.*)

Now create a couple of fractions, using the *log_calls* global mute to do it in silence:

```
>>> log_calls.mute = True
>>> fr56 = Frac(5,6)
>>> fr78 = Frac(7,8)
>>> log_calls.mute = False
```

Before using these, let's redecorate to improve *log_calls* output. After trying other examples at the command line it becomes apparent that `__str__` gets called a lot, and the calls become just noise, so let's `omit` that. To eliminate more clutter, let's suppress the exit lines ("... returning to..."). We'll also display return values. Here's how to accomplish all of that, with another call to `decorate_class`, which won't wrap the *log_calls* wrappers already created but will instead just update their settings:

```
>>> log_calls.decorate_class(Frac,
...                          omit='__str__', log_exit=False, log_retval=True)
```

Finally, let's do some arithmetic on fractions:

```
>>> print(fr78 - fr56)
Fraction._operator_fallbacks.<locals>.forward (__sub__) <== called by <module>
    arguments: a=Fraction(7, 8), b=Fraction(5, 6)
    Fraction.denominator <== called by _sub <== Fraction._operator_fallbacks.<locals>.
↪forward (__sub__)
        arguments: a=Fraction(7, 8)
        Fraction.denominator return value: 8
    Fraction.denominator <== called by _sub <== Fraction._operator_fallbacks.<locals>.
↪forward (__sub__)
        arguments: a=Fraction(5, 6)
        Fraction.denominator return value: 6
    Fraction.numerator <== called by _sub <== Fraction._operator_fallbacks.<locals>.
↪forward (__sub__)
        arguments: a=Fraction(7, 8)
        Fraction.numerator return value: 7
    Fraction.numerator <== called by _sub <== Fraction._operator_fallbacks.<locals>.
↪forward (__sub__)
        arguments: a=Fraction(5, 6)
        Fraction.numerator return value: 5
    Fraction.__new__ <== called by _sub <== Fraction._operator_fallbacks.<locals>.
↪forward (__sub__)
        arguments: cls=<class 'fractions.Fraction'>, numerator=2, denominator=48
        defaults:  _normalize=True
        Fraction.__new__ return value: 1/24
    Fraction._operator_fallbacks.<locals>.forward (__sub__) return value: 1/24
1/24
```

The topmost call is to an inner function `forward` of the method `Fraction._operator_fallbacks`, presumably a closure. The `__name__` of the callable is actually `__sub__` (its `__qualname__` is `Fraction._operator_fallbacks.<locals>.forward`). We know that classes implement the infix subtraction operator `-` with "dunder" methods `__sub__` and `__rsub__`, so it appears that in `Fraction`, the closure *is* the value of the attribute `__sub__`:

```
>>> Frac.__sub__
<function Fraction._operator_fallbacks.<locals>.forward...>
>>> Frac.__sub__.__qualname__
'Fraction._operator_fallbacks.<locals>.forward'
>>> Frac.__sub__.__name__
'__sub__'
```

The closure calls an undecorated function or method `_sub`. Because `_sub` isn't decorated we don't know what its arguments are, and the call chains for the decorated `numerator`, `denominator` and `__new__` chase back to `__sub__`. It appears to know about both operands, so we might guess that it takes two arguments. A look at the source code for `fractions`, fractions.py confirms that guess (`_sub` is on line 433).

Why isn't `_sub` decorated?

Let's check that:

```
>>> print(Frac._sub(fr78, fr56))
Fraction._sub <== called by <module>
    arguments: a=Fraction(7, 8), b=Fraction(5, 6)
    Fraction.denominator <== called by Fraction._sub
        arguments: a=Fraction(7, 8)
        Fraction.denominator return value: 8
    Fraction.denominator <== called by Fraction._sub
        arguments: a=Fraction(5, 6)
        Fraction.denominator return value: 6
    Fraction.numerator <== called by Fraction._sub
        arguments: a=Fraction(7, 8)
        Fraction.numerator return value: 7
    Fraction.numerator <== called by Fraction._sub
        arguments: a=Fraction(5, 6)
        Fraction.numerator return value: 5
    Fraction.__new__ <== called by Fraction._sub
        arguments: cls=<class 'fractions.Fraction'>, numerator=2, denominator=48
        defaults:  _normalize=True
        Fraction.__new__ return value: 1/24
    Fraction._sub return value: 1/24
1/24
```

Aha: it *is* decorated after all, and the *log_calls* output certainly looks familiar.

Consulting the source code makes clear what's going on. When `Fraction` is created, on line 439 `__sub__` is set equal to a closure returned by `_operator_fallbacks(_sub, operator.sub)`, defined on line 318. The closure is an instance of its inner function `forward` on line 398, which implements generic dispatch based on argument types to one of the two functions passed to `_operator_fallbacks`. When called with two `Fractions`, `__sub__` calls `_sub` and not `operator.sub`. On line 407, `_operator_fallbacks` sets the name of the closure to `__sub__`.

So, the closure `forward` that implements `__sub__` has a nonlocal variable bound to the real `_sub` at class initialization, before the methods of the class were decorated. The closure calls the inner, decorated `_sub`, not the *log_calls* wrapper around it.

How the code works

Ultimately, then, subtraction of fractions is performed by a function `_sub`, to which `__sub__` i.e. `Fraction._operator_fallbacks.<locals>.forward` dispatches. `_sub` uses the public properties `denominator` and `numerator` to retrieve the fields of the `Fractions`, and returns a new `Fraction`, with a numerator of 2 (= 7 * 6 - 8 * 5) and denominator of 48 (= 6 * 8). `__new__` (line 124 of the source code) reduces the returned `Fraction` to lowest terms just before returning it (because its parameter `_normalize` is true, its default value, which gives Python 3.4 behavior).

Scrolling through `fractions.py` reveals that other operators are implemented in exactly the same way.

For more information

The `decorate_*` methods are presented in the chapter Bulk (Re)Decoration, (Re)Decorating Imports.

1.4.5 Where to go from here

These examples have shown just a few of the features that make *log_calls* powerful, versatile, yet easy to use. They introduced a few of *log_calls*'s keyword parameters, the source of much of its versatility, as well as one of the `decorate_*` methods.

The next chapter, *What log_calls Can Decorate*, gives general culture but also introduces terminology and concepts subsequently used throughout. An essential chapter follows: *Keyword Parameters* documents the parameters in detail. That chapter is a reference; it's not necessary to assimilate its details before proceeding on to further topics. For an even more concise reference, in cheatsheet format, see Appendix I: Keyword Parameters Reference.

log_calls provides a lot of functionality, which these examples have only introduced. The remaining chapters document all of it.

1.5 What *log_calls* Can Decorate

In this document, the phrase "decorated callable" appears frequently. Generally we use *callable* as a generic term that includes global functions as well as methods and properties of classes. We use it to emphasize that what is said applies equally to global functions, methods and properties, and indeed to anything that *log_calls* can decorate.

We use more the specific terms *decorated function*, *decorated method*, and so on, as appropriate for examples, and when what is said applies to the narrower class of callables named but perhaps not to all callables.

"functional"

A *functional* is a higher-order function, a function of functions.

- When passed a function *fn*, `log_calls(**kwds)(fn)` returns a function;
- when passed a class *klass*, `log_calls(**kwds)(klass)` returns the class *klass*.

Functions defined with `def`, methods and properties don't exhaust the callables that *log_calls* can decorate. Lambda expressions are functions, and *can* be decorated by using `log_calls()` as a *functional*, without the `@` syntactic sugar:

```
>>> f = log_calls()(lambda x: 2 * x)
>>> f(3)
<lambda> <== called by <module>
    arguments: x=3
<lambda> ==> returning to <module>
6
```

The question arises: what, exactly, *can log_calls* decorate? (and thus, what can't it decorate?) We won't attempt to give necessary and sufficient conditions for that set of callables. But the following is true:

Anything that *log_calls* can decorate is a callable,
but not every callable can be decorated by *log_calls*.

Whatever *log_calls* **cannot** decorate, it simply returns unchanged.

1.5.1 What is a "callable"?

Loosely, a "callable" is anything that can be called. In Python, the term has a precise meaning, encompassing not only functions and methods but also classes, as well as instances of classes that implement a `__call__` method. A correct though unsatisfying definition is: an object is *callable* iff the builtin `callable` function returns `True` on that object. The Python documentation for callable is good as far as it goes, but a bit breezy; greater detail can be found in the stackoverflow Q&A What is a "callable" in Python? and in the articles cited there.

1.5.2 A few negative examples

log_calls can't decorate callable builtins, such as `len` — it just returns the builtin unchanged:

```
>>> len is log_calls()(len)     # No "wrapper" around len -- not deco'd
True
>>> dict.update is log_calls()(dict.update)
True
```

Similarly, *log_calls* doesn't decorate builtin or extension type classes, returning the class unchanged:

```
>>> _ = log_calls()(dict)
>>> dict(x=1)                # dict.__init__ not decorated, no output
```

It also doesn't decorate various objects which are callables by virtue of having a `__call__` method, such as `functools.partial` objects:

```
>>> from functools import partial
>>> def h(x, y): return x + y
>>> h2 = partial(h, 2)         # so h2(3) == 5
>>> h2lc = log_calls()(h2)
>>> h2lc is h2                 # not deco'd
True
```

However, *log_calls can* decorate *classes* whose instances are callables by virtue of implementing a `__call__` method:

```
>>> @log_calls()
... class Rev():
...     def __call__(self, s):  return s[::-1]
>>> rev = Rev()
>>> callable(rev)
True
>>> rev('ABC')
Rev.__call__ <== called by <module>
    arguments: self=<Rev object at 0x...>, s='ABC'
Rev.__call__ ==> returning to <module>
'CBA'
```

1.6 Keyword Parameters

log_calls has many features, and thus many, mostly independent, keyword parameters (21 in release 0.3.2). This section covers most of them thoroughly, one at a time (though of course you can use multiple parameters in any call to the decorator):

- *enabled*
- *args_sep*
- *log_args*
- *log_retval*
- *log_exit*
- *log_call_numbers*
- *log_elapsed*
- *indent*
- *name*
- *prefix*
- *file*
- *mute* (also discusses the global mute switch `log_calls.mute`)
- *settings*
- *NO_DECO*
- *override*

The remaining parameters are fully documented in later chapters, For completeness, they're briefly introduced at the end of this chapter, together with links to their actual documentation.

- *omit, only*
- *logger, loglevel*
- *record_history, max_history*

1.6.1 What is a *setting*?

When *log_calls* decorates a callable (a function, method, property, ...), it "wraps" that callable in a function — the *wrapper* of the callable. Subsequently, calls to the decorated callable actually call the wrapper, which delegates to the original, in between its own pre- and post-processing. This is simply what decorators do.

log_calls gives the wrapper a few attributes pertaining to the wrapped callable, notably `log_calls_settings`, a dict-like object that contains the *log_calls* state of the callable. The keys of `log_calls_settings` are *log_calls* keyword parameters, such as `enabled` and `log_retval` — in fact, most of the keyword parameters, though not all of them.

The *settings of a decorated callable* are the key/value pairs of its `log_calls_settings` object, which is an attribute of the callable's wrapper. The settings comprise the *log_calls* state of the callable.

Initially the value of a setting is the value passed to the *log_calls* decorator for the corresponding keyword parameter, or the default value for that parameter if no argument was supplied for it. `log_calls_settings` can then be used to read *and write* settings values.

`log_calls_settings` is documented in *The log_calls_settings attribute — the settings API*.

Usage of "setting"

We also use the term "settings" to refer to the keys of `log_calls_settings`, as well as to its key/value pairs. For example,

> "the `indent` setting",

or

> "`enabled` is a setting, but `override` is not".

This overloading shouldn't cause any confusion.

The "settings"

The following keyword parameters are settings:

> `enabled` `args_sep` `log_args` `log_retval` `log_exit` `log_call_numbers` `log_elapsed` `indent` `prefix` `file` `mute` `logger` `loglevel` `record_history` `max_history`

As described in the chapter *Dynamic Control of Settings*, all of a decorated callable's settings can be accessed through `log_calls_settings`, and almost all can be changed on the fly.

The non-settings

The other keyword parameters are *not* settings:

> `NO_DECO` `settings` `name` `override` `omit` `only`

These are directives to the decorator telling it how to initialize itself. Their initial values are not subsequently available via attributes of the wrapper, and cannot subsequently be changed.

1.6.2 `enabled` (default: `True` (`== 1`))

Every example of *log_calls* that we've seen so far has produced output, as they have all used the default value `True` of the `enabled` parameter. Passing `enabled=False` to the decorator suppresses output:

```
>>> @log_calls(enabled=False)
... def f(a, b, c):
...     pass
>>> f(1, 2, 3)                  # no output
```

This is not totally pointless!, because, as with almost all *log_calls* settings, you can dynamically change the "enabled" state for a particular function or method. (Later chapters *Bulk (Re)Decoration, (Re)Decorating Imports* and *Dynamic Control of Settings* show ways to do so that could change this `enabled` setting.) The above decorates `f` and sets its *initial* "enabled" state to `False`.

Note: The `enabled` setting is in fact an `int`. This can be used advantageously.

See the examples *Using enabled as a level of verbosity* and *A metaclass example*, which illustrate using different positive values to specify increasing levels of verbosity in *log_calls*-related output.

Bypass

If you supply a negative integer as the value of `enabled`, that is interpreted as *bypass*: *log_calls* immediately calls the decorated callable and returns its value. When the value of `enabled` is false (`False` or `0`), the decorator performs a little more processing than that before it delegates to the decorated callable (it increments the number of the call, for example), though of course less than when `enabled` is positive (e.g. `True`).

1.6.3 `args_sep` (default: `',  '`)

The `args_sep` parameter specifies the string used to separate arguments. If the string ends in `\n` (in particular, if `sep` *is* `'\n'`), additional whitespace is interspersed so that arguments line up nicely:

```
>>> @log_calls(args_sep='\\n')
... def f(a, b, c, **kwargs):
...     print(a + b + c)
>>> f(1, 2, 3, u='you')
f <== called by <module>
    arguments:
        a=1
        b=2
        c=3
        **kwargs={'u': 'you'}
6
f ==> returning to <module>
```

1.6.4 `log_args` (default: `True`)

When true, as seen in all examples so far, arguments passed to the decorated callable are written together with their values. If the callable's signature contains positional and/or keyword "varargs", those are included if they're nonempty. (These are conventionally named `*args` and `**kwargs`, but *log_calls* will use the parameter names that actually appear in the callable's definition.) Any default values of keyword parameters with no corresponding argument are also logged, on a separate line:

```
>>> @log_calls()
... def f_a(a, *args, something='that thing', **kwargs): pass
>>> f_a(1, 2, 3, foo='bar')
f_a <== called by <module>
    arguments: a=1, *args=(2, 3), **kwargs={'foo': 'bar'}
    defaults:  something='that thing'
f_a ==> returning to <module>
```

Here, no argument information is logged at all:

```
>>> @log_calls(log_args=False)
... def f_b(a, *args, something='that thing', **kwargs): pass
>>> f_b(1, 2, 3, foo='bar')
f_b <== called by <module>
f_b ==> returning to <module>
```

If a callable has no parameters, *log_calls* won't display any "arguments" section:

```
>>> @log_calls()
... def f(): pass
>>> f()
f <== called by <module>
f ==> returning to <module>
```

If a callable has parameters but is passed no arguments, *log_calls* will display `arguments:  <none>`, plus any default values used:

```
>>> @log_calls()
... def ff(*args, **kwargs): pass
>>> ff()
ff <== called by <module>
    arguments: <none>
ff ==> returning to <module>
```

```
>>> @log_calls()
... def fff(*args, kw='doh', **kwargs): pass
>>> fff()
fff <== called by <module>
    arguments: <none>
    defaults:  kw='doh'
fff ==> returning to <module>
```

1.6.5 `log_retval` (default: `False`)

When this setting is true, values returned by a decorated callable are reported:

```
>>> @log_calls(log_retval=True)
... def f(a, b, c):
...     return a + b + c
>>> _ = f(1, 2, 3)
f <== called by <module>
    arguments: a=1, b=2, c=3
    f return value: 6
f ==> returning to <module>
```

Note: By default, *log_calls* suppresses the return value of `__init__` methods, even when `log_retval=True` has been passed to a decorator of the method's class. To override this, you'd have to decorate `__init__` itself and supply `log_retval=True`. However, there's no reason to: `__init__` returns `None`.

