Autologging Documentation

Release 1.3.0

Matthew Zipay

Feb 16, 2019

Contents

Release 1.3.0

Autologging eliminates boilerplate logging setup code and tracing code, and provides a means to separate application logging from program flow and data tracing.

Python modules that make use of Autologging are cleaner, leaner, and more resilient to changes that would otherwise require updating tracing statements.

Autologging allows for tracing to be configured (and controlled) independently from application logging. Toggle tracing on/off, write trace log records to a separate log, and use different formatting for trace log entries - all via standard Python logging facilities, and without affecting your application logging.

Python 2.7 and Python 3.4+ are supported using the same codebase. All examples given on this site use Python 3 syntax.

New in version 1.0.1: Autologging is now officially tested and working under Jython, IronPython, PyPy, and Stackless Python.

Autologging exposes two decorators (`autologging.logged`, `autologging.traced`) and a custom log level (`autologging.TRACE`).

New in version 1.1.0: Autologging now exposes the `autologging.install_traced_noop` function. This function **replaces** the `traced` decorator with a no-op that returns traced classes and functions unmodified (effectively disabling all tracing capabilities). This is useful for cases where *any* overhead from tracing is not desired (for example, when running in production environments, or when running performance tests).

New in version 1.2.0: Generator iterators now emit YIELD/STOP trace logging records (in addition to the CALL/RETURN tracing of the generator function itself).

A brief example:

```python
import logging
import sys

from autologging import logged, TRACE, traced

@traced
@logged
class Example:

    def __init__(self):
        self.__log.info("initialized")

    def backwards(self, *words):
        for word in words:
            yield "".join(reversed(word))


if __name__ == "__main__":
    logging.basicConfig(
            level=TRACE, stream=sys.stderr,
            format="%(levelname)s:%(filename)s,%(lineno)d:%(name)s.%(funcName)s:
↪%(message)s")
    example = Example()
    for result in example.backwards("spam", "eggs"):
        print(result)
```

Logging and tracing output:

```
$ python example.py
TRACE:example.py,10:__main__.Example.__init__:CALL *() **{}
INFO:example.py,11:__main__.Example.__init__:initialized
TRACE:example.py,11:__main__.Example.__init__:RETURN None
TRACE:example.py,13:__main__.Example.backwards:CALL *('spam', 'eggs') **{}
TRACE:example.py,15:__main__.Example.backwards:RETURN <generator object Example.
↪backwards at 0x7f298a450de0>
TRACE:example.py,15:__main__.Example.backwards:YIELD 'maps'
maps
TRACE:example.py,15:__main__.Example.backwards:YIELD 'sgge'
sgge
TRACE:example.py,15:__main__.Example.backwards:STOP
```

CHAPTER 1

Table of Contents

1.1 Introduction to Autologging

Release 1.3.0

When using the Python `logging` module to log classes, there are a couple of challenges that usually must be addressed by the developer:

1. The standard `logging` module is not inherently "aware" of classes in the context of logging statements made within class methods.
2. The standard `logging` module has no concept of tracing (i.e. there are neither log methods for tracing nor any log levels lower than `logging.DEBUG` to use for tracing). (See logging vs. tracing.)

Challenge #1 is not a failing of the `logging` module, but rather a side-effect of using Python stack frames to determine caller information (see `logging.Logger.findCaller`).

A reasonable workaround for #1 is to simply create a class-level logger that uses the class's qualified name as the logger name. This approach is consistent with respect to the `logging` module's recommended usage for logger naming (as well as being analogous to java.util.logging and log4j usage, upon which Python's `logging` module is based).

Challenge #2 can also be worked around, though it requires a bit more effort. Defining a new log level/name for tracing (via `logging.addLevelName`) is a start, but writing (and maintaining) the tracing log statements becomes tedious and error prone. In a language as dynamic as Python, it should not be (and isn't) necessary to do this "by hand."

As it turns out, the code necessary to create appropriately-named loggers for classes **and** to trace functions or class methods is boilerplate. The *`autologging`* module encapsulates this boilerplate code for you, allowing you to use simple decorators to get consistent class logging and tracing.

1.1.1 Logging and tracing "by hand"

```
# my_module.py

import logging

logging.addLevelName(1, "TRACE")


class MyClass:

   __log = logging.getLogger("{}.MyClass".format(__name__))

   def my_method(self, arg, keyword=None):
      self.__log.log(TRACE, "CALL %r keyword=%r", arg, keyword)
      self.__log.info("my message")
      phrase = "%s and %s" % (arg, keyword)
      self.__log.log(TRACE, "RETURN %r", phrase)
      return phrase
```

Assuming we've already configured logging to use the format *"%(levelname)s:%(name)s:%(funcName)s:%(message)s"*, calling "my_method" produces the following log output:

```
TRACE:my_module.MyClass:my_method:CALL 'spam' keyword='eggs'
INFO:my_module.MyClass:my_method:my message
TRACE:my_module.MyClass:my_method:RETURN 'spam and eggs'
```

Using this approach, we end up with several lines of visual clutter:

- The purpose of "my_method" is to join the arg and keyword together into a phrase, but there are more lines dedicated to logging/tracing than to the method logic.
- Because we wish to trace the return value of the method, we **must** set the return value as an intermediate local variable so that it can be traced, then returned. This means we can't simply use the much more succinct expression `return "%s and %s" % (arg, keyword)`.

Aside from visual clutter, there are maintenance issues as well:

- If the name of the class changes, the logger name should be updated accordingly.
- If anything about the method signature changes (number and/or position of arguments, number and/or names of keyword arguments), then the "CALL" tracing log statement must be updated manually.
- If "my_method" were ever refactored to, say, return with a different value if keyword is `None`, then we'd need to either add *another* logging statement to trace the early return, or we'd need to reconstruct the method body to set `phrase` accordingly before tracing and returning.

1.1.2 Logging and tracing with `autologging`

Autologging addresses **all** of the issues in the previous sample, resulting in less code that's more readable and easier to maintain:

```
# my_module.py

from autologging import logged, traced

@traced
@logged
class MyClass:
```

(continues on next page)

(continued from previous page)

```
    def my_method(self, arg, keyword=None):
        self.__log.info("my message")
        return "%s and %s" % (arg, keyword)
```

The method is now much easier to follow visually, requires zero logging or tracing "maintenance," and produces log output that is semantically identical to the previous example:

```
TRACE:my_module.MyClass:my_method:CALL *('spam',) **{'keyword': 'eggs'}
INFO:my_module.MyClass:my_method:my message
TRACE:my_module.MyClass:my_method:RETURN 'spam and eggs'
```

Please see *The autologging API* for details, then check out *Examples of using autologging*.

1.2 The `autologging` API

Release 1.3.0

`autologging.`**`TRACE = 1`**

A custom tracing log level, lower in severity than `logging.DEBUG`.

`autologging.`**`logged`**`(`*obj*`)`

Add a logger member to a decorated class or function.

Parameters **`obj`** – the class or function object being decorated, or an optional `logging.Logger` object to be used as the parent logger (instead of the default module-named logger)

Returns *obj* if *obj* is a class or function; otherwise, if *obj* is a logger, return a lambda decorator that will in turn set the logger attribute and return *obj*

If *obj* is a `class`, then `obj.__log` will have the logger name "<module-name>.<class-name>":

```
>>> import sys
>>> logging.basicConfig(
...     level=logging.DEBUG, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @logged
... class Sample:
...
...     def test(self):
...         self.__log.debug("This is a test.")
...
>>> Sample().test()
DEBUG:autologging.Sample:test:This is a test.
```

Note: Autologging will prefer to use the class's `__qualname__` when it is available (Python 3.3+). Otherwise, the class's `__name__` is used. For example:

```
class Outer:

   @logged
   class Nested: pass
```

Under Python 3.3+, `Nested.__log` will have the name "autologging.Outer.Nested", while under Python 2.7 or 3.2, the logger name will be "autologging.Nested".

Changed in version 0.4.0: Functions decorated with `@logged` use a *single* underscore in the logger variable name (e.g. `my_function._log`) rather than a double underscore.

If *obj* is a function, then `obj._log` will have the logger name "<module-name>":

```
>>> import sys
>>> logging.basicConfig(
...     level=logging.DEBUG, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @logged
... def test():
...     test._log.debug("This is a test.")
...
>>> test()
DEBUG:autologging:test:This is a test.
```

Note: Within a logged function, the `_log` attribute must be qualified by the function name.

If *obj* is a `logging.Logger` object, then that logger is used as the parent logger (instead of the default module-named logger):

```
>>> import sys
>>> logging.basicConfig(
...     level=logging.DEBUG, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @logged(logging.getLogger("test.parent"))
... class Sample:
...     def test(self):
...         self.__log.debug("This is a test.")
...
>>> Sample().test()
DEBUG:test.parent.Sample:test:This is a test.
```

Again, functions are similar:

```
>>> import sys
>>> logging.basicConfig(
...     level=logging.DEBUG, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @logged(logging.getLogger("test.parent"))
... def test():
...     test._log.debug("This is a test.")
...
>>> test()
DEBUG:test.parent:test:This is a test.
```

Note: For classes, the logger member is made "private" (i.e. `__log` with double underscore) to ensure that log messages that include the *%(name)s* format placeholder are written with the correct name.

Consider a subclass of a `@logged`-decorated parent class. If the subclass were **not** decorated with `@logged` and could access the parent's logger member directly to make logging calls, those log messages would display the name of the parent class, not the subclass.

Therefore, subclasses of a `@logged`-decorated parent class that wish to use a provided `self.__log` object must themselves be decorated with `@logged`.

Warning: Although the `@logged` and `@traced` decorators will "do the right thing" regardless of the order in which they are applied to the same function, it is recommended that `@logged` always be used as the innermost decorator:

```
@traced
@logged
def my_function():
    my_function._log.info("message")
```

This is because `@logged` simply sets the `_log` attribute and then returns the original function, making it "safe" to use in combination with any other decorator.

Note: Both Jython and IronPython report an "internal" class name using its mangled form, which will be reflected in the default logger name.

For example, in the sample code below, both Jython and IronPython will use the default logger name "autologging._Outer__Nested" (whereas CPython/PyPy/Stackless would use "autologging.__Nested" under Python 2 or "autologging.Outer.__Nested" under Python 3.3+)

```
class Outer:
    @logged
    class __Nested:
        pass
```

Warning: IronPython does not fully support frames (even with the -X:FullFrames option), so you are likely to see things like misreported line numbers and "<unknown file>" in log records emitted when running under IronPython.

`autologging.`**`traced`**`(`**args*, ***keywords*`)`

Add call and return tracing to an unbound function or to the methods of a class.

The arguments to `traced` differ depending on whether it is being used to trace an unbound function or the methods of a class:

Trace an unbound function using the default logger

Parameters **`func`** – the unbound function to be traced

By default, a logger named for the function's module is used:

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced
... def func(x, y):
```

(continues on next page)

(continued from previous page)

```
...     return x + y
...
>>> func(7, 9)
TRACE:autologging:func:CALL *(7, 9) **{}
TRACE:autologging:func:RETURN 16
16
```

Trace an unbound function using a named logger

Parameters **logger** (*logging.Logger*) – the parent logger used to trace the unbound function

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced(logging.getLogger("my.channel"))
... def func(x, y):
...     return x + y
...
>>> func(7, 9)
TRACE:my.channel:func:CALL *(7, 9) **{}
TRACE:my.channel:func:RETURN 16
16
```