1.6.6 `log_exit` (default: `True`)

When false, this parameter suppresses the `... ==> returning to ...` line that indicates the callable's return to its caller:

```
>>> @log_calls(log_exit=False)
... def f(a, b, c):
...     return a + b + c
>>> _ = f(1, 2, 3)
f <== called by <module>
    arguments: a=1, b=2, c=3
```

1.6.7 `log_call_numbers` (default: `False`)

log_calls keeps a running tally of the number of times a decorated callable has been called. You can display this number using the `log_call_numbers` parameter:

```
>>> @log_calls(log_call_numbers=True)
... def f(): pass
>>> for i in range(2): f()
f [1] <== called by <module>
f [1] ==> returning to <module>
f [2] <== called by <module>
f [2] ==> returning to <module>
```

The call number is also displayed with the function name when `log_retval` is true:

```
>>> @log_calls(log_call_numbers=True, log_retval=True)
... def f():
...     return 81
>>> _ = f()
f [1] <== called by <module>
    f [1] return value: 81
f [1] ==> returning to <module>
```

The display of call numbers is particularly valuable in the presence of recursion or reentrance — see the example *Indentation and call numbers with recursion*, where the feature is used to good effect.

Clearing the number of calls (setting it to 0)

To reset the number of calls to a decorated function `f`, so that the next call number will be 1, call the `f.stats.clear_history()` method. To reset it for a callable in a class, call `wrapper.stats.clear_history()` where `wrapper` is the callable's wrapper, obtained via one of the two methods described in the section on *accessing the wrappers of methods*.

See *The stats.clear_history(max_history=0) method* in the *Call History and Statistics* chapter for details about the `clear_history()` method and, more generally, the `stats` attribute.

1.6.8 `log_elapsed` (default: `False`)

For performance profiling, you can measure the time a callable took to execute by using the `log_elapsed` parameter. When this setting is true, *log_calls* reports how long it took the decorated callable to complete, in seconds. Two measurements are reported:

- *elapsed time* (system-wide, including time elapsed during sleep), given by `time.perf_counter()`, and
- *process time* (system + CPU time, i.e. kernel + user time, sleep time excluded), given by `time.process_time()`.

```
>>> @log_calls(log_elapsed=True)
... def f(n):
...     for i in range(n):
...         # do something nontrivial...
...         pass
>>> f(5000)
f <== called by <module>
    arguments: n=5000
    elapsed time: ... [secs], process time: ... [secs]
f ==> returning to <module>
```

1.6.9 `indent` (default: `True`)

The `indent` parameter, when true, indents each new level of logged messages by 4 spaces, providing a visualization of the call hierarchy.

A decorated callable's logged output is indented only as much as is necessary. Here, the even-numbered functions don't indent, so the indented functions that they call are indented just one level more than their "inherited" indentation level:

```
>>> @log_calls()
... def g1():
...     pass
>>> @log_calls(indent=False)     # no extra indentation for g2
... def g2():
...     g1()
>>> @log_calls()
... def g3():
...     g2()
>>> @log_calls(indent=False)     # no extra indentation for g4
... def g4():
...     g3()
>>> @log_calls()
... def g5():
...     g4()
>>> g5()
g5 <== called by <module>
g4 <== called by g5
    g3 <== called by g4
    g2 <== called by g3
        g1 <== called by g2
        g1 ==> returning to g2
    g2 ==> returning to g3
    g3 ==> returning to g4
```

(continues on next page)

(continued from previous page)

```
g4 ==> returning to g5
g5 ==> returning to <module>
```

1.6.10 `name` (default: `''`)

The `name` parameter lets you change the "display name" of a decorated callable. The *display name* is the name by which *log_calls* refers to the callable, in these contexts:

- when logging a call to, and a return from, the callable
- when reporting its return value
- when it's in a *call chain*.

A value provided for the `name` parameter should be a string, of one of the following forms:

- the preferred name of the callable (a string literal), or
- an old-style format string with just one occurrence of `%s`, which the `__name__` of the decorated callable will replace.

For example:

```
>>> @log_calls(name='f (STUB)')
... def f(): pass
>>> f()
f (STUB) <== called by <module>
f (STUB) ==> returning to <module>
```

Another simple example:

```
>>> @log_calls(name='"%s" (lousy name)', log_exit=False)
... def g(): pass
>>> g()
"g" (lousy name) <== called by <module>
```

This parameter is useful mainly to simplify the display names of inner functions, and to disambiguate the display names of *getter* and *deleter* property methods.

If the `name` setting is empty (the default), the display name of a decorated callable is its `__qualname__`, followed by (a space and) its `__name__` in parentheses if the `__name__` is not a substring of the `__qualname__`.

Example — using `name` with an inner function

The qualified names of inner functions are ungainly – in the following example, the "qualname" of `inner` is `outer.<locals>.inner`:

```
>>> @log_calls()
... def outer():
...     @log_calls()
...     def inner(): pass
...     inner()
>>> outer()
outer <== called by <module>
    outer.<locals>.inner <== called by outer
    outer.<locals>.inner ==> returning to outer
outer ==> returning to <module>
```

You can use the `name` parameter to simplify the displayed name of the inner function:

```
>>> @log_calls()
... def outer():
...     @log_calls(name='%s')
...     def inner(): pass
...     inner()
>>> outer()
outer <== called by <module>
    inner <== called by outer
    inner ==> returning to outer
outer ==> returning to <module>
```

See *this* section on using the `name` parameter with *setter* and *deleter* property methods, which demonstrates the use of `name` to distinguish the methods of properties defined with `@property` decorators.

1.6.11 `prefix` (default: `''`)

The *prefix* keyword parameter lets you specify a string with which to prefix the name of a callable, thus giving it a new display name.

Here's a simple example:

```
>>> @log_calls(prefix='--- ')
... def f(): pass
>>> f()
--- f <== called by <module>
--- f ==> returning to <module>
```

Because versions 0.3.0+ of *log_calls* use `__qualname__` for the display name of decorated callables, what had been the main use case for `prefix` — prefixing method names with their class name — has gone away. Furthermore, clearly the `name` parameter can produce any display name that `prefix` can. However, `prefix` is not deprecated, at least not presently: for what it's worth, it *is* a setting and can be changed dynamically, neither of which is true of `name`.

1.6.12 `file` (default: `sys.stdout`)

The `file` parameter specifies a stream (an instance of `io.TextIOBase`) to which *log_calls* will print its messages. This value is supplied to the `file` keyword parameter of the `print` function, which has the same default value. This parameter is ignored if you've supplied a logger for output using the *logger* parameter.

When the output stream is the default `sys.stdout`, *log_calls* always uses the current meaning of that expression to obtain its output stream, not just what "sys.stdout" meant at program initialization. Your program can capture, change and redirect `sys.stdout`, and *log_calls* will write to that stream, whatever it currently is. (*doctest* is a good example of a program which manipulates `sys.stdout` dynamically.)

If your program writes to the console a lot, you may not want *log_calls* messages interspersed with your real output: your understanding of both logically distinct streams might be hindered, and it may be better to make them actually distinct. Splitting off the *log_calls* output can also be useful for understanding or for creating documentation: you can gather all, and only all, of the *log_calls* messages in one place. The `indent` setting will be respected, whether messages go to the console or to a file.

It's not easy to test this feature with *doctest*, so we'll just give an example of writing to `sys.stderr`, and then reproduce the output:

```
import sys
@log_calls(file=sys.stderr)
def f(n):
    if n <= 0:
        return 'a'
    return '(%s)' % f(n-1)
```

Calling `f(2)` returns `'((a))'` and writes the following to `sys.stderr`:

```
f <== called by <module>
    arguments: n=2
    f <== called by f
        arguments: n=1
        f <== called by f
            arguments: n=0
        f ==> returning to f
    f ==> returning to f
f ==> returning to <module>
```

1.6.13 `mute` (default: `log_calls.MUTE.NOTHING`)

The *mute* parameter gives you control over *log_calls* output from a given decorated callable. It can take any of the following three numeric values, shown here in increasing order:

`log_calls.MUTE.NOTHING` (default) doesn't mute any output

`log_calls.MUTE.CALLS` mutes all logging of function/method call details, but the output of any calls to the methods *log_calls.print()* and *log_calls.print_exprs()* is allowed through

`log_calls.MUTE.ALL` mutes all output of *log_calls*.

`mute` is a *setting* — part of the state maintained for a decorated callable — and can be changed dynamically.

Examples are best deferred until the `log_*()` methods are discussed: see *Indent-aware writing methods and muting — examples*.

The `mute` parameter lets *log_calls* behave just like the *record_history* decorator, collecting statistics silently which are accessible via the `stats` attribute of a decorated callable. See *The record_history Decorator* for a precise statement of the analogy; see the tests/examples in `tests/test_log_calls_as_record_history` for illustration.

The global mute switch `log_calls.mute` (default: `log_calls.MUTE.NOTHING`)

In addition to the `mute` settings maintained for each decorated callable, *log_calls* also has a single class attribute `log_calls.mute`. It can assume the same three values `log_calls.MUTE.NOTHING`, `log_calls.MUTE.CALLS`, and `log_calls.MUTE.ALL`. Before each write originating from a call to a decorated callable, *log_calls* uses the max of `log_calls.mute` and the callable's `mute` setting to determine whether to output anything. Thus, realtime changes to `log_calls.mute` take effect immediately.

To see this in action, refer to *Indent-aware writing methods and muting — examples*.

1.6.14 `settings` (default: `None`)

The `settings` parameter lets you collect several keyword parameter/value pairs in one place and pass them to *log_calls* with a single parameter. `settings` is a useful shorthand if you have, for example, a module with several *log_calls*-decorated functions, all with multiple, mostly identical settings which differ from *log_calls*'s defaults. Instead of repeating multiple identical settings across several uses of the decorator, a tedious and error-prone practice, you can gather them all into one `dict` or text file, and use the `settings` parameter to concisely specify them all *en masse*. You can use different groups of settings for different sets of functions, or classes, or modules — you're free to organize them as you please.

When not `None`, the `settings` parameter can be either a `dict`, or a `str` specifying the location of a *settings file* — a text file containing *key=value* pairs and optional comments. (Details about settings files, their location and their format appear below, in *settings as a pathname (str)*.) In either case, the valid keys are *the keyword parameters that are "settings"* (as defined in *What is a setting?*) plus, as a convenience, `NO_DECO`. *Invalid keys are ignored.*

The values of settings specified in the dictionary or settings file override *log_calls*'s default values for those settings, and any of the resulting settings are in turn overridden by corresponding keywords passed directly to the decorator. Of course, you *don't* have to provide a value for every valid key.

Note: The values can also be *indirect values* for parameters that allow indirection (almost all do), as described in the chapter *Indirect Values of Keyword Parameters*.

`settings` as a `dict`

The value of `settings` can be a `dict`, or more generally any object `d` for which it's true that `isinstance(d, dict)`. A simple example should suffice. Here is a settings `dict` and two *log_calls*-decorated functions using it:

```
>>> d = dict(
...     args_sep=' | ',
...     log_args=False,
...     log_call_numbers=True,
... )
>>> @log_calls(settings=d)
... def f(n):
...     if n <= 0: return
...     f(n-1)
```

```
>>> @log_calls(settings=d, log_args=True)
... def g(s, t): print(s + t)
```

```
>>> f(2)
f [1] <== called by <module>
    f [2] <== called by f [1]
        f [3] <== called by f [2]
        f [3] ==> returning to f [2]
    f [2] ==> returning to f [1]
f [1] ==> returning to <module>
```

```
>>> g('aaa', 'bbb')
g [1] <== called by <module>
    arguments: s='aaa' | t='bbb'
aaabbb
g [1] ==> returning to <module>
```

`settings` as a pathname (`str`)

When the value of the `settings` parameter is a `str`, it must be a path to a *settings file* — a text file containing *key*=*value* pairs and optional comments. If the pathname is just a directory, *log_calls* looks there for a file named `.log_calls` and uses that as a settings file; if the pathname is a file, *log_calls* uses that file. In either case, if the file doesn't exist then no error results *nor is any warning issued*, and the `settings` parameter is ignored.

Format of a settings file

A *settings file* is a text file containing zero or more lines of the form

setting_name=value

Whitespace is permitted around *setting_name* and *value*, and is stripped. Blank lines are ignored, as are lines whose first non-whitespace character is `#`, which therefore you can use as comments.

Here are the allowed "direct" values for settings in a settings file:

Setting	Allowed "direct" value

`log_args`
`log_retval`
`log_elapsed`
`log_exit`
`indent`
`log_call_numbers`
`record_history`
`NO_DECO`
	boolean
	(case-insensitive –
	`True`, `False`, `tRuE`, `FALSE`, etc.)

`enabled`	int, or case-insensitive boolean as above
`args_sep` `prefix`	string, enclosed in quotes
`loglevel` `max_history`	int
`file`	`sys.stdout` or `sys.stderr`, **not** enclosed in quotes (or `None`)
`logger`	name of a logger, enclosed in quotes (or `None`)

Warning: Ill-formed lines, bad values, and nonexistent settings are all ignored, **silently**.

Settings file example

Here's an example of what a settings file might contain:

```
args_sep    = ' | '
log_args    = False
log_retval = TRUE
logger      = 'my_logger'
# file: this is just for illustration, as logger takes precedence.
#       file can only be sys.stderr or sys.stdout [*** NOT IN QUOTES! ***] (or None)
file=sys.stderr
# ``log_elapsed`` has an indirect value:
log_elapsed='elapsed_='
# The following lines are bad in one way or another, and are ignored:
prefix=1492
loglevel=
no_such_setting=True
indent
```

Note: You can use the `log_calls.set_defaults()` classmethod to change the *log_calls* default settings, instead of passing the same `settings` argument to every `@log_calls(...)` decoration. See the chapter *Retrieving and Changing the Defaults*.

Where to find more examples

The test file `tests/test_log_call_more.py`, in the docstring of the function `main__settings()`, contains several examples (doctests) of the `settings` parameter. Two of the tests there use "good" settings files in the `tests/` directory: `.log_calls` and `log_calls-settings.txt`. Two more test what happens (nothing) when specifying a nonexistent file or a file with "bad" settings (`tests/bad-settings.txt`). Another tests the `settings` parameter as a `dict`.

1.6.15 `NO_DECO` (default: `None`)

The `NO_DECO` parameter prevents *log_calls* from decorating a callable or class: when true, the decorator returns the decorated thing itself, unwrapped and unaltered. Intended for use at program startup, it provides a single "true bypass" switch.

Using this parameter in a settings dict or settings file lets you control "true bypass" with a single switch, e.g. for production, without having to comment out every decoration.

`NO_DECO` can only prevent decoration, it cannot undo decoration.

For example, if `f` is already decorated, then:

```
f = log_calls(NO_DECO=True)(f)
```

has no effect: `f` remains decorated.

Use `NO_DECO=True` for production

Even even when it's disabled or bypassed, *log_calls* imposes some overhead. For production, therefore, it's best to not use it at all. One tedious way to guarantee that would be to comment out every `@log_calls()` decoration in every source file. `NO_DECO` allows a more humane approach: Use a settings file or settings dict containing project-wide settings, including an entry for `NO_DECO`. For development, use:

```
NO_DECO=False
```

and for production, change that to:

```
NO_DECO=True
```

Even though it isn't actually a "setting", `NO_DECO` is permitted in settings files and dicts in order to allow this.

Examples

The tests in `tests/test_no_deco__via_file.py` demonstrate using `NO_DECO` in an imported `dict` and in a settings file.

1.6.16 `override` (default: `False`)

The `override` parameter is mainly intended for use when redecorating functions and classes with the `log_calls.decorate_*` classmethods, as discussed in the chapter *Bulk (Re)Decoration, (Re)Decorating Imports*. `override` can also be used with class decorators to give its settings precedence over any explicitly given for callables or inner classes. See *Precedence of inner decorators over outer decorators* for a simple example, and *Example — decorating a class in scikit-learn* for a larger one.

1.6.17 Parameters Documented In Other Chapters

The remaining parameters are more specialized and require discussion of the contexts in which they are used. For completeness, we catalog them here, together with links to their documentation.

`omit`, `only` (defaults: `tuple()`)

In the chapter *Decorating Classes*, the section *The omit and only keyword parameters (default: ())* documents the two parameters that control which callables of a class get decorated. The value of each is a string or a sequence of strings; each string is either the name of a callable, or a "glob" pattern matching names of callables.