Trace default methods using the default logger

Parameters **class** – the class whose methods will be traced

By default, all "public", "_nonpublic", and "__internal" methods, as well as the special "__init__" and "__call__" methods, will be traced. Tracing log entries will be written to a logger named for the module and class:

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced
... class Class:
...     def __init__(self, x):
...         self._x = x
...     def public(self, y):
...         return self._x + y
...     def _nonpublic(self, y):
...         return self._x - y
...     def __internal(self, y=2):
...         return self._x ** y
...     def __repr__(self):
...         return "Class(%r)" % self._x
...     def __call__(self):
...         return self._x
...
>>> obj = Class(7)
```

(continues on next page)

(continued from previous page)

```
TRACE:autologging.Class:__init__:CALL *(7,) **{}
>>> obj.public(9)
TRACE:autologging.Class:public:CALL *(9,) **{}
TRACE:autologging.Class:public:RETURN 16
16
>>> obj._nonpublic(5)
TRACE:autologging.Class:_nonpublic:CALL *(5,) **{}
TRACE:autologging.Class:_nonpublic:RETURN 2
2
>>> obj._Class__internal(y=3)
TRACE:autologging.Class:__internal:CALL *() **{'y': 3}
TRACE:autologging.Class:__internal:RETURN 343
343
>>> repr(obj) # not traced by default
'Class(7)'
>>> obj()
TRACE:autologging.Class:__call__:CALL *() **{}
TRACE:autologging.Class:__call__:RETURN 7
7
```

Note: When the runtime Python version is >= 3.3, the *qualified* class name will be used to name the tracing logger (i.e. a nested class will write tracing log entries to a logger named “module.Parent.Nested”).

Trace default methods using a named logger

Parameters **logger** (*logging.Logger*) – the parent logger used to trace the methods of the class

By default, all “public”, “_nonpublic”, and “__internal” methods, as well as the special “__init__” method, will be traced. Tracing log entries will be written to the specified logger:

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced(logging.getLogger("my.channel"))
... class Class:
...     def __init__(self, x):
...         self._x = x
...     def public(self, y):
...         return self._x + y
...     def _nonpublic(self, y):
...         return self._x - y
...     def __internal(self, y=2):
...         return self._x ** y
...     def __repr__(self):
...         return "Class(%r)" % self._x
...     def __call__(self):
...         return self._x
...
>>> obj = Class(7)
TRACE:my.channel.Class:__init__:CALL *(7,) **{}
```

(continues on next page)

(continued from previous page)

```
>>> obj.public(9)
TRACE:my.channel.Class:public:CALL *(9,) **{}
TRACE:my.channel.Class:public:RETURN 16
16
>>> obj._nonpublic(5)
TRACE:my.channel.Class:_nonpublic:CALL *(5,) **{}
TRACE:my.channel.Class:_nonpublic:RETURN 2
2
>>> obj._Class__internal(y=3)
TRACE:my.channel.Class:__internal:CALL *() **{'y': 3}
TRACE:my.channel.Class:__internal:RETURN 343
343
>>> repr(obj) # not traced by default
'Class(7)'
>>> obj()
TRACE:my.channel.Class:__call__:CALL *() **{}
TRACE:my.channel.Class:__call__:RETURN 7
7
```

Trace specified methods using the default logger

Parameters **method_names** (*tuple*) – the names of the methods that will be traced

Tracing log entries will be written to a logger named for the module and class:

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced("public", "__internal")
... class Class:
...     def __init__(self, x):
...         self._x = x
...     def public(self, y):
...         return self._x + y
...     def _nonpublic(self, y):
...         return self._x - y
...     def __internal(self, y=2):
...         return self._x ** y
...     def __repr__(self):
...         return "Class(%r)" % self._x
...     def __call__(self):
...         return self._x
...
>>> obj = Class(7)
>>> obj.public(9)
TRACE:autologging.Class:public:CALL *(9,) **{}
TRACE:autologging.Class:public:RETURN 16
16
>>> obj._nonpublic(5)
2
>>> obj._Class__internal(y=3)
TRACE:autologging.Class:__internal:CALL *() **{'y': 3}
TRACE:autologging.Class:__internal:RETURN 343
343
```

(continues on next page)

(continued from previous page)

```
>>> repr(obj)
'Class(7)'
>>> obj()
7
```

Warning: When method names are specified explicitly via *args*, Autologging ensures that each method is actually defined in the body of the class being traced. (This means that inherited methods that are not overridden are **never** traced, even if they are named explicitly in *args*.)

If a defintion for any named method is not found in the class body, either because the method is inherited or because the name is misspelled, Autologging will issue a `UserWarning`.

If you wish to trace a method from a super class, you have two options:

1. Use `traced` to decorate the super class.
2. Override the method and trace it in the subclass.

Note: When the runtime Python version is >= 3.3, the *qualified* class name will be used to name the tracing logger (i.e. a nested class will write tracing log entries to a logger named "module.Parent.Nested").

Trace specified methods using a named logger

Parameters

- **logger** (*logging.Logger*) – the parent logger used to trace the methods of the class
- **method_names** (*tuple*) – the names of the methods that will be traced

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced(logging.getLogger("my.channel"), "public", "__internal")
... class Class:
...     def __init__(self, x):
...         self._x = x
...     def public(self, y):
...         return self._x + y
...     def _nonpublic(self, y):
...         return self._x - y
...     def __internal(self, y=2):
...         return self._x ** y
...     def __repr__(self):
...         return "Class(%r)" % self._x
...     def __call__(self):
...         return self._x
...
>>> obj = Class(7)
>>> obj.public(9)
TRACE:my.channel.Class:public:CALL *(9,) **{}
TRACE:my.channel.Class:public:RETURN 16
```

(continues on next page)

(continued from previous page)

```
16
>>> obj._nonpublic(5)
2
>>> obj._Class__internal(y=3)
TRACE:my.channel.Class:__internal:CALL *() **{'y': 3}
TRACE:my.channel.Class:__internal:RETURN 343
343
>>> repr(obj) # not traced by default
'Class(7)'
>>> obj()
7
```

Warning: When method names are specified explicitly via *args*, Autologging ensures that each method is actually defined in the body of the class being traced. (This means that inherited methods that are not overridden are **never** traced, even if they are named explicitly in *args*.)

If a defintion for any named method is not found in the class body, either because the method is inherited or because the name is misspelled, Autologging will issue a `UserWarning`.

If you wish to trace a method from a super class, you have two options:

1. Use `traced` to decorate the super class.
2. Override the method and trace it in the subclass.

Exclude specified methods from tracing

New in version 1.3.0.

Parameters

- **`method_names`** (*`tuple`*) – the names of the methods that will be excluded from tracing
- **`exclude`** (*`bool`*) – `True` to cause the method names list to be interpreted as an exclusion list (`False` is the default, and causes the named methods to be **included** as described above)

The example below demonstrates exclusions using the default logger.

```
>>> import sys
>>> logging.basicConfig(
...     level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> @traced("_nonpublic", "__internal", exclude=True)
... class Class:
...     def __init__(self, x):
...         self._x = x
...     def public(self, y):
...         return self._x + y
...     def _nonpublic(self, y):
...         return self._x - y
...     def __internal(self, y=2):
...         return self._x ** y
...     def __repr__(self):
...         return "Class(%r)" % self._x
...     def __call__(self):
...         return self._x
```

(continues on next page)

(continued from previous page)

```
...
>>> obj = Class(7)
>>> obj.public(9)
TRACE:autologging.Class:public:CALL *(9,) **{}
TRACE:autologging.Class:public:RETURN 16
16
>>> obj._nonpublic(5)
2
>>> obj._Class__internal(y=3)
343
>>> repr(obj)
'Class(7)'
>>> obj()
TRACE:autologging.Class:__call__:CALL *() **{}
TRACE:autologging.Class:__call__:RETURN 7
7
```

When method names are excluded via *args* and the *exclude* keyword, Autologging **ignores** methods that are not actually defined in the body of the class being traced.

Warning: If an exclusion list causes the list of traceable methods to resolve empty, then Autologging will issue a `UserWarning`.

Note: When the runtime Python version is >= 3.3, the *qualified* class name will be used to name the tracing logger (i.e. a nested class will write tracing log entries to a logger named "module.Parent.Nested").

Note: When tracing a class, if the default (class-named) logger is used **and** the runtime Python version is >= 3.3, then the *qualified* class name will be used to name the tracing logger (i.e. a nested class will write tracing log entries to a logger named "module.Parent.Nested").

Note: If method names are specified when decorating a function, a `UserWarning` is issued, but the methods names are ignored and the function is traced as though the method names had not been specified.

Note: Both Jython and IronPython report an "internal" class name using its mangled form, which will be reflected in the default tracing logger name.

For example, in the sample code below, both Jython and IronPython will use the default tracing logger name "autologging._Outer__Nested" (whereas CPython/PyPy/Stackless would use "autologging.__Nested" under Python 2 or "autologging.Outer.__Nested" under Python 3.3+):

```
class Outer:
   @traced
   class __Nested:
      pass
```

Warning: Neither Jython nor IronPython currently implement the `function.__code__.co_lnotab` attribute, so the last line number of a function cannot be determined by Autologging. As a result, the line number reported in tracing RETURN log records will actually be the line number on which the function is defined.

`autologging.`**`install_traced_noop`**`()`

Replace the *`traced`* decorator with an identity (no-op) decorator.

Although the overhead of a `@traced` function or method is minimal when the `TRACED` log level is disabled, there is still *some* overhead (the logging level check, an extra function call).

If you would like to completely *eliminate* this overhead, call this function **before** any classes or functions in your application are decorated with `@traced`. The *`traced`* decorator will be replaced with a no-op decorator that simply returns the class or function unmodified.

Note: The **recommended** way to install the no-op `@traced` decorator is to set the `AUTOLOGGING_TRACED_NOOP` environment variable to any non-empty value.

If the `AUTOLOGGING_TRACED_NOOP` environment variable is set to a non-empty value when Autologging is loaded, the `@traced` no-op will be installed automatically.

As an alternative to setting the `AUTOLOGGING_TRACED_NOOP` environment variable, you can also call this function directly in your application's bootstrap module. For example:

```python
import autologging

if running_in_production:
    autologging.install_traced_noop()
```

Warning: This function **does not** "revert" any already-`@traced` class or function! It simply replaces the `autologging.traced` module reference with a no-op.

For this reason it is imperative that `autologging.install_traced_noop()` be called **before** the `@traced` decorator has been applied to any class or function in the application. (This is why the `AUTOLOGGING_TRACED_NOOP` environment variable is the recommended approach for installing the no-op - it allows Autologging itself to guarantee that the no-op is installed before any classes or functions are decorated.)

1.3 Examples of using `autologging`

Release 1.3.0

1.3.1 Using the `@logged` decorator

Release 1.3.0

- *Add a module-named logger to a class*
- *Add a user-named logger to a class*
- *Add a logger to a nested class*

- *Add a module-named logger to a function*
- *Add a user-named logger to a function*
- *Add a logger to a nested function*

Add a module-named logger to a class

```
# my_module.py

from autologging import logged


@logged
class MyClass:

    @staticmethod
    def my_staticmethod():
        MyClass.__log.info("my message")

    @classmethod
    def my_classmethod(cls):
        cls.__log.info("my message")

    def my_method(self):
        self.__log.info("my message")
```

```
>>> import logging, sys
>>> logging.basicConfig(
...     level=logging.INFO, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> MyClass.my_staticmethod()
INFO:my_module.MyClass:my_staticmethod:my message
>>> MyClass.my_classmethod()
INFO:my_module.MyClass:my_classmethod:my message
>>> my_obj = MyClass()
>>> my_obj.my_method()
INFO:my_module.MyClass:my_method:my message
```