- `omit` — *log_calls* will *not* decorate these callables;
- `only` — *log_calls* decorates only these callables, excluding any specified by `omit`

`logger` (default: `None`), `loglevel` (default: `logging.DEBUG`)

Using Loggers presents the two parameters that let you output *log_calls* messages to a `Logger`:

- *logger* – a logger name (a `str`) or a `logging.Logger` object;
- *loglevel* (an `int`) – `logging.DEBUG`, `logging.INFO`, ..., or a custom loglevel.

`record_history` (default: `False`), `max_history` (default: `0`)

Call History and Statistics discusses the two parameters governing call history collection:

- *record_history* governs whether call history is retained, and then
- *max_history* controls how much (cache size).

1.7 Decorating Classes

In the *Decorating methods* and *Decorating classes* sections of the *Quick Start* chapter we already introduced the use of *log_calls* to decorate methods and properties of classes. As shown in the latter section, if you want to decorate every callable of a class, you don't have to decorate each one individually: you can simply decorate the class. As that section also shows, this convenience isn't an all or nothing affair: you can use the `omit` and `only` keyword parameters to a *log_calls* class decorator for more fine-grained control over which callables get decorated. The first sections of this chapter detail the use of those parameters. The remaining sections discuss other topics pertinent to class decoration.

1.7.1 The `omit` and `only` keyword parameters (default: `()`)

These parameters let you concisely specify which methods and properties of a decorated class get decorated. *log_calls* ignores `omit` and `only` when decorating a function. When not empty, the value of each of these parameters specifies one or more methods or properties of a decorated class. If you provide just `omit`, all callables of the class will be decorated except for those specified by `omit`. If you provide just `only`, only the callables it specifies will be decorated. If you provide both, the callables decorated will be those specified by `only`, excepting any specified by `omit`.

log_calls allows considerable flexibility in the format of values provided for the `omit` and `only` parameters. First, we give the general definition of what those values can be, and then several examples illustrating their use.

Values of the `omit` and `only` parameters

For this section only, it's convenient to define the following term:

callable designator (That's *designator of callables*, not a "designator that can be called", whatever that might be.)

A string which is one of the following:

- the name of a method
- the name of a property, possibly followed by one of the qualifiers `.setter`, `.getter`, `.deleter`
- a *glob* (Unix-style shell pattern) — a string possibly containing
  - wildcard characters `*`, `?`

  - character sets `[` s_1 s_2 ... s_n `]` where each s_k can be either a character or a character range $s_{k,1}$ – $s_{k,2}$ (e.g. `[acr-tx]`, which denotes `acrstx`)
  - complements of character sets `[!` s_1 s_2 ... s_n `]` — all characters *except* those denoted by `[` s_1 s_2 ... s_n `]`

  Matching of globs against method and property names is case-sensitive.

A value of the `omit` or `only` parameter can be:

- A single callable designator,
- a string consisting of multiple callable designators separated by spaces or by commas and spaces, or
- a sequence (list, tuple, or other iterable) of callable designators.

Examples of callable designators

Given the following class `X`:

```
>>> class X():
...     def fn(self): pass
...     def gn(self): pass
...     def hn(self): pass
...     @property
...     def pr(self): pass
...     @pr.setter
...     def pr(self, val): pass
...     @pr.deleter
...     def pr(self): pass
...
...     def pdg(self): pass
...     def pds(self, val): pass
...     pd = property(pdg, pds, None)
...     class I():
...         def i1(self): pass
...         def i2(self): pass
```

the following table shows several callable designators for `X` and what they designate. To reduce clutter, we've omitted initial `X.` from the literals in the righthand column:

Callable designator	designates these methods and/or properties
`fn`	`fn`
`pr.getter`	`pr` getter
`pr.setter`	`pr` setter
`pr.deleter`	`pr` deleter
`pr`	{`pr` getter, `pr` setter, `pr` deleter}
`pr.?etter`	{`pr` getter, `pr` setter}
`p*`	{`pr` getter, `pr` setter, `pr` deleter, `pds`, `pdg`}
`?n`	{`fn`, `gn`, `hn`}
`[fg]n`	{`fn`, `gn`}
`[f-h]n`	{`fn`, `gn`, `hn`}
`pd`	{`pdg`, `pds`}
`pdg`, `pd.getter`	`pdg`
`pds`, `pd.setter`	`pds`
`pd.deleter`	nothing (`pd` has no deleter)
`no_such_*`	nothing (there are no matches)

Warning: Be aware that:

1. wildcards can match the dot `'.'` in qualified names;
2. both qualified and unqualified method and property names are matched — e.g. for a method `mymethod` in a class `C`, each callable designator is checked for a match against both `mymethod` and `C.mymethod`.

As the second and third to last examples in the above table illustrate, these matching rules can lead to surprises, especially when using complements of character sets.

1.7.2 `omit` and `only` — Examples

A useful settings dict for the examples of this chapter:

```
>>> MINIMAL = dict(
...     log_args=False,
...     log_exit=False
... )
```

Basic examples

First, simple examples for methods, without wildcards, illustrating possible values for `omit` and `only` and the interaction of those parameters.

In class `A`, only `f` is decorated:

```
>>> @log_calls(only='f', settings=MINIMAL)
... class A():
...     def f(self): pass
...     def g(self): pass
>>> a = A(); a.f(); a.g()
A.f <== called by <module>
```

In class `B`, `f` and `g` are omitted, so only `h` is decorated (and so, gives output):

```
>>> @log_calls(omit='f g', settings=MINIMAL)
... class B():
...     def f(self): pass
...     def g(self): pass
...     def h(self): pass
>>> b = B(); b.f(); b.g(); b.h()
B.h <== called by <module>
```

In class `C`, only `f` and `h` are decorated:

```
>>> @log_calls(only='f, h', settings=MINIMAL)
... class C():
...     def f(self): pass
...     def g(self): pass
...     def h(self): pass
>>> c = C(); c.f(); c.g(); c.h()
```

(continues on next page)

(continued from previous page)

```
C.f <== called by <module>
C.h <== called by <module>
```

In class `D`, only `f` is decorated:

```
>>> @log_calls(only=['f', 'g'], omit=('g',), settings=MINIMAL)
... class D():
...     def f(self): pass
...     def g(self): pass
...     def h(self): pass
>>> d = D(); d.f(); d.g(); d.h()
D.f <== called by <module>
```

Precedence of inner decorators over outer decorators

By default, the *explicitly given* settings of a callable's decorator take precedence over those of the decorator of its class:

```
>>> @log_calls(settings=MINIMAL)
... class E():
...     def f(self): pass
...     @log_calls(log_exit=True)
...     def g(self): pass
>>> E().f(); E().g()
E.f <== called by <module>
E.g <== called by <module>
E.g ==> returning to <module>
```

The same holds for inner classes: settings provided explicitly to the decorator of an inner class take precedence over the corresponding settings of the outer class. To give the outer settings priority, supply `override=True` to the outer decorator:

```
>>> @log_calls(settings=MINIMAL, override=True)
... class E():
...     def f(self): pass
...     @log_calls(log_exit=True)
...     def g(self): pass
>>> E().f(); E().g()
E.f <== called by <module>
E.g <== called by <module>
```

Decorating properties

There are two ways to specify properties: using `property` as a decorator, and using it as a function, as described in the Python documentation for property. *log_calls* handles both approaches. The name of the property alone, with no appended qualifier, designates *all* of the property's existing callables — the *getter*, *setter*, and *deleter*.

Decorating properties specified with the `@property` decorator

Python lets you define properties using decorators. You decorate the *getter* property *prop* with `@property`, and then any corresponding *setter* and *deleter* methods with `@`*prop*`.setter` and `@`*prop*`.deleter` respectively.

Using `only` to decorate just the *getter*:

```
>>> @log_calls(only='prop.getter', settings=MINIMAL)
... class A():
...     @property
...     def prop(self): pass
...     @prop.setter
...     def prop(self, val): pass
>>> A().prop; A().prop = 17
A.prop <== called by <module>
```

Using `only` with the property name — all property methods are decorated:

```
>>> @log_calls(only='prop', settings=MINIMAL)
... class A():
...     @property
...     def prop(self): pass
...     @prop.setter
...     def prop(self, val): pass
...     @prop.deleter
...     def prop(self): pass
>>> A().prop; A().prop = 17; del A().prop
A.prop <== called by <module>
A.prop <== called by <module>
A.prop <== called by <module>
```

Using the `name` parameter with *setter* and *deleter* property methods

As the previous example shows, *log_calls* cannot presently give distinct display names to the different callables of a property defined by decorators. However, you can use the `name` parameter to overcome this limitation, as shown in the following example. (The *log_calls* decorators come after the property decorators.)

```
>>> @log_calls(settings=MINIMAL)
... class A():
...     @property
...     def prop(self): pass
...
...     @prop.setter
...     @log_calls(name='A.%s.setter')
...     def prop(self, val): pass
...
...     @prop.deleter
...     @log_calls(name='A.%s.deleter')
...     def prop(self, val): pass
>>> A().f(); A().prop; A().prop = 17; del A().prop
A.prop <== called by <module>
A.prop.getter <== called by <module>
A.prop.deleter <== called by <module>
```

Decorating properties specified with the `property` function

Python also lets you define properties using `property` as a function. *log_calls* uses the unique names of the methods that comprise the property.

```
>>> @log_calls(omit='prop.setter', settings=MINIMAL)
... class XX():
...     def getxx(self):        pass
...     def setxx(self, val):   pass
...     def delxx(self):        pass
...     prop = property(getxx, setxx, delxx)
>>> xx = XX(); xx.prop; xx.prop = 5; del xx.prop
XX.getxx <== called by <module>
XX.delxx <== called by <module>
```

1.7.3 Decorating inner classes

By default, the explicitly given settings of a decorator of (or within) an inner class take precedence over those of the decorator of its outer class.

```
>>> @log_calls(settings=MINIMAL)
... class O():
...     def f(self): pass
...     class I():
...         @log_calls(log_call_numbers=True)
...         def fi(self): pass
...         def gi(self): pass
O().f(); O().I().fi(); O().I().gi()
O.f <== called by <module>
O.I.fi [1] <== called by <module>
O.I.gi <== called by <module>
```

To give the outer settings priority, supply `override=True` to the outer decorator, as illustrated above in *Precedence of inner decorators over outer decorators*.

This default precedence of outer over inner is different for `omit`, in a way that attempts to meet expectations:

`only` on inner and outer class decorators

When present and nonempty, inner `only` overrides outer `only`. In `I1`, only `g1` is decorated, despite the outer class's `only` specifier:

```
>>> @log_calls(only='*_handler', settings=MINIMAL)
... class O():
...     def f(self): pass
...     def my_handler(self): pass
...     def their_handler(self): pass
...     @log_calls(only='g1')
...     class I1():
...         def g1(self): pass
...         def some_handler(self): pass
>>> oi1 = O.I1(); oi1.g1(); oi1.some_handler()
O.I1.g1 <== called by <module>
```

`omit` on inner and outer class decorators

`omit` is cumulative — inner `omit` is added to outer `omit`:

```
>>> @log_calls(omit='*_handler', settings=MINIMAL)
... class O():
...     def f(self): pass
...     def my_handler(self): pass
...     def their_handler(self): pass
...     @log_calls(omit='*_function')
...     class I1():
...         def g1(self): pass
...         def some_handler(self): pass
...         def some_function(self): pass
>>> oi1 = O.I1(); oi1.g1(); oi1.some_handler(); oi1.some_function()
O.I1.g1 <== called by <module>
```

Further examples

For more examples of inner class decoration, consult the docstrings of the functions `main__lc_class_deco__inner_classes()` and `main__lc_class_deco__omit_only__inner_classes()` in `tests/test_log_calls__class_deco.py`.

1.7.4 *log_calls* does not decorate `__repr__`

To avoid infinite, possibly indirect recursions, *log_calls* does not itself decorate `__repr__` methods, but it will decorate them with `reprlib.recursive_repr()`:

```
>>> @log_calls()
... class A():
...     def __init__(self, x): self.x = x
...     def __repr__(self): return str(self.x)
```

The `__init__` method is decorated:

```
>>> a = A(5)
A.__init__ <== called by <module>
    arguments: self=<__main__.A object at 0x...>, x=5
A.__init__ ==> returning to <module>
```

but `__repr__` is not:

```
>>> print(a)     # no log_calls output
5
```

1.8 Writing *log_calls*-Aware Debugging Messages

log_calls provides the `log_calls.print` method as a better alternative to the global `print` function for writing debugging messages. It also provides the `log_calls.print_exprs` method as an easy way to "dump" variables and expressions together with their values.

`log_calls.print` writes to the *log_calls* output stream — whether that's a console stream, file or logger — where its output is properly synced and aligned with respect to the decorated callable it was called from. If later you undecorate the callable (for example, by deleting or commenting out the decorator, or by passing it `NO_DECO=True`), you don't *have* to remove the call to `log_calls.print`, because by default the method writes something only when called from within a decorated callable, and doesn't raise an exception otherwise.

It's quite typical to use debugging messages to print out (or "dump") the values of local variables and expressions, together with labels. Accomplishing this with `print` or even `log_calls.print` usually requires ad-hoc though boilerplate string formatting. As a convenience, *log_calls* provides the *log_calls.print_exprs()* method, which prints variables and expressions together with their values in the context of the caller.

1.8.1 The `log_calls.print()` method

When you call `log_calls.print('my message')` from within a decorated callable `f`, `my message` appears in the *log_calls* output stream, between the "entering" and "exiting" parts of the *log_calls* report for the call to `f`, aligned with the "arguments" section:

```
>>> @log_calls()
... def f(x):
...     log_calls.print('About to try new approach...')
...     return 4 * x
>>> f(2)
f <== called by <module>
    arguments: x=2
    About to try new approach...
f ==> returning to <module>
8
```

If you undecorate the callable — say, by deleting or commenting out the decorator, or by passing it `NO_DECO=True` — by default you *don't* have to comment out the call to `log_calls.print`, as it won't write anything and won't raise an exception:

```
>>> # @log_calls()
>>> def f(x):
...     log_calls.print('About to try new approach...')
...     return 4 * x
>>> f(2)
8
```

You can change this default by setting the *log_calls* global variable `log_calls.print_methods_raise_if_no_deco` to `True`, as discussed *below*.

log_calls.print() details

log_calls.**print**(**msgs*, *sep=' '*, *extra_indent_level=1*, *prefix_with_name=False*)

Join one or more messages with sep, and write the result to the *log_calls* output destination of the caller, a decorated callable. The "messages" are strings, or objects to be displayed as strs. The method does nothing if no messages are passed.

Parameters

- **msgs** – messages to write.
- **extra_indent_level** (int) – a number of 4-column-wide *indent levels* specifying where to begin writing that message. This value x 4 is an offset in columns from the left margin of the visual frame established by *log_calls* – that is, an offset from the column in which the callable's entry and exit messages begin. The default of 1 aligns the message with the "arguments: " line of *log_calls*'s output.
- **prefix_with_name** (bool) – If true, the final message is prefaced with the name of the callable, plus its call number in square brackets if the log_call_numbers setting is true.

Raises AttributeError if called from within an undecorated callable and log_calls.print_methods_raise_if_no_deco is true.

Note: If the *mute* setting of the caller is log_calls.MUTE.CALLS, log_calls.print() forces prefix_with_name to True, and extra_indent_level to 0. A little reflection should reveal that these are sensible adjustments. See the following sections for examples.

1.8.2 Writing expressions and their values with log_calls.print_exprs()

log_calls.print_exprs() is a convenience method built upon log_calls.print() which makes it easy to print variables and expressions together with their values.

The *Writing log_calls-aware debugging messages* section of the *Quick Start* chapter contains a few examples. Others can be found in the docstring of the function test__log_exprs() in tests/test_log_calls_v30_minor_features_fixes.py, and in the test module tests/test_log_calls_log_methods.py.

log_calls.print_exprs() details

log_calls.**print_exprs**(**exprs*, *sep=', '*, *extra_indent_level=1*, *prefix_with_name=False*, *prefix=''*, *suffix=''*)

Evaluate each expression in exprs in the context of the caller, a decorated callable; make a string *expr* = *val* from each, and pass those strings to (the internal method called by) log_calls.print() as messages to write, separated by sep.

Parameters

- **exprs** (sequence of str) – expressions to evaluate and log with their values
- **sep** – separator for *expr* = *val* substrings
- **extra_indent_level** – as for log_calls.print()
- **prefix_with_name** – as for log_calls.print()
- **prefix** – additional text to prepend to output message.

- **suffix** – additional text to append to output message.

Raises AttributeError if called from within an undecorated callable and `log_calls.print_methods_raise_if_no_deco` is true.

1.8.3 The global variable `log_calls.print_methods_raise_if_no_deco` (default: `False`)

By default (when `print_methods_raise_if_no_deco == False`), if you call `log_calls.log_*` from within a method or function that isn't decorated, it does nothing (except waste a few cycles). You can comment out or delete the `@log_calls` decorator, or use the `NO_DECO` parameter to suppress decoration, and the `.log_*` method calls will play nicely: they won't output anything, **and** the calls won't raise an exception. In short, leaving the `log_calls.log_*` lines uncommented is as benign as it can be.

But probably at some point you *do* want to know when you have lingering code that's supposedly development-only. *log_calls* will inform you of that if you set `log_calls.print_methods_raise_if_no_deco` to `True` (or any truthy value).

When this flag is true, calls to `log_calls.print` and `log_calls.print_exprs` from within an undecorated function or method will raise `AttributeError`. This compels you to comment out or delete any calls to `log_calls.log_*` from within undecorated functions or methods. (A call to `log_calls.log_*` from within a callable that *never* was decorated is just a mistake, and it *should* raise an exception; with this flag set to true, it will.)

1.8.4 Indent-aware writing methods and muting — examples

Presently, "muting" has three states, of a possible four:

- `log_calls.MUTE.NOTHING` — mute nothing
- `log_calls.MUTE.CALLS` — mute the output of the `@log_calls` decorators while allowing the output of the `log_calls.log_*` methods
- `log_calls.MUTE.ALL` — mute all *log_calls* output

There's a global mute, `log_calls.mute`, and each decorated callable has its own `mute` setting.

Examples using the *mute* setting

When a decorated callable is not muted (its `mute` setting is `log_calls.MUTE.NOTHING`, i.e. `False`, the default), *log_calls* produces output as do `log_calls.print()` and `log_calls.print_exprs()`:

```
>>> @log_calls()
... def f():
...     log_calls.print('Hello, world!')
>>> f()
f <== called by <module>
    Hello, world!
f ==> returning to <module>
```

When the callable's `mute` setting is `log_calls.MUTE.CALLS`, no extra indent level is added, and messages are prefixed with the callable's display name:

```
>>> f.log_calls_settings.mute = log_calls.MUTE.CALLS
>>> f()
f: Hello, world!
```

When the callable's `mute` setting is `log_calls.MUTE.ALL`, `log_calls.print()` and `log_calls.print_exprs()` produce no output:

```
>>> f.log_calls_settings.mute = log_calls.MUTE.ALL
>>> f()      # (no output)
```

Using global *mute*

Setting `log_calls.mute = log_calls.MUTE.CALLS` mutes *decorator* output from all decorated callables, allowing only output from `log_calls.log_*` methods.

global mute interactions with the *mute* setting — examples

First, define a couple of simple functions:

```
>>> @log_calls()
... def g(): log_calls.print("Hi")
>>> @log_calls()
... def f(): log_calls.print("Hi"); g()
```

Assume that `log_calls.mute == log_calls.MUTE.NOTHING`, which is the default. Calling `f()` gives all possible output:

```
>>> f()
f <== called by <module>
    Hi
    g <== called by f
        Hi
    g ==> returning to f
f ==> returning to <module>
```

Now change `log_calls.mute`, call `f()`, and observe the effects:

```
>>> log_calls.mute = log_calls.MUTE.CALLS
>>> f()
f: Hi
    g: Hi
```

```
>>> log_calls.mute = log_calls.MUTE.ALL
>>> f()      # (no output)
```

Now alter `log_calls.mute` and `g.log_calls_settings.mute`, call `f()`, and observe the effects:

```
>>> log_calls.mute = log_calls.MUTE.NOTHING
>>> g.log_calls_settings.mute = log_calls.MUTE.CALLS
>>> f()
f <== called by <module>
    Hi
    g: Hi
f ==> returning to <module>
```

```
>>> log_calls.mute = log_calls.MUTE.CALLS
>>> g.log_calls_settings.mute = log_calls.MUTE.ALL
>>> f()
f: Hi
```

Further examples can be found in `tests/test_log_calls_v30_minor_features_fixes.py`. `test__global_mute()` illustrate that global mute is always checked in realtime; `test__log_message__indirect_mute()` illustrates using an indirect value for the `mute` setting.

1.8.5 Using `log_calls.print()` in classes

The following class illustrates all possibilities of calling `log_calls.print()` from a method. To reduce clutter in this example, *log_calls* call output is muted, and therefore `.print()` automatically prefixes its output with the name of the caller, and doesn't indent by an extra 4 spaces:

```
>>> @log_calls(omit='no_deco', mute=log_calls.MUTE.CALLS)
... class B():
...     def __init__(self):
...         log_calls.print('Hi')
...     def method(self):
...         log_calls.print('Hi')
...     def no_deco(self):
...         log_calls.print('Hi')
...     @classmethod
...     def clsmethod(cls):
...         log_calls.print('Hi')
...     @staticmethod
...     def statmethod():
...         log_calls.print('Hi')
...
...     @property
...     def prop(self):
...         log_calls.print('Hi')
...     @prop.setter
...     @log_calls(name='B.%s.setter')  # o/w, display name of setter is also 'B.prop'
...     def prop(self, val):
...         log_calls.print('Hi')
...
...     def setx(self, val):
...         log_calls.print('Hi from setx alias x.setter')
...     def delx(self):
...         log_calls.print('Hi from delx alias x.deleter')
...     x = property(None, setx, delx)
```

```
>>> b = B()
B.__init__: Hi
>>> b.method()
B.method: Hi
>>> b.no_deco()      # outputs nothing
>>> b.statmethod()
B.statmethod: Hi
>>> b.clsmethod()
B.clsmethod: Hi
```

(continues on next page)

(continued from previous page)

```
>>> b.prop
B.prop: Hi
>>> b.prop = 17
B.prop.setter: Hi
>>> b.x = 13
B.setx: Hi from setx alias x.setter
>>> del b.x
B.delx: Hi from delx alias x.deleter
```

Observe that the call to `b.no_deco()` does nothing, even though the method isn't decorated. If `log_calls.print_methods_raise_if_no_deco` were true, the call from `b.no_deco()` to `log_calls.print` would raise `AttributeError`.

1.8.6 *wrapper*`.log_message()`, *wrapper*`.log_exprs()` [deprecated]

Before `log_calls.print()` and `log_calls.print_exprs()` existed, the methods `log_message()` and `log_exprs()` provided similar functionality. Using these within a class was (and remains) somewhat clumsy: a method or property must first access its own "wrapper" order to use these methods. The section on *accessing wrappers of methods* shows how to do so.

1.9 Using Loggers

log_calls works well with loggers obtained from Python's `logging` module, objects of type `logging.Logger`. If you use the `logger` keyword parameter, *log_calls* will write to the specified logger rather than to a file or stream using `print`.

We'll need a logger for the examples of this chapter — a simple one with a single handler that writes to the console. Because *doctest* doesn't capture output written to `stderr` (the default stream to which console handlers write), we'll send the console handler's output to `stdout`, using the format `<loglevel>:<loggername>:<message>`.

```
>>> import logging
>>> import sys
>>> ch = logging.StreamHandler(stream=sys.stdout)
>>> c_formatter = logging.Formatter('%(levelname)8s:%(name)s:%(message)s')
>>> ch.setFormatter(c_formatter)
>>> logger = logging.getLogger('a_logger')
>>> logger.addHandler(ch)
>>> logger.setLevel(logging.DEBUG)
```

1.9.1 The `logger` keyword parameter (default: `None`)

The `logger` keyword parameter tells *log_calls* to write its output using that logger rather than to the `file` setting using the `print` function. If the `logger` setting is nonempty, it takes precedence over `file`. The value of `logger` can be either a logger instance (a `logging.Logger`) or a string giving the name of a logger, which will be passed to `logging.getLogger()`.

```
>>> @log_calls(logger=logger)
... def somefunc(v1, v2):
...     logger.debug(v1 + v2)
```

(continues on next page)

(continued from previous page)

```
>>> somefunc(5, 16)
DEBUG:a_logger:somefunc <== called by <module>
DEBUG:a_logger:    arguments: v1=5, v2=16
DEBUG:a_logger:21
DEBUG:a_logger:somefunc ==> returning to <module>
```

Instead of passing the logger instance as above, we can simply pass its name, `'a_logger'`:

```
>>> @log_calls(logger='a_logger', indent=False)
... def anotherfunc():
...     somefunc(17, 19)
>>> anotherfunc()
DEBUG:a_logger:anotherfunc <== called by <module>
DEBUG:a_logger:somefunc <== called by anotherfunc
DEBUG:a_logger:    arguments: v1=17, v2=19
DEBUG:a_logger:36
DEBUG:a_logger:somefunc ==> returning to anotherfunc
DEBUG:a_logger:anotherfunc ==> returning to <module>
```

This works because

> "all calls to `logging.getLogger(name)` with a given name return the same logger instance", so that "logger instances never need to be passed between different parts of an application".
>
> —Python documentation for logging.getLogger().

Note: If the value of `logger` is a `Logger` instance that has no handlers (which can happen if you specify a logger name for a (theretofore) nonexistent logger), that logger won't be able to write anything, so *log_calls* will fall back to `print`.

1.9.2 The `loglevel` keyword parameter (default: `logging.DEBUG`)

log_calls also takes a `loglevel` keyword parameter, an `int` whose value must be one of the `logging` module's constants - `logging.DEBUG`, `logging.INFO`, etc. – or a custom logging level if you've added any. *log_calls* writes output messages using `logger.log(loglevel, ...)`. Thus, if the `logger`'s log level is higher than `loglevel`, no output will appear:

```
>>> logger.setLevel(logging.INFO)    # raise logger's level to INFO
>>> @log_calls(logger='a_logger', loglevel=logging.DEBUG)
... def f(x, y, z):
...     return y + z
>>> # No log_calls output from f
>>> # because loglevel for f < level of logger
>>> f(1,2,3)
5
```

1.9.3 Where to find further examples

A realistic example can be found in *Using a logger with multiple handlers that have different loglevels*.

Yet more examples appear in the docstrings of the function

`main_logging()`

in `test_log_calls.py`, and of the functions

`main__more_on_logging__more()`, `main__logging_with_indent__minimal_formatters()`, and `main__log_message__all_possible_output_destinations()`

in `test_log_calls_more.py`.

1.10 Retrieving and Changing the Defaults

1.10.1 *log_calls* classmethods `set_defaults()`, `reset_defaults()`

The *settings parameter* lets you specify an entire collection of *settings* at once. If you find that you're passing the same settings dict or settings file to most *log_calls* decorators in a program, *log_calls* offers a further economy. At program startup, you can use the `log_calls.set_defaults` classmethod to change the *log_calls* defaults to the settings you want, and eliminate most of the `settings` arguments.

classmethod `log_calls.`**set_defaults**(*new_default_settings=None*, ***more_defaults*)
Change the *log_calls* default values for settings, different from the "factory defaults".

Parameters

- **new_default_settings** (`dict` (a settings dict) or `str` (pathname for a settings file)) – a settings dict or settings file: any valid value for the *settings parameter*.
- **more_defaults** – keyword parameters where every key is a *setting*. These override settings in `new_default_settings`.

The new defaults are not retroactive! (Settings of already-decorated callables remain unchanged.) They apply to every decoration that occurs subsequently.

You can easily undo all changes effected by `set_defaults()`:

classmethod `log_calls.`**reset_defaults**()
Restore the "factory default" defaults.

Examples

Although these are "toy" examples, they illustrate how the `*_defaults()` methods behave:

Decorate `f` with "factory defaults":

```
>>> @log_calls()
... def f(x, y): return x
```

Define a settings dict:

```
>>> new_settings = dict(
...     log_call_numbers=True,
...     log_exit=False,
```

(continues on next page)

(continued from previous page)

```
...     log_retval=True,
... )
```

Call `set_defaults()` with the above settings dict, and in addition change the default for `args_sep`:

```
>>> log_calls.set_defaults(new_settings, args_sep=' $ ')
```

Decorate `g` while these defaults are in force:

```
>>> @log_calls()
... def g(x,y): return y
```

Restore the "factory defaults":

```
>>> log_calls.reset_defaults()
```

and decorate `h`:

```
>>> @log_calls()
... def h(u, v): return v
```

Call `f`, `g`, and `h`: only `g` will use the defaults of the `set_defaults()` call:

```
>>> _ = f(0, 1); _ = g(2, 3); _ = h(4, 5)
f <== called by <module>
    arguments: x=0, y=1
f ==> returning to <module>
g [1] <== called by <module>
    arguments: x=2 $ y=3
    g [1] return value: 3
h <== called by <module>
    arguments: u=4, v=5
h ==> returning to <module>
```

1.10.2 *log_calls* classmethods `get_defaults_OD()`, `get_factory_defaults_OD()`

For convenience, *log_calls* also provides classmethods for retrieving the current defaults and the "factory defaults", each as an `OrderedDict`:

classmethod log_calls.**get_defaults_OD**()
: Return an `OrderedDict` of the current *log_calls* defaults.

classmethod log_calls.**get_factory_defaults_OD**()
: Return an `OrderedDict` of the *log_calls* "factory defaults".

Examples

If `log_calls.set_default()` has not been called, then the current defaults *are* the factory defaults:

```
>>> log_calls.get_defaults_OD() == log_calls.get_factory_defaults_OD()
True
```

The dictionaries returned by the `get*_defaults_OD()` methods can be compared with those obtained from a callable's `log_calls_settings.as_OD()` or `log_calls_settings.as_dict()` method to determine whether, and if so how, the callable's settings differ from the defaults.

```
>>> def dict_minus(d1, d2: 'Mapping') -> dict:
...     """Return a dict of the "dictionary difference" of d1 and d2:
...     all items in d1 such that either the key is not in d2,
...     or the key is in both but values differ.
...     """
...     return {
...         key: val for key, val in d1.items()
...         if not (key in d2 and val == d2[key])
...     }
```

```
>>> @log_calls(log_exit=False)
>>> def func(): pass
>>> dict_minus(func.log_calls_settings.as_OD(), log_calls.get_defaults_OD())
{'log_exit': False}
```

1.11 Bulk (Re)Decoration, (Re)Decorating Imports

This chapter discusses the `log_calls.decorate_*` classmethods. These methods allow you to:

- decorate or redecorate functions and classes,
- decorate an entire class hierarchy (a class and all its subclasses), and even
- decorate all classes and/or functions in a module.

These methods are handy in situations where altering source code is impractical (too many things to decorate) or questionable practice (third-party modules and packages). They can also help you learn a new codebase, by shedding light on its internal operations.

The `decorate_*` methods provide another way to dynamically change the settings of already-decorated functions and classes.

Like any decorator, *log_calls* is a *functional* — a function that takes a function argument and returns a function. The following typical use:

```
@log_calls()
def f(): pass
```

is equivalent to:

```
f = log_calls()(f)
```

If `f` occurs in your own code, then no doubt you'll prefer the former. The `log_calls.decorate_*` methods let you decorate `f` when its definition does *not* necessarily appear in your code.

Note: You can't decorate Python builtins. Attempting to do is harmless (anyway, it's supposed to be!), and *log_calls* will return the builtin class or callable unchanged. For example, the following have no effect:

```
log_calls.decorate_class(dict)
log_calls.decorate_class(dict, only='update')
log_calls.decorate_function(dict.update)
```

1.11.1 Decorating classes programmatically

Decorating a class and optionally, all of its subclasses

classmethod log_calls.**decorate_class**(*klass: type*, *decorate_subclasses=False*, ***setting_kwargs*) → None

Decorate class `klass` and, optionally, all of its descendants recursively. If `decorate_subclasses == True`, and if any subclasses are decorated, their explicitly given *settings* remain unchanged by those in `setting_kwargs` *unless* `override=True` is in `setting_kwargs`.

`log_calls.decorate_class(C, **kwds)` is basically a syntactically sweetened version of `log_calls(**kwds)(C)`, with the addition of the flag parameter `decorate_subclasses`. There's another difference, however: `log_calls.decorate_class(...)` returns `None`, whereas `log_calls(**kwds)(C)` returns `C`.

Decorating a class and all of its subclasses

classmethod log_calls.**decorate_hierarchy**(*baseclass: type*, ***setting_kwargs*) → None

Decorate `baseclass` and, recursively, all of its descendants. If any subclasses are directly decorated, their explicitly given *settings* remain unchanged by those in `setting_kwargs` *unless* `override=True` is in `setting_kwargs`.