Add a user-named logger to a class

```
# my_module.py

import logging
from autologging import logged


@logged(logging.getLogger("my.app"))
class MyClass:

    @staticmethod
    def my_staticmethod():
        MyClass.__log.info("my message")
```

(continues on next page)

(continued from previous page)

```
    @classmethod
    def my_classmethod(cls):
        cls.__log.info("my message")

    def my_method(self):
        self.__log.info("my message")
```

```
>>> import logging, sys
>>> logging.basicConfig(
...     level=logging.INFO, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> MyClass.my_staticmethod()
INFO:my.app.MyClass:my_staticmethod:my message
>>> MyClass.my_classmethod()
INFO:my.app.MyClass:my_classmethod:my message
>>> my_obj = MyClass()
>>> my_obj.my_method()
INFO:my.app.MyClass:my_method:my message
```

Add a logger to a nested class

```
# my_module.py

from autologging import logged


@logged
class MyClass:

   @logged
   class _Nested:

      def __init__(self):
         self.__log.info("my message")

   def my_method(self):
      self.__log.info("my message")
      nested = self._Nested()
```

```
>>> import logging, sys
>>> logging.basicConfig(
...     level=logging.INFO, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass()
>>> my_obj.my_method()
INFO:my_module.MyClass:my_method:my message
INFO:my_module.MyClass._Nested:__init__:my message
```

Add a module-named logger to a function

```
# my_module.py

from autologging import logged


@logged
def my_function():
    my_function._log.info("my message")
```

```
>>> import logging, sys
>>> logging.basicConfig(
...     level=logging.INFO, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function()
INFO:my_module:my_function:my message
```

Add a user-named logger to a function

```
# my_module.py

import logging
from autologging import logged


@logged(logging.getLogger("my.app"))
def my_function():
    my_function._log.info("my message")
```

```
>>> import logging, sys
>>> logging.basicConfig(
...     level=logging.INFO, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function()
INFO:my.app:my_function:my message
```

Add a logger to a nested function

```
# my_module.py

from autologging import logged

@logged
def my_function():
   @logged
   def nested_function():
      nested_function._log.info("my message")
   my_function._log.info("my message")
   nested_function()
```

```
>>> import logging, sys
>>> logging.basicConfig(
...     level=logging.INFO, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function()
INFO:my_module:my_function:my message
INFO:my_module:nested_function:my message
```

1.3.2 Using the `@traced` decorator

Release 1.3.0

- *Trace all methods of a class using a module-named logger*
- *Trace all methods of a class using a user-named logger*
- *Trace only certain methods of a class*
- *Exclude certain methods of a class from tracing*
- *Trace a generator iterator*
- *Trace a nested class*
- *Trace a function using a module-named logger*
- *Trace a function using a user-named logger*
- *Trace a nested function*

Warning: The `@traced` decorator will not work as you might expect when it decorates a function (method) in the body of a class. In general, prefer to decorate the class itself, explicitly identifying the method names to trace if the default (all class, static, and instance methods, excluding "__special__" methods, with the exception of `__init__` and `__call__`) doesn't suit your needs.

Trace all methods of a class using a module-named logger

This is the simplest way to use the `autologging.traced` decorator. All *non-special* methods of the class are traced to a logger that is named after the containing module and class. (Note: the special `__init__` method is an exception to the rule - it is traced by default if it is defined.)

Note: Inherited methods are **never** traced. If you want tracing for inherited methods, either trace them in the super class, or override and trace them in the subclass.

```
# my_module.py

from autologging import traced


@traced
class MyClass:

   def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        self._value = "ham"

    def my_method(self, arg, keyword=None):
        return "%s, %s, and %s" % (arg, self._value, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass()
TRACE:my_module.MyClass:__init__:CALL *() **{}
TRACE:my_module.MyClass:__init__:RETURN None
>>> my_obj.my_method("spam", keyword="eggs")
TRACE:my_module.MyClass:my_method:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:my_module.MyClass:my_method:RETURN 'spam, ham, and eggs'
'spam, ham, and eggs'
```

Trace all methods of a class using a user-named logger

This example is identical to the above example, except that the tracing logger has a user-defined name ("tracing.example" in this case). Simply pass the user-defined logger as the first positional argument to `traced`:

```
# my_module.py

import logging
from autologging import traced


@traced(logging.getLogger("tracing.example"))
class MyClass:

    def __init__(self):
        self._value = "ham"

    def my_method(self, arg, keyword=None):
        return "%s, %s, and %s" % (arg, self._value, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass()
TRACE:tracing.example.MyClass:__init__:CALL *() **{}
TRACE:tracing.example.MyClass:__init__:RETURN None
>>> my_obj.my_method("spam", keyword="eggs")
TRACE:tracing.example.MyClass:my_method:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:tracing.example.MyClass:my_method:RETURN 'spam, ham, and eggs'
'spam, ham, and eggs'
```

Trace only certain methods of a class

The `traced` decorator accepts a variable number of positional string arguments. As you saw in the previous example, passing a user-defined logger as the first argument allows you to specify the parent logger for tracing. You may also pass a variable number of method names as arguments to `traced`. Autologging will then trace only the methods that are named (assuming that they are defined in the class body). And as in the previous example, you may still choose whether or not to pass in a named logger as the *first* argument (not shown below).

```
# my_module.py

from autologging import traced


@traced("my_method", "__eq__")
class MyClass:

   def __init__(self):
      self._value = "ham"

   def my_method(self, arg, keyword=None):
      return "%s, %s, and %s" % (arg, self._value, keyword)

   def __eq__(self, other):
      return False
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass()  # __init__ is not in the list, so not traced
>>> my_obj.my_method("spam", keyword="eggs")
TRACE:my_module.MyClass:my_method:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:my_module.MyClass:my_method:RETURN 'spam, ham, and eggs'
'spam, ham, and eggs'
>>> my_obj == 79  # __eq__ is explicitly named in the list
TRACE:my_module.MyClass:__eq__:CALL *(79,) **{}
TRACE:my_module.MyClass:__eq__:RETURN False
False
```

Exclude certain methods of a class from tracing

New in version 1.3.0.