This is just a shorthand for `log_calls.decorate_class(baseclass, decorate_subclasses=True, **setting_kwargs)`.

1.11.2 Decorating functions programmatically

Decorating a function in your namespace

classmethod log_calls.**decorate_function**(*f: 'Callable'*, ***setting_kwargs*) → None

Decorate `f` using `settings_kwds`, and replace the definition of `f.__name__` with the decorated function (i.e. the wrapper) in the global namespace *of the caller*.

Parameters

- **f** – a function object, with no package/module qualifier: however it would be referred to in code at the point of the call to `decorate_function`. `f` itself refers to a function which is either defined in or imported into the module of the caller.
- **setting_kwargs** – settings for decorator

`log_calls.decorate_function(f, **kwds)` is basically a syntactically sweetened version of `log_calls(**kwds)(f)`. However, `log_calls.decorate_function(...)` returns `None`, whereas `log_calls(**kwds)(f)` returns the wrapper of `f`.

Decorating an "external" function in a package

classmethod log_calls.**decorate_package_function**(*f: 'Callable'*, ***setting_kwargs*) → None

Decorate `f` using settings in `settings_kwds`; replace the definition of `f.__name__` with the decorated function in the `__dict__` of the *module* of `f`.

Parameters

- **f** – a function object, qualified with a package, e.g. `somepackage.somefunc`, however it would be referred to in code at the point of a call to `decorate_package_function`.
- **setting_kwargs** – settings for decorator

Decorating an "external" function in a module

classmethod log_calls.**decorate_module_function**(*f: 'Callable'*, ***setting_kwargs*) → None

Decorate `f` using settings in `settings_kwds`; replace the definition of `f.__name__` with the decorated function in the `__dict__` of the module of `f`.

Parameters

- **f** – a function object, qualified with a module, e.g. `thatmodule.afunc`, however it would be referred to in code at the point of a call to `decorate_module_function`.
- **setting_kwargs** – settings for decorator

1.11.3 Decorating all functions and/or classes in a module

`decorate_module` lets you decorate the functions and/or classes of an imported module:

classmethod log_calls.**decorate_module**(*cls*, *mod: 'module'*, *functions: bool = True*, *classes: bool = True*, ***setting_kwargs*) → None

Parameters

- **mod** – module whose members are to be decorated
- **functions** – decorate all functions in `mod` if true
- **classes** – decorate all classes in `mod` if true
- **setting_kwargs** – keyword parameters for decorator

Raises TypeError

1.11.4 Examples

These modules in the `tests/` subdirectory contain several examples:

- `test_decorate_module.py` The docstring of the function `test_decorate_module()` contains simple tests of decorating the module `tests/some_module.py`.

A few examples/tests use the *Skikit-Learn* package if it's installed. (The following subsection reproduces one of them.) Those in these two modules are run by `run_tests.py`:

- `test_decorate_sklearn_KMeans.py`
- `test_decorate_sklearn_KMeans_functions.py`

The test in the following module decorates an entire module of *Skikit-Learn*:

- `_test_decorate_module_of_sklearn.py`

As the settings it imposes mess up the other `sklearn` tests, it is **not** run by `run_tests.py`. It can be run separately.

Example — decorating a class in *scikit-learn*

This example demonstrates:

- decorating a class that's not part of your project (unless you're working on scikit-learn:), and
- using the `override` parameter with one of the `log_calls.decorate_*` functions to dynamically change the settings of (all the callables of) an already-decorated class.

Except for the `log_calls.decorate_*` calls, the following code is excerpted from the *sklearn* site, e.g. Demonstration of k-means assumptions. The double backslashes in the two added lines accommodate *doctest*.

```
>>> from log_calls import log_calls
>>> from sklearn.cluster import KMeans
>>> from sklearn.datasets import make_blobs
>>> n_samples = 1500
>>> random_state = 170
>>> X, y = make_blobs(n_samples=n_samples, random_state=random_state)
```

First, let's decorate the class hierarchy, with settings that show just the call tree:

```
>>> log_calls.decorate_hierarchy(KMeans, log_args=False)        ### THIS LINE ADDED
```

Now let's call `KMeans.fit_predict`:

```
>>> y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(X)
KMeans.__init__ <== called by <module>
KMeans.__init__ ==> returning to <module>
KMeans.fit_predict <== called by <module>
    KMeans.fit <== called by KMeans.fit_predict
        KMeans._check_fit_data <== called by KMeans.fit
        KMeans._check_fit_data ==> returning to KMeans.fit
    KMeans.fit ==> returning to KMeans.fit_predict
KMeans.fit_predict ==> returning to <module>
```

`MiniBatchKMeans` is a subclass of `KMeans` so that class is decorated too:

```
>>> mbk = MiniBatchKMeans(init='k-means++', n_clusters=2, batch_size=45,
...                       n_init=10, max_no_improvement=10)
MiniBatchKMeans.__init__ <== called by <module>
    KMeans.__init__ <== called by MiniBatchKMeans.__init__
    KMeans.__init__ ==> returning to MiniBatchKMeans.__init__
MiniBatchKMeans.__init__ ==> returning to <module>
```

Now let's call `MiniBatchKMeans.fit`:

```
>>> mbk.fit(X)
MiniBatchKMeans.fit <== called by <module>
    MiniBatchKMeans._labels_inertia_minibatch <== called by MiniBatchKMeans.fit
    MiniBatchKMeans._labels_inertia_minibatch ==> returning to MiniBatchKMeans.fit
MiniBatchKMeans.fit ==> returning to <module>
MiniBatchKMeans(batch_size=45, compute_labels=True, init='k-means++',
```

(continues on next page)

(continued from previous page)

```
        init_size=None, max_iter=100, max_no_improvement=10, n_clusters=2,
        n_init=10, random_state=None, reassignment_ratio=0.01, tol=0.0,
        verbose=0)
```

To view arguments as well (and trigger more output), change setting to `log_args=True` and use `override=True`. Here, we call `log_calls.decorate_class` for class `KMeans` with the parameter `decorate_subclasses=True`, which is equivalent to calling `log_calls.decorate_hierarchy`:

```
>>> log_calls.decorate_class(KMeans, decorate_subclasses=True,
...                          log_args=True, args_sep='\\n',
...                          override=True)
>>> mbk.fit(X)
MiniBatchKMeans.fit <== called by <module>
    arguments:
        self=MiniBatchKMeans(batch_size=45, compute_labels=True, init='k-means++',
        init_size=None, max_iter=100, max_no_improvement=10, n_clusters=2,
        n_init=10, random_state=None, reassignment_ratio=0.01, tol=0.0,
        verbose=0)
        X=array([[ -5.19811282e+00,   6.41869316e-01],
       [ -5.75229538e+00,   4.18627111e-01],
       [ -1.08448984e+01,  -7.55352273e+00],
       ...,
       [  1.36105255e+00,  -9.07491863e-01],
       [ -3.54141108e-01,   7.12241630e-01],
       [  1.88577252e+00,   1.41185693e-03]])
    defaults:
        y=None
    MiniBatchKMeans._labels_inertia_minibatch <== called by MiniBatchKMeans.fit
        arguments:
            self=MiniBatchKMeans(batch_size=45, compute_labels=True, init='k-means++',
            init_size=None, max_iter=100, max_no_improvement=10, n_clusters=2,
            n_init=10, random_state=None, reassignment_ratio=0.01, tol=0.0,
            verbose=0)
            X=array([[ -5.19811282e+00,   6.41869316e-01],
           [ -5.75229538e+00,   4.18627111e-01],
           [ -1.08448984e+01,  -7.55352273e+00],
           ...,
           [  1.36105255e+00,  -9.07491863e-01],
           [ -3.54141108e-01,   7.12241630e-01],
           [  1.88577252e+00,   1.41185693e-03]])
    MiniBatchKMeans._labels_inertia_minibatch ==> returning to MiniBatchKMeans.fit
MiniBatchKMeans.fit ==> returning to <module>
MiniBatchKMeans(batch_size=45, compute_labels=True, init='k-means++',
        init_size=None, max_iter=100, max_no_improvement=10, n_clusters=2,
        n_init=10, random_state=None, reassignment_ratio=0.01, tol=0.0,
        verbose=0)
```

Note: the ellipses in the values of the `numpy` array `X` are produced by its `repr`.

1.12 Dynamic Control of Settings

Sometimes, you'll need or want to change a *log_calls* setting for a decorated callable on the fly. The major impediment to doing so is that the values of the *log_calls* parameters are set once the definition of the decorated callable is interpreted. Those values are established once and for all when the Python interpreter processes the definition.

1.12.1 The problem, and *log_calls* solutions

Even if a variable is used as a parameter value, its value at the time Python processes the definition is "frozen" for the created callable object. What gets stored is not the variable, but its value. Subsequently changing the value of the variable will *not* affect the behavior of the decorator.

For example, suppose `DEBUG` is a module-level variable initialized to `False`:

```
>>> DEBUG = False
```

and you use this code:

```
>>> @log_calls(enabled=DEBUG)
... def foo(**kwargs): pass
>>> foo()       # No log_calls output: DEBUG is False
```

If later you set `DEBUG = True` and call `foo`, that call still won't be logged, because the `enabled` setting of `foo` is bound to the original *value* of `DEBUG`, established when the definition was processed:

```
>>> DEBUG = True
>>> foo()       # Still no log_calls output
```

This is simply how default values of keyword parameters work in Python.

log_calls provides *three* ways to overcome this limitation and dynamically control the settings of a decorated callable:

- the `decorate_*` classmethods, described in the previous chapter *Bulk (Re)Decoration, (Re)Decorating Imports*,
- the `log_calls_settings` attribute, described in this chapter, which provides a mapping interface and an attribute-based interface to settings, and
- *indirect values*, as described in the next chapter *Indirect Values of Keyword Parameters*.

1.12.2 The `log_calls_settings` attribute — the *settings* API

log_calls adds an attribute `log_calls_settings` to the wrapper of a decorated callable, through which you can access the settings for that callable. This attribute is an object that lets you read and write the settings of the callable via a mapping (`dict`-like) interface, and equivalently, via attributes of the object. The mapping keys and the attribute names are simply the *log_calls* settings keywords. `log_calls_settings` also implements many of the standard `dict` methods for interacting with the settings in familiar ways.

The mapping interface and the attribute interface to settings

Once you've decorated a callable with *log_calls*,

```
>>> @log_calls()
... def f(*args, **kwargs):
...     return 91
```

you can access and change its settings via the `log_calls_settings` attribute of the decorated callable, which behaves like a dictionary. You can read and write settings using the *log_calls* keywords as keys:

```
>>> f.log_calls_settings['enabled']
True
>>> f.log_calls_settings['enabled'] = False
>>> _ = f()                    # no output (not even 91, because of "_ = ")
>>> f.log_calls_settings['enabled']
False
>>> f.log_calls_settings['log_retval']
False
>>> f.log_calls_settings['log_retval'] = True
>>> f.log_calls_settings['log_elapsed']
False
>>> f.log_calls_settings['log_elapsed'] = True
```

You can also use the same keywords as attributes of `log_calls_settings` instead of as keys to the mapping interface — they're completely equivalent:

```
>>> f.log_calls_settings.log_elapsed
True
>>> f.log_calls_settings.log_call_numbers
False
>>> f.log_calls_settings.log_call_numbers = True
>>> f.log_calls_settings.enabled = True     # turn it back on!
>>> _ = f()
f [1] <== called by <module>
    arguments: <none>
    f [1] return value: 91
    elapsed time: ... [secs], process time: ... [secs]
f [1] ==> returning to <module>
```

```
>>> f.log_calls_settings.log_args = False
>>> f.log_calls_settings.log_elapsed = False
>>> f.log_calls_settings.log_retval = False
>>> _ = f()
f [2] <== called by <module>
f [2] ==> returning to <module>
```

`log_calls_settings` has a length `len(log_calls_settings)`; its keys and `items()` can be iterated through; you can use `in` to test for key membership; and it has an `update()` method. As with an ordinary dictionary, attempting to access a nonexistent setting raises `KeyError`. Unlike an ordinary dictionary, you can't add new keys – the `log_calls_settings` dictionary is closed to new members, and attempts to add one will also raise `KeyError`.

The `update()`, `as_dict()`, and `as_OD()` methods

The `update()` method of the `log_calls_settings` object lets you update several settings at once:

```
>>> f.log_calls_settings.update(
...     log_args=True, log_elapsed=False, log_call_numbers=False,
...     log_retval=False)
>>> _ = f()
f <== called by <module>
    arguments: <none>
f ==> returning to <module>
```

You can retrieve the entire collection of settings as a `dict` using `as_dict()`, and as an `OrderedDict` using `as_OD()`. Either can serve as a snapshot of the settings, so that you can change settings temporarily, use the new settings, and then use `update()` to restore settings from the snapshot. in addition to taking keyword arguments, as shown above, `update()` can take one or more dicts – in particular, a dictionary retrieved from one of the `as_*` methods:

`wrapper.log_calls_settings.`**`update`**(**dicts*, ***d_settings*) → None

Update the settings from all dicts in `dicts`, in order, and then from `d_settings`. Allow but ignore attempts to write to immutable keys (`max_history`). This permits the user to retrieve a copy of the settings with `as_dict()` or `as_OD()`, obtaining a dictionary which will contain items for immutable settings too; make changes to settings and use them; then restore the original settings by passing the retrieved dictionary to `update()`.

Parameters

- **`dicts`** – a sequence of dicts containing setting keywords and values
- **`d_settings`** – additional settings and values

Example

This example illustrates the use-case described above.

First, retrieve settings (here, as an `OrderedDict` because those are more *doctest*-friendly, but in "real life" using `as_dict()` suffices):

```
>>> od = f.log_calls_settings.as_OD()
>>> od
OrderedDict([('enabled', True),             ('args_sep', ', '),
             ('log_args', True),            ('log_retval', False),
             ('log_elapsed', False),        ('log_exit', True),
             ('indent', True),              ('log_call_numbers', False),
             ('prefix', ''),                ('file', None),
             ('logger', None),              ('loglevel', 10),
             ('mute', False),
             ('record_history', False),     ('max_history', 0)])
```

Change settings temporarily:

```
>>> f.log_calls_settings.update(
...     log_args=False, log_elapsed=True, log_call_numbers=True,
...     log_retval=True)
```

Use the new settings for `f`:

```
>>> _ = f()
f [4] <== called by <module>
    f [4] return value: 91
    elapsed time: ... [secs], process time: ... [secs]
f [4] ==> returning to <module>
```

Now restore original settings, this time passing the retrieved settings dictionary rather than keywords (we *could* pass `**od`, but that's unnecessary and a pointless expense):

```
>>> f.log_calls_settings.update(od)
>>> od == f.log_calls_settings.as_OD()
True
```

1.13 Indirect Values of Keyword Parameters

Most parameters of *log_calls* (all *settings* paramaters except `prefix` and `max_history`) can take two kinds of values: *direct* and *indirect*, which you can think of as *static* and *dynamic* respectively. Direct/static values are actual values, such as those computed when the definition of a decorated callable is interpreted, e.g. `enabled=True`, `args_sep=" / "`. As discussed in the previous chapter, *Dynamic Control of Settings*, the values of parameters are set once and for all when the Python interpreter creates a callable object from the source code of a decorated function or method. Even if you use a variable as the value of a setting, subsequently changing the variable's value has no effect on the decorator's setting.

log_calls provides yet another way to overcome this limitation, in addition to those described in the previous two chapters: *indirect values*.

Caveat

Using this capability is more intrusive than the approaches to dynamically changing settings already discussed: it introduces more "debug-only" code which you'll have to ensure doesn't run in production. As such, it's less appealing. However, it has its place, in demos and producing documentation.

1.13.1 Definition and basic examples

log_calls lets you specify any "setting" parameter except *prefix* or *max_history* with one level of indirection, by using *indirect values*:

indirect value of a *setting* parameter A string that names a keyword parameter of a decorated callable. When the callable is called, the value of *that* keyword argument is used as the value of the setting.

To specify an indirect value for a parameter whose normal values are (or can be) `str``s (this applies only to ``args_sep` and `logger`, at present), append an `'='` to the value. For consistency, any indirect value can end in a trailing `'='`, which is stripped. Thus, `enabled='enable_='` indicates an indirect value *to be supplied* with the keyword `enable_`.