In cases where a class has a relatively large number of methods, and you want to trace *most* (but not all) of them, it is more intuitive to **exclude** the methods that should not be traced. By specifying the `exclude=True` keyword argument, you can "invert" the semantic meaning of the named method list passed to `@traced()` – now Autologging will determine the default list of method names to trace, then *remove* the named methods from the list.

```
# my_module.py

from autologging import traced


@traced("__init__", exclude=True)
class MyClass:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self._value = "ham"

    def my_method(self, arg, keyword=None):
        return "%s, %s, and %s" % (arg, self._value, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass() # __init__ was explicitly exlcuded
>>> my_obj.my_method("spam", keyword="eggs")
TRACE:my_module.MyClass:my_method:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:my_module.MyClass:my_method:RETURN 'spam, ham, and eggs'
'spam, ham, and eggs'
```

Trace a generator iterator

New in version 1.2.0.

Generator functions employ the `yield` keyword in the function body, instructing Python to create (and return) a generator iterator when the function is invoked:

```
# my_module.py

from autologging import traced


@traced
class MyClass:

    def my_iter(self, word):
        for character in reversed(word):
            yield character.upper()
```

To observe how Autologging traces both the *generator* and its returned *generator iterator*, assume we run the program like so:

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass()
>>> for c in my_obj.my_iter("spam"):
...     print(c)
...
(continued below)
```

Because the *generator* function `my_iter` is traced, Autologging will dutifully emit the CALL/RETURN trace logging records:

```
TRACE:my_module.MyClass:my_iter:CALL *('spam',) **{}
TRACE:my_module.MyClass:my_iter:RETURN <generator object MyClass.my_iter at␣
↪0x7f54f4043840>
(continued below)
```

In versions of Autologging **prior to 1.2.0**, this would be the only tracing output. But as of version 1.2.0, the *generator iterator* is now traced as well, and will emit additional YIELD/STOP trace logging records:

```
TRACE:my_module.MyClass:my_iter:YIELD 'M'
M
TRACE:my_module.MyClass:my_iter:YIELD 'A'
A
TRACE:my_module.MyClass:my_iter:YIELD 'P'
P
TRACE:my_module.MyClass:my_iter:YIELD 'S'
S
TRACE:my_module.MyClass:my_iter:STOP
```

Trace a nested class

Tracing a nested class is no different than tracing a module-level class:

```
# my_module.py

from autologging import traced


class MyClass:

   @traced
   class Nested:

      def do_something(self):
         pass
```

Note: Under Python 3.3+, Autologging will use a class's qualified name (**PEP 3155**) when creating loggers. In this example, the tracing log entries will be logged using the name "my_module.MyClass.Nested". (Under versions of Python <3.3, where "__qualname__" is not available, the logger name would be simply "my_module.Nested".)

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> nested = MyClass.Nested()
>>> nested.do_something()
TRACE:my_module.MyClass.Nested:do_something:CALL *() **{}
TRACE:my_module.MyClass.Nested:do_something:RETURN None
```

Trace a function using a module-named logger

```
# my_module.py

from autologging import traced


@traced
def my_function(arg, keyword=None):
   return "%s and %s" % (arg, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function("spam", keyword="eggs")
TRACE:my_module:my_function:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:my_module:my_function:RETURN 'spam and eggs'
'spam and eggs'
```

Trace a function using a user-named logger

```
# my_module.py

import logging
from autologging import traced


@traced(logging.getLogger("my.app"))
def my_function(arg, keyword=None):
   return "%s and %s" % (arg, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function("spam", keyword="eggs")
TRACE:my.app:my_function:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:my.app:my_function:RETURN 'spam and eggs'
'spam and eggs'
```

Trace a nested function

```
# my_module.py

from autologging import traced


def my_function(arg, keyword=None):
   @traced
   def nested_function(word1, word2):
```

(continues on next page)

(continued from previous page)

```
        return "%s and %s" % (word1, word2)
    return nested_function(arg, keyword if (keyword is not None) else "eggs")
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function("spam")
TRACE:my_module:nested_function:CALL *('spam', 'eggs') **{}
TRACE:my_module:nested_function:RETURN 'spam and eggs'
'spam and eggs'
```

1.3.3 Disabling `@traced` at runtime

Release 1.3.0

New in version 1.1.0.

- *Recommended approach: Let Autologging install the @traced no-op automatically*
- *Install the @traced no-op directly*

When a class or a function is decorated with `autologging.traced`, Autologging *replaces* a method (or function) with a tracing "proxy" function that intercepts any invocation to perform the CALL/RETURN tracing. (See *How does autologging work?* for a more in-depth explanation.)

Autologging is careful to minimize the overhead imposed by these tracing proxy functions, but there will always be *some* overhead - even when the `autologging.TRACE` log level is disabled, the tracing proxy function must still (a) check the log level and then (b) delegate to the original method/function.

In some cases, it may be desirable to completely eliminate *any* tracing overhead. Running the application in a production environment, or executing a performance test, are the two most obvious examples. For these cases, Autologging now provides a "killswitch" for the `@traced` decorator that effectively turns it into a no-op.

When the `autologging.install_traced_noop` function is called **before** the traced decorator is imported and any classes or functions have been decorated, then the `@traced` decorator will function as an "identity" decorator, simply returning the decorated class or function **unmodified**. In this way, tracing can be completely "removed" from an application *without* having to comment-out the `@traced` decorator (or conditionally apply it).

Warning: Calling `autologging.install_traced_noop()` *after* any class or function has been decorated **will not uninstall the tracing proxy function.** It is imperative that `autologging.install_traced_noop()` be called *before* `autologging.traced` is imported and before any class or function is decorated.

The process is irreversible in the running Python interpreter. Once `autologging.install_traced_noop()` has been called, the only way to reinstate tracing functionality is to restart the interpreter (and *not* install the no-op).

Consider a simple application consisting of two modules - a bootstrap module that is responsible for launching the application, and the application module itself:

```
# bootstrap.py

```

(continues on next page)

(continued from previous page)

```
import app

if __name__ == "__main__":
    app.launch()
```

```
# app.py

from autologging import traced


@traced
class Application:

   def __init__(self):
      self._prepare_connections()
      # other initialization

   def _prepare_connections(self):
      # prepare the connections

   def run(self):
      while True:
         # listen for events, then handle them

   def handle(self, event):
      # handle the event


@traced
def launch():
   Application().run()
```

As written, the "__init__", "_prepare_connections", "run", and "handle" methods of the application object, and the "launch" function in the `app` module, will be replaced by tracing proxies.

Note: Whether or not any tracing info is *emitted* to logs can be controlled simply by configuring the logging system appropriately.

Let's suppose that for this particular application we wish to completely *remove* tracing capability when running in a production or performance testing environment. To accomplish this, we need to call the *autologging.install_traced_noop* function **before** any class or function is decorated with `@traced`.

Recommended approach: Let Autologging install the `@traced` no-op automatically

The recommended approach is to instruct Autologging to install the no-op automatically. This can be accomplished by setting the `AUTOLOGGING_TRACED_NOOP` environment variable to any **non-empty** value.

For example, we would run the application in a production or performance testing environment like so:

```
$ export AUTOLOGGING_TRACED_NOOP=1
$ python bootstrap.py
```

Note: This is the recommended approach because it requires no change to the application or bootstrap source code.

If the conditions under which tracing should be deactivated are more complex, then direct installation of the `@traced` no-op may be necessary.

Install the `@traced` no-op directly

If automatic installation of the `@traced` no-op is not possible (or not preferred), then the application bootstrap code can be modified to install the no-op directly.

The following modification to *bootstrap.py* accomplishes the goal:

```python
# bootstrap.py

import os
import autologging

# MUST happen before importing app!
if os.getenv("APP_ENV") in ("PRODUCTION", "PERFORMACE_TEST"):
   autologging.install_traced_noop()

import app

if __name__ == "__main__":
    app.launch()
```

1.3.4 Using `@logged` and `@traced` together

Release 1.3.0

- *Add logging and tracing to a class*
- *Add logging and tracing to a function*

Add logging and tracing to a class

```python
# my_module.py

from autologging import logged, traced


@traced
@logged
class MyClass:

   def __init__(self, value):
      self.__log.info("I like %s.", value)
      self._value = value

   def my_method(self, arg, keyword=None):
      return "%s, %s, and %s" % (arg, self._value, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
```

(continues on next page)

(continued from previous page)

```
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import MyClass
>>> my_obj = MyClass("ham")
TRACE:my_module.MyClass:__init__:CALL *('ham',) **{}
INFO:my_module.MyClass:__init__:I like ham.
TRACE:my_module.MyClass:__init__:RETURN None
>>> my_obj.my_method("spam", keyword="eggs")
TRACE:my_module.MyClass:my_method:CALL *('spam',) **{'keyword': 'eggs'}
TRACE:my_module.MyClass:my_method:RETURN 'spam, ham, and eggs'
'spam, ham, and eggs'
```

Add logging and tracing to a function

Warning: Although the `@logged` and `@traced` decorators will "do the right thing" regardless of the order in which they are applied to the same function, it is recommended that `@logged` always be used as the innermost decorator.

This is because `@logged` simply sets the logger member and then returns the original function, making it safe to use in combination with any other decorator.

```
# my_module.py

from autologging import logged, traced


@traced
@logged
def my_function(arg, keyword=None):
   my_function._log.info("my message")
   return "%s and %s" % (arg, keyword)
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import my_function
>>> my_function("spam", keyword="eggs")
TRACE:my_module:my_function:CALL *('spam',) **{'keyword': 'eggs'}
INFO:my_module:my_function:my message
TRACE:my_module:my_function:RETURN 'spam and eggs'
'spam and eggs'
```

1.3.5 Logging, tracing, and inheritance

Release 1.3.0

Autologging's policy on inheritance is simple:

Loggers are never inherited, and inherited methods are never traced.

Practically speaking, this means that you must be explicit when using *autologging.logged* and *autologging.traced* throughout a class's bases.

If a class in the hierarchy will use a logger, then *that* class definition must be `@logged`. Likewise, if a method inherited from a super class in the hierarchy will be traced, then either the method must be traced in the *super* class or the method must be overridden (and traced) in a subclass.

The following example illustrates these concepts:

```
# my_module.py

from autologging import logged, traced


@traced
@logged
class Base:

   # this method will be traced
   def method(self):
      # log channel will be "my_module.Base"
      self.__log.info("base message")
      return "base"


@logged
class Parent(Base):

   # this method will NOT be traced
   def method(self):
      # log channel will be "my_module.Parent"
      self.__log.info("parent message")
      return super().method() + ",parent"


@traced
class Child(Parent):

   # this method will be traced
   def method(self):
      return super().method() + ",child"
```

```
>>> import logging, sys
>>> from autologging import TRACE
>>> logging.basicConfig(level=TRACE, stream=sys.stdout,
...     format="%(levelname)s:%(name)s:%(funcName)s:%(message)s")
>>> from my_module import Child
>>> child = Child()
>>> child.method()
TRACE:my_module.Child:method:CALL *() **{}
INFO:my_module.Parent:method:parent message
TRACE:my_module.Base:method:CALL *() **{}
INFO:my_module.Base:method:base message
TRACE:my_module.Base:method:RETURN 'base'
TRACE:my_module.Child:method:RETURN 'base,parent,child'
'base,parent,child'
```

1.3.6 Separate configurations for logging and tracing

Release 1.3.0

The purpose of logging is to capture information (caught errors, branching decisions, calculation results, etc.).