Explicit indirect values

An indirect value can be an explicit keyword argument present in the signature of the callable:

```
>>> @log_calls(enabled='enable_')
... def f(x, y, enable_=False): pass
```

Thus, calling `f` above without passing a value for `enable_` uses the default value `False` of `enable_`, and the call gives no output:

```
>>> f(1, 2)
```

Supplying a value `True` for `enable_` does give *log_calls* output:

```
>>> f(3, 4, enable_=True)     # output:
f <== called by <module>
    arguments: x=3, y=4, enable_=True
f ==> returning to <module>
```

Implicit indirect values

An indirect value doesn't have to be present in the signature of a decorated callable. It can be an implicit keyword argument that ends up in `**kwargs`:

```
>>> @log_calls(args_sep_=', ')       # same as log_calls default
... def g(x, y, **kwargs): pass
```

When the decorated callable is called, the arguments passed by keyword, and the decorated callable's explicit keyword parameters with default values, are both searched for the named parameter; if it is found and of the correct type, *its* value is used; otherwise a default value is used.

Here, the value of the `args_sep` setting will be the default value given for `args_sep_`:

```
>>> g(1, 2)
g <== called by <module>
    arguments: x=1, y=2
g ==> returning to <module>
```

whereas here, the `args_sep` value used will be `' $ '`:

```
>>> g(3, 4, args_sep_=' $ ')
g <== called by <module>
    arguments: x=3 $ y=4 $ **kwargs={'args_sep_': ' $ '}
g ==> returning to <module>
```

Note: If an indirect value is specified for `enabled` and it is "not found", then the default value of `False` is used. For example:

```
>>> @log_calls(enabled='enable_')
... def h(**kwargs): pass
```

Here, the indirect value `enable_` has no default value — there is no default indirect value for `enabled`. In this special case only, the enabled setting will be `False` if no value is supplied for `enable_` in a call to `h`:

```
>>> h()              # no output
>>> h(enable_=True)  # output:
h <== called by <module>
    arguments: **kwargs={'enable_': True}
h ==> returning to <module>
```

1.13.2 Indirect values in settings dicts and files

In a settings file, the value of a keyword is treated as an indirect value if it's enclosed in (single or double) quotes and its last non-quote character is '='. For example:

```
``file='file_='``
```

Of course, indirect values can be used in settings dicts as well, and there, only indirect values of `args_sep` and `logger` require a trailing =.

1.13.3 Using `log_calls_settings` to set indirect values

Similarly, it's perfectly legitimate to assign an indirect value to a setting via `log_calls_settings`:

```
>>> @log_calls(enabled=False)
... def g(*args, **kwargs):
...     return sum(args)
>>> g(0, 1, 2)              # no log_calls output
3
>>> g.log_calls_settings.enabled = 'enable_log_calls='
>>> g(1, 2, 3, enable_log_calls=True)
g <== called by <module>
    arguments: *args=(1, 2, 3), **kwargs={'enable_log_calls': True}
g ==> returning to <module>
6
```

1.13.4 Controlling format 'from above'

This indirection mechanism allows a caller to control the appearance of logged calls lower in the call chain, provided all decorated callables use the same indirect parameter keywords.

In the next example, the separator value supplied to `g` by keyword argument propagates to `f`. Note that the arguments `42` and `99` end up in `g`'s positional *varargs* tuple.

```
>>> @log_calls(args_sep='sep=')
... def f(a, b, c, **kwargs): pass
>>> @log_calls(args_sep='sep=')
... def g(a, b, c, *g_args, **g_kwargs):
...     f(a, b, c, **g_kwargs)
>>> g(1,2,3, 42, 99, sep='\\n')
g <== called by <module>
    arguments:
        a=1
        b=2
        c=3
```

(continues on next page)

(continued from previous page)

```
        *g_args=(42, 99)
        **g_kwargs={'sep': '\\n'}
    f <== called by g
        arguments:
            a=1
            b=2
            c=3
            **kwargs={'sep': '\\n'}
    f ==> returning to g
g ==> returning to <module>
```

1.13.5 Paradigms for handling keyword parameters

Several uses of "indirect values" described in this section rely on multiple functions and methods treating `**kwargs` as a kind of "common area" or "bulletin board" – a central store for data of common interest. This paradigm for `**kwargs` handling, which we might call *promiscuous cooperation*, conflicts with the one usually espoused, for example in discussions about the design of composable classes which cooperatively call `super()`. In his article Python's super() considered super!, Raymond Hettinger clearly describes that approach as one in which:

> every method [`f`, say, is] cooperatively designed to accept keyword arguments
> and a keyword-arguments dictionary, to remove any arguments that it needs,
> and to forward the remaining arguments using `**kwds` [via `super().f(..., **kwds)`,
> where `...` are positional arguments], eventually leaving the dictionary empty
> for the final call in the chain.

Certainly, this condition implies that a subclass's implementation of a method should never share keywords with a parent class's implementation. But it's more stringent than that. It requires that a class's implementation of a method *never* share keywords with any implementation of that method in *any* class that might *ever* be on its mro list. Indeed, following this prescription, an implementation simply *can't* share keyword parameters: each method will "remove any [parameters] that it needs" before passing the baton via `super()` to its kinfolk further on down the mro list. In the presence of multiple inheritance, which alters a class's static mro, this might be difficult to guarantee.

This is a clear if stern approach to cooperation, one consistent with the behavior of certain "final calls in the chain" that land in core Python. For example, `object.__init__` and `type.__init__` raise an exception if they receive any `**kwargs`. (Would that they didn't: this is often a nuisance.) But the "promiscuous" paradigm of cooperation is also valid and useful, and causes no harm as long as it's clear what all cooperating parties are agreeing *to*.

1.14 Accessing Method Wrappers

1.14.1 The `get_log_calls_wrapper()` and `get_own_log_calls_wrapper()` classmethods

log_calls decorates a callable by "wrapping" it in a function (the *wrapper*) which has attributes containing data about the callable: `log_calls_settings`, containing settings, and `stats`, containing statistics. Access to these attributes requires access to the callable's wrapper.

It's straightforward to access the wrapper of a decorated global function `f`: after decoration, `f` refers to the wrapper. For methods and properties, however, the various kinds of methods and the two ways of defining properties require different navigation paths to the wrapper. *log_calls* hides this complexity, providing uniform access to the wrappers of methods and properties.

classmethod decorated_class.**get_log_calls_wrapper**(*fname: str*)

Classmethod of a decorated class. Call this on a decorated class or an instance thereof to access the wrapper of the callable named `fname`, in order to access the *log_calls*-added attributes for `fname`.

Parameters **fname** – name of a method (instance method, staticmethod or classmethod), or the name of a property (treated as denoting the getter), or the name of a property concatenated with '*.getter*', '*.setter*' or '*.deleter*'.

Note: If a property is defined using the `property` function, as in

```
propx = property(getx, setx, delx),
```

where `getx`, `setx`, `delx` are methods of a class (or `None`), then each individual property can be referred to in two ways:

- via the name of the method, eg. `setx`, or
- via `propx.`*qualifier*, where *qualifier* is one of `setter`, `getter`, `deleter`, as appropriate (so `propx.setter` also refers to `setx`)

Thus you can use either `dc.log_calls_wrapper('setx')` or `dc.log_calls_wrapper('propx.setter')` where `dc` is a decorated class or an instance thereof.

Raises `TypeError` if `fname` is not a `str`; ValueError if `fname` isn't as described above or isn't in the `__dict__` of *decorated_class*.

Returns wrapper of `fname` if `fname` is decorated, `None` otherwise.

classmethod decorated_class.**get_own_log_calls_wrapper**()

Classmethod of a decorated class. Call from *within* a method or property of a decorated class. Typically called on `self` from within instance methods, on `cls` from within classmethods, and on the explicitly named enclosing, decorated class `decorated_class` from within staticmethods.

Raises `ValueError` if caller is not decorated.

Returns the wrapper of the caller (a function), so that the caller can access its own *log_calls* attributes.

1.15 Call History and Statistics

Unless it's *bypassed*, *log_calls* always collects at least a few basic statistics about each call to a decorated callable. It can collect the entire history of calls to a function or method if asked to (using the *record_history* setting). The statistics and history are accessible via the `stats` attribute that *log_calls* adds to the wrapper of a decorated callable.

The two settings parameters we haven't yet discussed govern the recording and retention of a decorated callable's call history.

1.15.1 The `record_history` parameter (default: `False`)

When the `record_history` setting is true for a decorated callable, *log_calls* accumulates a sequence of records holding the details of each *logged* call to the callable:

> A **logged call** to a decorated callable is one that occurs when the callable's `enabled` setting is true.

That history is accessible via attributes of the `stats` object.

Let's define a function `f` with `record_history` set to true:

```
>>> @log_calls(record_history=True, log_call_numbers=True, log_exit=False)
... def f(a, *args, x=1, **kwargs): pass
```

In the next few sections, we'll call this function, manipulate its settings, and examine its statistics and history.

1.15.2 The `max_history` parameter (default: `0`)

The `max_history` parameter bounds the number of call history records retained in a decorated callable's recorded call history. If this value is 0 or negative, unboundedly many records are retained (unless or until you change the `record_history` setting to false, or call the `stats.clear_history()` method). If the value of `max_history` is > 0, call history operates as a least-recently-used cache: *log_calls* will retain at most that many records, discarding the oldest record to make room for a new ones if the history is at capacity.

You cannot change `max_history` using the mapping interface or the attribute of the same name; attempts to do so raise `ValueError`. The only way to change its value is with the `stats.clear_history(max_history=0)` method, discussed *below*.

1.15.3 The `stats` attribute and *its* attributes

The `stats` attribute of a decorated callable is an object that provides read-only statistics and data about the calls to a decorated callable:

- *stats.num_calls_logged*
- *stats.num_calls_total*
- *stats.elapsed_secs_logged*
- *stats.process_secs_logged*
- *stats.history*
- *stats.history_as_csv*
- *stats.history_as_DataFrame*

The first four of these don't depend on the `record_history` setting at all. The last three values, `stats.history*`, are empty unless `record_history` is or has been true.

The `stats` attribute also provides one method, *stats.clear_history()*.

Let's call the above-defined function `f` twice:

```
>>> f(0)
f [1] <== called by <module>
    arguments: a=0
    defaults:  x=1
>>> f(1, 100, 101, x=1000, y=1001)
```

(continues on next page)

(continued from previous page)

```
f [2] <== called by <module>
    arguments: a=1, *args=(100, 101), x=1000, **kwargs={'y': 1001}
```

and explore its `stats`.

The `stats.num_calls_logged` attribute

The `stats.num_calls_logged` attribute holds the number of the most recent logged call to a decorated callable. Thus, `f.stats.num_calls_logged` will equal 2:

```
>>> f.stats.num_calls_logged
2
```

This counter is incremented on each logged call to the callable, even if its `log_call_numbers` setting is false.

The `stats.num_calls_total` attribute

The `stats.num_calls_total` attribute holds the *total* number of calls to a decorated callable. This counter is incremented even when logging is disabled for a callable (its `enabled` setting is `False`, i.e. `0`), but *not* when it's *bypassed*.

To illustrate, let's now *disable* logging for `f` and call it 3 more times:

```
>>> f.log_calls_settings.enabled = False
>>> for i in range(3): f(i)
```

Now `f.stats.num_calls_total` will equal 5, but `f.stats.num_calls_logged` will still equal 2:

```
>>> f.stats.num_calls_total
5
>>> f.stats.num_calls_logged
2
```

Finally, let's re-enable logging for `f` and call it again. The displayed call number will be the number of the *logged* call, 3, the same value as `f.stats.num_calls_logged` after the call:

```
>>> f.log_calls_settings.enabled = True
>>> f(10, 20, z=5000)
f [3] <== called by <module>
    arguments: a=10, *args=(20,), **kwargs={'z': 5000}
    defaults:  x=1
```

```
>>> f.stats.num_calls_total
6
>>> f.stats.num_calls_logged
3
```

The `stats.elapsed_secs_logged` attribute

The `stats.elapsed_secs_logged` attribute holds the sum of the elapsed times of all *logged* calls to a decorated callable, in seconds. Here's its value for the three logged calls to `f` above (this doctest is actually `+SKIP`ped):

```
>>> f.stats.elapsed_secs_logged
6.67572021484375e-06
```

The `stats.process_secs_logged` attribute

The `stats.process_secs_logged` attribute holds the sum of the process times of all *logged* calls to a decorated callable, in seconds. Here's its value for the three logged calls to `f` above (this doctest is actually `+SKIP`ped):

```
>>> f.stats.process_secs_logged
1.1000000000038757e-05
```

The `stats.history` attribute

The `stats.history` attribute of a decorated callable provides the call history of logged calls as a tuple of records. Each record is a `namedtuple` of type `CallRecord`, whose fields are those shown in the following example. Here's `f`'s call history, output reformatted for readability:

```
>>> print('\\n'.join(map(str, f.stats.history)))
CallRecord(call_num=1, argnames=['a'], argvals=(0,), varargs=(),
                       explicit_kwargs=OrderedDict(),
                       defaulted_kwargs=OrderedDict([('x', 1)]), implicit_kwargs={},
                       retval=None,
                       elapsed_secs=3.0049995984882116e-06,
                       process_secs=2.9999999999752447e-06,
                       timestamp='10/28/14 15:56:13.733763',
                       prefixed_func_name='f', caller_chain=['<module>'])
CallRecord(call_num=2, argnames=['a'], argvals=(1,), varargs=(100, 101),
                       explicit_kwargs=OrderedDict([('x', 1000)]),
                       defaulted_kwargs=OrderedDict(), implicit_kwargs={'y': 1001},
                       retval=None,
                       elapsed_secs=3.274002665420994e-06,
                       process_secs=3.0000000000030003e-06,
                       timestamp='10/28/14 15:56:13.734102',
                       prefixed_func_name='f', caller_chain=['<module>'])
CallRecord(call_num=3, argnames=['a'], argvals=(10,), varargs=(20,),
                       explicit_kwargs=OrderedDict(),
                       defaulted_kwargs=OrderedDict([('x', 1)]), implicit_kwargs={'z
↪': 5000},
                       retval=None,
                       elapsed_secs=2.8769973141606897e-06,
                       process_secs=2.9999999999752447e-06,
                       timestamp='10/28/14 15:56:13.734412',
                       prefixed_func_name='f', caller_chain=['<module>'])
```

The CSV representation, discussed next, pairs the `argnames` with their values in `argvals` (each parameter name in `argnames` become a column heading), making it more human-readable, especially when viewed in a program that presents CSVs nicely.

The `stats.history_as_csv` attribute

The value `stats.history_as_csv` attribute is a text representation in CSV format of a decorated callable's call history. You can save this string and import it into the program or tool of your choice for further analysis. (If your tool of choice is Pandas, you can use *The stats.history_as_DataFrame attribute*, discussed next, to obtain history directly in the representation you really want.)

The CSV representation breaks out each argument into its own column, throwing away information about whether an argument's value was explicitly passed or is a default.

The CSV separator is `'|'` rather than `','` because some of the fields – `args`, `kwargs` and `caller_chain` – use commas intrinsically. Let's examine `history_as_csv` for a function that has all of those fields nontrivially:

```
>>> @log_calls(record_history=True, log_call_numbers=True,
...            log_exit=False, log_args=False)
... def f(a, *extra_args, x=1, **kw_args): pass
>>> def g(a, *args, **kwargs):
...     f(a, *args, **kwargs)
>>> @log_calls(log_exit=False, log_args=False)
... def h(a, *args, **kwargs):
...     g(a, *args, **kwargs)
>>> h(0)
h <== called by <module>
    f [1] <== called by g <== h
>>> h(10, 17, 19, z=100)
h <== called by <module>
    f [2] <== called by g <== h
>>> h(20, 3, 4, 6, x=5, y='Yarborough', z=100)
h <== called by <module>
    f [3] <== called by g <== h
```

Here's the call history of `f` in CSV format (ellipses added for the `elapsed_secs`, `process_secs` and `timestamp` fields):

```
>>> print(f.stats.history_as_csv)
call_num|a|extra_args|x|kw_args|retval|elapsed_secs|process_secs|timestamp|prefixed_
↪fname|caller_chain
1|0|()|1|{}|None|...|...|...|'f'|['g', 'h']
2|10|(17, 19)|1|{'z': 100}|None|...|...|...|'f'|['g', 'h']
3|20|(3, 4, 6)|5|{'y': 'Yarborough', 'z': 100}|None|...|...|...|'f'|['g', 'h']
```

In tabular form,

call_num	a	extra_args	x	kw_args	retval	elapsed_secs	process_secs	timestamp	prefixed_func_name	caller_chain
1	0	()	1	{}	None	...	...	...	'f'	['g', 'h']
2	10	(17, 19)	1	{'z': 100}	None	...	...	...	'f'	['g', 'h']
3	20	(3, 4, 6)	5	{'y': 'Yarborough', 'z': 100}	None	...	...	...	'f'	['g', 'h']

As usual, *log_calls* will use whatever names you use for *varargs* parameters (here, `extra_args` and `kw_args`). Whatever the name of the `kwargs` parameter, items within that field are guaranteed to be in sorted order.

The `stats.history_as_DataFrame` attribute

The `stats.history_as_DataFrame` attribute returns the history of a decorated callable as a Pandas DataFrame, if the Pandas library is installed. This saves you the intermediate step of calling `DataFrame.from_csv` with the proper arguments (and also saves you from having to know or care what those are).

If Pandas is not installed, the value of this attribute is `None`.

The `stats.clear_history(max_history=0)` method

As you might expect, the `stats.clear_history(max_history=0)` method clears the call history of a decorated callable. In addition, it resets all running sums:

- `num_calls_total` and `num_calls_logged` are reset to `0`,
- `elapsed_secs_logged` and `process_secs_logged` are reset to `0.0`.

This method is the only way to change the value of the ``max_history`` setting, via the optional keyword parameter for which you can supply any (integer) value, by default `0`.

The function `f` has a nonempty history, as we just saw. Let's clear `f`'s history, setting `max_history` to `33`:

```
>>> f.stats.clear_history(max_history=33)
```

and check that settings and `stats` tallies are reset:

```
>>> f.log_calls_settings.max_history
33
>>> f.stats.num_calls_logged
0
>>> f.stats.num_calls_total
0
```

(continues on next page)

(continued from previous page)

```
>>> f.stats.elapsed_secs_logged
0.0
>>> f.stats.process_secs_logged
0.0
```

1.16 Call Chains

log_calls does its best to chase back along the call chain to find the first *enabled log_calls*-decorated callable on the stack. If there's no such function, it just displays the immediate caller. If there is such a function, however, it displays the entire list of functions on the stack up to and including that function when reporting calls and returns. Without this, you'd have to guess at what was called in between calls to functions decorated by *log_calls*. If you specified a `prefix` or `name` for the decorated caller on the end of a call chain, *log_calls* will use the requested display name:

1.16.1 Basic examples

```
>>> @log_calls()
... def g1():
...     pass
>>> def g2():
...     g1()
>>> @log_calls(prefix='mid.')
... def g3():
...     g2()
>>> def g4():
...     g3()
>>> @log_calls()
... def g5():
...     g4()
>>> g5()
g5 <== called by <module>
    mid.g3 <== called by g4 <== g5
        g1 <== called by g2 <== mid.g3
        g1 ==> returning to g2 ==> mid.g3
    mid.g3 ==> returning to g4 ==> g5
g5 ==> returning to <module>
```

In the next example, `g` is *log_calls*-decorated but logging is disabled, so the reported call chain for `f` stops at its immediate caller:

```
>>> @log_calls()
... def f(): pass
>>> def not_decorated(): f()
>>> @log_calls(enabled=False)
... def g(): not_decorated()
>>> g()
f <== called by not_decorated
f ==> returning to not_decorated
```

Elaborating on the previous example, here are longer call chains with an intermediate decorated function that has logging disabled:

```
>>> @log_calls()
... def e(): pass
>>> def not_decorated_call_e(): e()
>>> @log_calls()
... def f(): not_decorated_call_e()
>>> def not_decorated_call_f(): f()
>>> @log_calls(enabled=False)
... def g(): not_decorated_call_f()
>>> @log_calls()
... def h(): g()
>>> h()
h <== called by <module>
    f <== called by not_decorated_call_f <== g <== h
        e <== called by not_decorated_call_e <== f
        e ==> returning to not_decorated_call_e ==> f
    f ==> returning to not_decorated_call_f ==> g ==> h
h ==> returning to <module>
```

log_calls chases back to the nearest *enabled* decorated callable, so that there aren't gaps between call chains.

1.16.2 Call chains and inner functions

When chasing back along the stack, *log_calls* also detects inner functions that it has decorated:

```
>>> @log_calls()
... def h0(z):
...     pass
>>> def h1(x):
...     @log_calls(name='h1_inner')
...     def h1_inner(y):
...         h0(x*y)
...     return h1_inner
>>> def h2():
...     h1(2)(3)
>>> def h3():
...     h2()
>>> def h4():
...     @log_calls(name='h4_inner')
...     def h4_inner():
...         h3()
...     return h4_inner
>>> @log_calls()
... def h5():
...     h4()()
>>> h5()
h5 <== called by <module>
    h4_inner <== called by h5
        h1_inner <== called by h2 <== h3 <== h4_inner
            arguments: y=3
            h0 <== called by h1_inner
                arguments: z=6
            h0 ==> returning to h1_inner
        h1_inner ==> returning to h2 ==> h3 ==> h4_inner
    h4_inner ==> returning to h5
h5 ==> returning to <module>
```

... even when the inner function is called from within the outer function it's defined in:

```
>>> @log_calls()
... def j0():
...     pass
>>> def j1():
...     j0()
>>> def j2():
...     @log_calls()
...     def j2_inner():
...         j1()
...     j2_inner()
>>> @log_calls()
... def j3():
...     j2()
>>> j3()
j3 <== called by <module>
    j2.<locals>.j2_inner <== called by j2 <== j3
        j0 <== called by j1 <== j2.<locals>.j2_inner
        j0 ==> returning to j1 ==> j2.<locals>.j2_inner
    j2.<locals>.j2_inner ==> returning to j2 ==> j3
j3 ==> returning to <module>
```

1.16.3 Call chains and *log_call_numbers*

If a decorated callable is enabled and has `log_call_numbers` set to true, then its call numbers will be displayed in call chains:

```
>>> @log_calls()
... def f(): pass
>>> def not_decorated(): f()
>>> @log_calls(log_call_numbers=True)
... def g(): not_decorated()
>>> g()
g [1] <== called by <module>
    f <== called by not_decorated <== g [1]
    f ==> returning to not_decorated ==> g [1]
g [1] ==> returning to <module>
```

Also apropos is the example *Indentation and call numbers with recursion*.

1.17 Further Examples and Use Cases

This chapter collects several longer examples that demonstrate techniques and not just individual features of *log_calls*.

1.17.1 Using `enabled` as a level of verbosity

Sometimes it's desirable for a function to print or log debugging messages as it executes. It's the oldest form of debugging! The `enabled` parameter is in fact an `int`, not just a `bool`. Instead of giving it a simple `bool` value, you can use a nonnegative `int` and treat it as a verbosity level:

```
>>> DEBUG_MSG_BASIC = 1
>>> DEBUG_MSG_VERBOSE = 2
>>> DEBUG_MSG_MOREVERBOSE = 3  # etc.
>>> @log_calls(enabled='debuglevel')
... def do_stuff_with_commentary(*args, debuglevel=0):
...     if debuglevel >= DEBUG_MSG_VERBOSE:
...         print("*** extra debugging info ***")
```

No output:

```
>>> do_stuff_with_commentary()
```

Only *log_calls* output:

```
>>> do_stuff_with_commentary(debuglevel=DEBUG_MSG_BASIC)
do_stuff_with_commentary <== called by <module>
    arguments: debuglevel=1
do_stuff_with_commentary ==> returning to <module>
```

log_calls output plus the function's debugging reportage:

```
>>> do_stuff_with_commentary(debuglevel=DEBUG_MSG_VERBOSE)
do_stuff_with_commentary <== called by <module>
    arguments: debuglevel=2
*** extra debugging info ***
do_stuff_with_commentary ==> returning to <module>
```

The metaclass example later in this chapter also uses this technique, and writes its messages with the *log_calls.print()* method.

1.17.2 Indentation and call numbers with recursion

Setting `log_call_numbers` to true is especially useful in with recursive, mutually recursive and reentrant callables. In this example, the function `depth` computes the *depth* of a dictionary (a non-dict has depth = 0, and a dict has depth = 1 + the max of the depths of its values):

```
>>> from collections import OrderedDict
>>> @log_calls(log_call_numbers=True, log_retval=True)
>>> def depth(d, key=None):
...     """Middle line (elif) is needed only because
...     max(empty_sequence) raises ValueError
...     (whereas returning 0 would be sensible and even expected)
...     """
...     if not isinstance(d, dict): return 0    # base case
```

(continues on next page)

(continued from previous page)

```
...     elif not d:                 return 1
...     else:                       return max(map(depth, d.values(), d.keys())) + 1
```

Now we call `depth` with a nested `OrderedDict`:

```
>>> depth(
...     OrderedDict(
...         (('a', 0),
...          ('b', OrderedDict( (('c1', 10), ('c2', 11)) )),
...          ('c', 'text'))
...     )
... )
depth [1] <== called by <module>
    arguments: d=OrderedDict([('a', 0), ('b', OrderedDict([('c1', 10), ('c2', 11)])),␣
↪('c', 'text')])
    defaults:  key=None
    depth [2] <== called by depth [1]
        arguments: d=0, key='a'
        depth [2] return value: 0
    depth [2] ==> returning to depth [1]
    depth [3] <== called by depth [1]
        arguments: d=OrderedDict([('c1', 10), ('c2', 11)]), key='b'
        depth [4] <== called by depth [3]
            arguments: d=10, key='c1'
            depth [4] return value: 0
        depth [4] ==> returning to depth [3]
        depth [5] <== called by depth [3]
            arguments: d=11, key='c2'
            depth [5] return value: 0
        depth [5] ==> returning to depth [3]
        depth [3] return value: 1
    depth [3] ==> returning to depth [1]
    depth [6] <== called by depth [1]
        arguments: d='text', key='c'
        depth [6] return value: 0
    depth [6] ==> returning to depth [1]
    depth [1] return value: 2
depth [1] ==> returning to <module>
2
```

The three calls `depth [2]`, `depth [3]`, and `depth [6]` handle the three items of the dictionary passed to `depth [1]`; they return `0`, `1`, and `0` respectively. Finally `depth [1]` returns 1 plus the `max` of those values.

Note: The optional `key` parameter is for instructional purposes, so you can see the key that's paired with the value of `d` in the caller's dictionary. Typically the signature of this function would be just `def depth(d)`, and the recursive case would return `1 + max(map(depth, d.values()))`.

1.17.3 Using a logger with multiple handlers that have different loglevels

First let's set up a logger with a console handler that writes to `stdout`:

```
>>> import logging
>>> import sys
>>> ch = logging.StreamHandler(stream=sys.stdout)
>>> c_formatter = logging.Formatter('%(levelname)s:%(name)s:%(message)s')
>>> ch.setFormatter(c_formatter)
>>> logger = logging.getLogger('mylogger')
>>> logger.addHandler(ch)
>>> logger.setLevel(logging.DEBUG)
```

Now let's add another handler, also sent to `stdout` for the sake of the example but best thought of as writing to a log file. We'll set up the existing console handler with level `INFO`, and the "file" handler with level `DEBUG` – a typical setup: you want to log all details to the file, but you only want to write more important messages to the console.

```
>>> fh = logging.StreamHandler(stream=sys.stdout)
>>> f_formatter = logging.Formatter('[FILE] %(levelname)8s:%(name)s: %(message)s')
>>> fh.setFormatter(f_formatter)
>>> fh.setLevel(logging.DEBUG)
>>> logger.addHandler(fh)
>>> ch.setLevel(logging.INFO)
```

Suppose we have two functions: one that's lower-level/often-called, and another that's higher-level/infrequently called. It's appropriate to give the infrequently called function a higher `loglevel`:

```
>>> @log_calls(logger=logger, loglevel=logging.DEBUG)
... def popular(): pass
>>> @log_calls(logger=logger, loglevel=logging.INFO)
... def infrequent(): popular()
```

Set the log level to `logging.DEBUG` – the console handler logs calls only for `infrequent`, but the "file" handler logs calls for both functions:

```
>>> logger.setLevel(logging.DEBUG)
>>> infrequent()
INFO:mylogger:infrequent <== called by <module>
[FILE]     INFO:mylogger: infrequent <== called by <module>
[FILE]    DEBUG:mylogger: popular <== called by infrequent
[FILE]    DEBUG:mylogger: popular ==> returning to infrequent
INFO:mylogger:infrequent ==> returning to <module>
[FILE]     INFO:mylogger: infrequent ==> returning to <module>
```

Now set log level to `logging.INFO` – both handlers logs calls only for `infrequent`:

```
>>> logger.setLevel(logging.INFO)
>>> infrequent()
INFO:mylogger:infrequent <== called by <module>
[FILE]     INFO:mylogger: infrequent <== called by <module>
INFO:mylogger:infrequent ==> returning to <module>
[FILE]     INFO:mylogger: infrequent ==> returning to <module>
```

1.17.4 A metaclass example

This example demonstrates a few techniques:

- writing debug messages with `log_calls.print()`, which handles global indentation for you;
- use of `enabled` as an integer level of verbosity.

The following class `A_meta` will serve as the metaclass for classes defined subsequently:

```
>>> # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
>>> # A_meta, a metaclass
>>> # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
>>> from collections import OrderedDict
>>> separator = '\n'    # default ', ' gives rather long lines
```

```
>>> A_DBG_NONE = 0
>>> A_DBG_BASIC = 1
>>> A_DBG_INTERNAL = 2
```

```
>>> @log_calls(args_sep=separator, enabled='A_debug=')
... class A_meta(type):
...     @classmethod
...     @log_calls(log_retval=True)
...     def __prepare__(mcs, cls_name, bases, **kwargs):
...         super_dict = super().__prepare__(cls_name, bases, **kwargs)
...         A_debug = kwargs.pop('A_debug', A_DBG_NONE)
...         if A_debug >= A_DBG_INTERNAL:
...             log_calls.print("    mro =", mcs.__mro__)
...             log_calls.print("    dict from super() = %r" % super_dict)
...         super_dict = OrderedDict(super_dict)
...         super_dict['key-from-__prepare__'] = 1729
...         return super_dict
...
...     def __new__(mcs, cls_name, bases, cls_members: dict, **kwargs):
...         cls_members['key-from-__new__'] = "No, Hardy!"
...         A_debug = kwargs.pop('A_debug', A_DBG_NONE)
...         if A_debug >= A_DBG_INTERNAL:
...             log_calls.print("    calling super() with cls_members =", cls_members)
...         return super().__new__(mcs, cls_name, bases, cls_members, **kwargs)
...
...     def __init__(cls, cls_name, bases, cls_members: dict, **kwargs):
...         A_debug = kwargs.pop('A_debug', A_DBG_NONE)
...         if A_debug >= A_DBG_INTERNAL:
...             log_calls.print("    cls.__mro__:", cls.__mro__)
...             log_calls.print("    type(cls).__mro__[1] =", type(cls).__mro__[1])
...         try:
...             super().__init__(cls_name, bases, cls_members, **kwargs)
...         except TypeError as e:
...             # call type.__init__
...             if A_debug >= A_DBG_INTERNAL:
...                 log_calls.print("    calling type.__init__ with no kwargs")
...             type.__init__(cls, cls_name, bases, cls_members)
```

The class `A_meta` is a metaclass: it derives from `type`, and defines (overrides) methods `__prepare__`, `__new__` and `__init__`. All of these *log_calls*-decorated methods awrite their messages using the indent-aware method *log_calls.print()*.

All of `A_meta`'s methods look for an implicit keyword parameter `A_debug`, used as the indirect value of the *log_calls*

parameter `enabled`. The methods treat its value as an integer verbosity level: they write extra messages when the value of `A_debug` is at least `A_DBG_INTERNAL`.

Rather than make `A_debug` an explicit keyword parameter of the metaclass methods, as in:

```
def __prepare__(mcs, cls_name, bases, *, A_debug=0, **kwargs):
```

instead we have left their signatures agnostic. If `A_debug` is passed by a class definition (as below), the methods use the passed value, and remove `A_debug` from `kwargs`; otherwise they use a default value `A_DBG_NONE`, which is less than their threshold value for writing debug messages.

When we include `A_debug=A_DBG_INTERNAL` as a keyword argument to a class that uses `A_meta` as its metaclass, that argument gets passed to all of `A_meta`'s methods, so not only will calls to the metaclass methods be logged, but those methods will also print extra debugging information:

```
>>> class A(metaclass=A_meta, A_debug=A_DBG_INTERNAL):
...     pass
A_meta.__prepare__ <== called by <module>
    arguments:
        mcs=<class '__main__.A_meta'>
        cls_name='A'
        bases=()
        **kwargs={'A_debug': 2}
        mro = (<class '__main__.A_meta'>, <class 'type'>, <class 'object'>)
        dict from super() = {}
    A_meta.__prepare__ return value: OrderedDict([('key-from-__prepare__', 1729)])
A_meta.__prepare__ ==> returning to <module>
A_meta.__new__ <== called by <module>
    arguments:
        mcs=<class '__main__.A_meta'>
        cls_name='A'
        bases=()
        cls_members=OrderedDict([('key-from-__prepare__', 1729),
                                 ('__module__', '__main__'),
                                 ('__qualname__', 'A')])
        **kwargs={'A_debug': 2}
        calling super() with cls_members = OrderedDict([('key-from-__prepare__',␣
↪1729),
                                                        ('__module__', '__main__'),
                                                        ('__qualname__', 'A'),
                                                        ('key-from-__new__', 'No,␣
↪Hardy!')])
A_meta.__new__ ==> returning to <module>
A_meta.__init__ <== called by <module>
    arguments:
        cls=<class '__main__.A'>
        cls_name='A'
        bases=()
        cls_members=OrderedDict([('key-from-__prepare__', 1729),
                                 ('__module__', '__main__'),
                                 ('__qualname__', 'A'),
                                 ('key-from-__new__', 'No, Hardy!')])
        **kwargs={'A_debug': 2}
        cls.__mro__: (<class '__main__.A'>, <class 'object'>)
        type(cls).__mro__[1] = <class 'type'>
A_meta.__init__ ==> returning to <module>
```

If we had passed `A_debug=A_DBG_BASIC`, then only *log_calls* output would have been printed: the metaclass methods would not have printed their extra debugging statements.

If we pass `A_debug=0` (or omit the parameter), we get no printed output at all, either from *log_calls* or from `A_meta`'s methods:

```
>>> class AA(metaclass=A_meta, A_debug=False):  # no output
...     pass
```

```
>>> class AAA(metaclass=A_meta):               # no output
...     pass
```

Note: This example is from the docstring of the function `main__metaclass_example()` in `tests/test_log_calls.py`. In that module, we perform a fixup to the docstring which changes `'__main__'` to `__name__`, so that the test works no matter how it's invoked.

1.18 The *record_history* Decorator

The *record_history* decorator is a stripped-down version of *log_calls* which records calls to a decorated callable but writes no messages. You can think of it as *log_calls* with the `record_history` and `log_call_numbers` settings always true, with `mute` always true (equal, that is, to `log_calls.MUTE.CALLS`), and without any of the automatic message-logging apparatus.

record_history shares a great deal of functionality with *log_calls*. This chapter will note differences where they exist, and point to the corresponding documentation for *log_calls* features.

1.18.1 Usage

Import *record_history* just as you would *log_calls*:

```
>>> from log_calls import record_history
```

We'll use the following function in our examples throughout this chapter:

```
>>> @record_history()
... def record_me(a, b, x):
...     return a * x + b
```

1.18.2 Keyword parameters

record_history has only the following keyword parameters:

- `enabled`
- `prefix`
- `max_history`
- `omit`
- `only`
- `NO_DECO`
- `settings`

Of these, only three are "settings" — data that *record_history* maintains about the state of a decorated callable:

Keyword parameter	Default value	Description
`enabled`	`True`	When true, call history will be recorded
`prefix`	`''`	A `str` to prefix the function name with in call records
`max_history`	`0`	An `int`. If the value is > 0, store at most *value*-many records, with oldest records overwritten; if the value is 0, store unboundedly many records.

Setting `enabled` to true in *record_history* is like setting both *enabled* and *record_history* to true in *log_calls* (granting the analogy above about `mute`). You can supply an *indirect value* for the `enabled` parameter, as for *log_calls*.

The `enabled` and `prefix` settings are mutable; `max_history` can only be changed with the `stats.clear_history(max_history=0)` method of a decorated callable.

Use `NO_DECO` for production

Like *log_calls*, the *record_history* decorator imposes some runtime overhead. As for *log_calls*, you can use the `NO_DECO` parameter in a settings file or settings dict so that you can easily toggle decoration, as explained in *Use NO_DECO=True for production*.

1.18.3 "Settings", and the `record_history_settings` attribute

Just as the settings of *log_calls* for a decorated callable are accessible dynamically through the `log_calls_settings` attribute, the settings of *record_history* are exposed via a `record_history_settings` attribute.

`record_history_settings` is an object of the same type as `log_calls_settings`, so it has the same methods and behaviors described in the *log_calls_settings* section.

As mentioned above, *record_history* has just a few "settings":

```
>>> len(record_me.record_history_settings)
3
>>> record_me.record_history_settings.as_OD()
OrderedDict([('enabled', True), ('prefix', ''), ('max_history', 0)])
```

1.18.4 The `stats` attribute and *its* attributes

Callables decorated by *record_history* have a full-featured `stats` attribute, as described in *The stats attribute and its attributes*. In the *record_history examples* section below, we'll illustrate its use with the `record_me` function.

1.18.5 The `.print()` and `.print_exprs()` methods

Callables decorated with *record_history* can use the methods `record_history.print()` and `record_history.print_exprs()` to write debug messages. Of course, you won't want to do so in a tight loop whose performance you're profiling, but the methods are available. Output is always via the global `print` function, as *record_history* doesn't write to loggers or files. *record_history* also has the global flag `record_history.print_methods_raise_if_no_deco`, completely analogous to that of *log_calls*. See the chapter *Writing log_calls-Aware Debugging Messages* for details about these methods and the global flag.

1.18.6 The `get_record_history_wrapper()` and `get_own_record_history_wrapper()` methods

These classmethods are completely analogous to the `get_log_calls_wrapper()` and `get_own_log_calls_wrapper()` classmethods, described in the section on *accessing wrappers of methods*. They return the wrapper of a method or property decorated by *record_history*, to allow access to its attributes.

1.18.7 The `record_history.decorate_*` classmethods

The `record_history.decorate_*` classmethods exist, and behave like their *log_calls* counterparts documented in *Bulk (Re)Decoration, (Re)Decorating Imports*.

1.18.8 *record_history* examples

Let's finally call the function defined above:

```
>>> for x in range(15):
...     _ = record_me(3, 5, x)       # "_ = " for doctest
```

```
>>> len(record_me.stats.history)
15
```

Some tallies (your mileage may vary for `elapsed_secs_logged`):

```
>>> record_me.stats.num_calls_logged
15
>>> record_me.stats.num_calls_total
15
>>> record_me.stats.elapsed_secs_logged
2.2172927856445312e-05
```

Call history in CSV format, with ellipses for 'elapsed_secs', 'process_secs' and 'timestamp' columns:

```
>>> print(record_me.stats.history_as_csv)
call_num|a|b|x|retval|elapsed_secs|process_secs|timestamp|prefixed_fname|caller_chain
1|3|5|0|5|...|...|...|'record_me'|['<module>']
2|3|5|1|8|...|...|...|'record_me'|['<module>']
3|3|5|2|11|...|...|...|'record_me'|['<module>']
4|3|5|3|14|...|...|...|'record_me'|['<module>']
5|3|5|4|17|...|...|...|'record_me'|['<module>']
6|3|5|5|20|...|...|...|'record_me'|['<module>']
7|3|5|6|23|...|...|...|'record_me'|['<module>']
8|3|5|7|26|...|...|...|'record_me'|['<module>']
9|3|5|8|29|...|...|...|'record_me'|['<module>']
10|3|5|9|32|...|...|...|'record_me'|['<module>']
11|3|5|10|35|...|...|...|'record_me'|['<module>']
12|3|5|11|38|...|...|...|'record_me'|['<module>']
13|3|5|12|41|...|...|...|'record_me'|['<module>']
14|3|5|13|44|...|...|...|'record_me'|['<module>']
15|3|5|14|47|...|...|...|'record_me'|['<module>']
```

Disable recording, and call the function one more time:

```
>>> record_me.record_history_settings.enabled = False
>>> _ = record_me(583, 298, 1000)
```

The call numbers of the last 2 calls to *record_me* remain `14` and `15`:

```
>>> list(map(lambda rec: rec.call_num, record_me.stats.history[-2:]))
[14, 15]
```

Here are the call counters:

```
>>> record_me.stats.num_calls_logged
15
>>> record_me.stats.num_calls_total
16
```

Re-enable recording and call the function again, once:

```
>>> record_me.record_history_settings.enabled = True
>>> _ = record_me(1900, 2000, 20)
```

Here are the last 3 lines of the CSV call history:

```
>>> lines = record_me.stats.history_as_csv.strip().split('\\n')
>>> # Have to skip next test in .md
>>> #  because doctest doesn't split it at all: len(lines) == 1
>>> for line in lines[-3:]:
...     print(line)
14|3|5|13|44|...|...|...|'record_me'|['<module>']
15|3|5|14|47|...|...|...|'record_me'|['<module>']
16|1900|2000|20|40000|...|...|...|'record_me'|['<module>']
```

and here are the updated call counters:

```
>>> record_me.stats.num_calls_logged
16
>>> record_me.stats.num_calls_total
17
```

Finally, let's call `stats.clear_history()`, setting `max_history` to 3, call `record_me` 15 times, and examine the call history again:

```
>>> record_me.stats.clear_history(max_history=3)
>>> for x in range(15):
...     _ = record_me(3, 5, x)
>>> print(record_me.stats.history_as_csv)
call_num|a|b|x|retval|elapsed_secs|process_secs|timestamp|prefixed_fname|caller_chain
13|3|5|12|41|...|...|...|'record_me'|['<module>']
14|3|5|13|44|...|...|...|'record_me'|['<module>']
15|3|5|14|47|...|...|...|'record_me'|['<module>']
```

1.19 Appendix I: Keyword Parameters Reference

The *log_calls* decorator has several keyword parameters, all with hopefully sensible defaults.

1.19.1 Keyword parameters for "settings"

Keyword parameter	Default value	Description
`enabled`	`1` (`True`)	An `int`. If positive (or `True`), then *log_calls* will output (or "log") messages. If false ("disabled": `0`, alias `False`), *log_calls* won't output messages or record history but will continue to increment the `stats.num_calls_total` call counter. If negative ("bypassed"), *log_calls* won't do anything.
`args_sep`	`', '`	`str` used to separate arguments. The default lists all args on the same line. If `args_sep` is (or ends in) `'\n'`, then additional spaces are appended to the separator for a neater display. Other separators in which `'\n'` occurs are left unchanged, and are untested – experiment/use at your own risk.
`log_args`	`True`	If true, arguments passed to the decorated callable, and default values used, will be logged.
`log_retval`	`False`	If true, log what the decorated callable returns. At most 77 chars are printed, with a trailing ellipsis if the value is truncated.
`log_exit`	`True`	If true, the decorator will log an exiting message after the decorated callable returns and before returning what the callable returned. The message

Of these, only `prefix` and `max_history` cannot be indirect, and only `max_history` is immutable.

1.19.2 Other keyword parameters (non-"settings")

Keyword parameter	Default value	Description
`settings`	`None`	A dict mapping settings keywords and/or `NO_DECO` to values — a *settings dict* — or a string giving the pathname to a *settings file* containing settings and values. If the pathname is a directory and not a file, *log_calls* looks for a file `.log_calls` in that directory; otherwise, it looks for the named file. The format of a settings file is: zero or more lines of the form *setting* = *value*; lines whose first non-whitespace character is `'#'` are comments. These settings are a baseline; other settings passed to *log_calls* can override their values.
`name`	`''`	Specifies the display name of a decorated callable, if nonempty, and then, it must be a `str`, of the form: * the preferred name of the callable (a string literal), or * an old-style format string with one occurrence of `%s`, which the `__name__` of the decorated callable replaces.
`omit`	`tuple()`	A string or sequence of strings designating callables of a class. Supplied to a class decorator, ignored in function decorations. The designated callables will *not* be decorated. Each of these "designators" can be a name of a method or property, a name of a property with an appended qualifier `.getter`, `.setter`, or `.deleter`; it can have prefixed class names (`Outer.Inner.mymethod`). It can also contain "glob" wildcards

1.20 Appendix II: What Has Been New

This document collects the full **What's New** sections of all earlier *log_calls* releases.

- 0.2.5.post3
  - Later binding for `prefix`, though it's still not dynamically changeable.
- *0.2.5.post1 & 0.2.5.post2*
  - Silly fixups (release-bungling)
- **0.2.5**

  Performance timing/profiling enhancements & additions
  - Both elapsed time and process time are both reported now. Python 3.3+ enhances the `time` module (see **PEP 418**), and we take advantage of the new functions perf_counter and process_time.
    - Use `time.perf_counter`, `time.process_time` (Python 3.3+).
    - Added `stats.process_secs_logged` attribute.
    - Added `process_secs` column to call history (new field for `CallRecords`).
    - `log_elapsed` reports both elapsed and process times.
  - Optimized the decorator wrapper, ~15% speedup

    (still trivial with ~big data, see the IPython notebook history_to_pandas-and-profiling).
  - Added a "true bypass" feature: when `enabled` < 0, adjourn to the decorated function immedately, with no further processing. Again, not a speed speed demon – see the IPython notebook referenced above.
  - Deprecation warning issued if `settings_path` parameter used.

    (You'll see this only if you run the Python interpreter with the `-W <action>` option, where `<action>` is any valid action string other than `ignore`, e.g. `default`.)
  - Updated tests and docs to reflect these changes.
- *0.2.4.post4*
  - *(docs & description changes only, no code changes)*
- *0.2.4.post3*
  - *(never existed)*
- **0.2.4.post2**
  - The `settings` parameter (formerly `settings_path`) lets you specify default values for multiple settings, either as a dictionary or as a file. The `settings_path` parameter is deprecated, as `settings` is a superset. See the documentation http://www.pythonhosted.org/log_calls#settings-parameter for details, discussion and examples.
- **0.2.4.post1**
  - `settings_path` feature: allow `file=sys.stderr` in settings files, under IPython too; neater internals of settings file parsing.
- **0.2.4**
  - The new `settings_path` parameter lets you specify a file containing default values for multiple settings. See the documentation http://www.pythonhosted.org/log_calls#settings-parameter for details, discussion and examples.

  - You can now use a logger name (something you'd pass to `logging.getLogger()`) as the value of the `logger` setting.
  - The `indent` setting now works with loggers too. See examples:
    * using `log_message` as a general output method that works as expected, whatever the destination – `stdout`, another stream, a file, or a logger [in `tests/test_log_calls_more.py`, docstring of `main__log_message__all_possible_output_destinations()`];
    * setting up a logger with a minimal formatter that looks just like the output of `print` [in `tests/test_log_calls_more.py`, docstring of `main__logging_with_indent__minimal_formatters()`].
  - Added the decorator `used_unused_keywords` to support the `settings_path` feature, and made it visible (you can import it from the package) because it's more broadly useful. This decorator lets a function obtain, on a per-call basis, two dictionaries of its explicit keyword arguments and their values: those which were actually passed by the caller, and those which were not and received default values. For examples, see the docstring of `main()` in `used_unused_kwds.py`.
  - When displaying returned values (`log_retval` setting is true), the maximum displayed length of values is now 77, up from 60, not counting trailing ellipsis.
  - The deprecated `indent_extra` parameter to `log_message` is gone.
  - Little bug fixes, improvements.

- **0.2.3** and **0.2.3.post** *N*
  - A better signature for "the indent-aware writing method `log_message()`", and more, better examples of it — full docs http://www.pythonhosted.org/log_calls#log_message.
- **0.2.2**
  - "The indent-aware writing method `log_message()`", which decorated functions and methods can use to write extra debugging messages that align nicely with *log_calls* messages.
  - Documentation http://www.pythonhosted.org/log_calls#log_message for `log_message()`.
  - Documentation http://www.pythonhosted.org/log_calls#accessing-own-attrs for how functions and methods can access the attributes that *log_calls* adds for them, within their own bodies.
- **0.2.1**
  - The stats.history_as_DataFrame attribute, whose value is the call history of a decorated function as a Pandas DataFrame (if Pandas is installed; else `None`).
  - An IPython notebook (`log_calls/docs/history_to_pandas.ipynb`, which compares the performance of using *record_history vs* a vectorized approach using numpy to amass medium to large datasets, and which concludes that if you can vectorize, by all means do so.
- **0.2.0**
  - Initial public release.
- genindex

INDEX