The purpose of tracing is to capture program flow and data (which functions and methods are called, in what sequence, and with what parameters).

In addition to serving different purposes, logging and tracing typically have different intended audiences. While logging output is generally useful for *anyone* who must observe an application (developers as well as QA testers, administrators, business analysts, etc.), tracing is of use primarily for developers.

Because the purpose of and audience for logging and tracing differ, it is often convenient to configure and control them separately. This may include, but is not limited to:

- being able to enable/disable logging and tracing independent of one another
- writing logging output and tracing output to different log files
- using different log entry formatting for logging and tracing

Standard Python `logging` configuration can be used in combination with Autologging to accomplish these goals.

In the example module below, we have logged and traced a simple class:

```
# my_module.py

import logging

from autologging import logged, traced


@traced
@logged
class MyClass:

   def my_method(self, arg, keyword=None):
      if keyword is not None:
         self.__log.debug("taking the keyword branch")
         return "{} and {}".format(arg, keyword)
      return arg.upper()
```

We will now configure the logging system to write *two* log files - one that contains all log entries (including tracing), and another that contains **only** non-tracing log entries:

```
# my_module_main.py

import logging
import logging.config

import autologging

from my_module import MyClass

logging.config.dictConfig({
    "version": 1,
    "formatters": {
        "logformatter": {
            "format":
                "%(asctime)s:%(levelname)s:%(name)s:%(funcName)s:%(message)s",
        },
        "traceformatter": {
            "format":
```

(continues on next page)

(continued from previous page)

```
                "%(asctime)s:%(process)s:%(levelname)s:%(filename)s:"
                    "%(lineno)s:%(name)s:%(funcName)s:%(message)s",
        },
    },
    "handlers": {
        "loghandler": {
            "class": "logging.FileHandler",
            "level": logging.DEBUG,
            "formatter": "logformatter",
            "filename": "app.log",
        },
        "tracehandler": {
            "class": "logging.FileHandler",
            "level": autologging.TRACE,
            "formatter": "traceformatter",
            "filename": "trace.log",
        },
    },
    "loggers": {
        "my_module.MyClass": {
            "level": autologging.TRACE,
            "handlers": ["tracehandler", "loghandler"],
        },
    },
})

if __name__ == "__main__":
    obj = MyClass()
    obj.my_method("test")
    obj.my_method("spam", keyword="eggs")
```

If we now run the application, it will produce two log files (“app.log” and “trace.log”).

The “app.log” file contains the single DEBUG record:

```
2016-01-17 19:58:52,639:DEBUG:my_module.MyClass:my_method:taking the keyword branch
```

The “trace.log” file contains call and return tracing for both method calls as well as the DEBUG record:

```
2016-01-17 19:58:52,639:24100:TRACE:my_module.py:12:my_module.MyClass:my_method:CALL␣
↪*('test',) **{}
2016-01-17 19:58:52,639:24100:TRACE:my_module.py:16:my_module.MyClass:my_
↪method:RETURN 'TEST'
2016-01-17 19:58:52,639:24100:TRACE:my_module.py:12:my_module.MyClass:my_method:CALL␣
↪*('spam',) **{'keyword': 'eggs'}
2016-01-17 19:58:52,639:24100:DEBUG:my_module.py:14:my_module.MyClass:my_
↪method:taking the keyword branch
2016-01-17 19:58:52,639:24100:TRACE:my_module.py:16:my_module.MyClass:my_
↪method:RETURN 'spam and eggs'
```

Many other configurations are possible using various combinations of `logging.config` settings and/or explicitly-named trace loggers via *`autologging.traced`*.

1.4 How does autologging work?

Release 1.3.0

The *autologging.logged* decorator is rather boring - it simply creates a `logging.Logger` object and sets it as an attribute of the decorated class (or function).

However, in order to automate tracing while **preserving** introspection and subclassing capabilities, the *autologging.traced* decorator has a tougher job.

To trace the call and return of a particular function, Autologging performs the following steps:

1. Intercept the function call, capturing all positional and keyword arguments that were passed.
2. Log the call at the *autologging.TRACE* log level.
3. Call the *original* function, passing the positional and keyword arguments, and capturing the return value.
4. Log the return at the *autologging.TRACE* log level.
5. Return the value.

Autologging installs a **replacement** function for any traced class method or function. That replacement function is responsible for the steps described above.

Note: If the `TRACE` level is disabled for the tracing logger when a traced function is called, the replacement function delegates directly to the original function.

Which functions are traced?

A quick way to determine which functions have been traced is to look for the `__autologging_traced__` attribute. Autologging sets this attribute to the value `True` on every replacement function. For example:

```
>>> from autologging import traced
>>> @traced
... def example():
...     return "OK"
...
>>> hasattr(example, "__autologging_traced__")
True
>>> @traced
... class Example:
...     @classmethod
...     def class_method(cls):
...         return "OK"
...     @staticmethod
...     def static_method():
...         return "OK"
...     def method(self):
...         return "OK"
...
>>> hasattr(Example.class_method, "__autologging_traced__")
True
>>> hasattr(Example.static_method, "__autologging_traced__")
True
>>> hasattr(Example.method, "__autologging_traced__")
True
```

Introspecting traced functions

When Autologging installs a replacement function for tracing, a reference to the original function is stored as the `__wrapped__` attribute of the replacement function.

```
>>> from autologging import traced
>>> def example():
...     return "OK"
...
>>> traced(example).__wrapped__ is example
True
```

```
>>> from autologging import traced
>>> class Example:
...     def method(self):
...         return "OK"
...
>>> original_method = Example.__dict__["method"]
>>> traced(Example).__dict__["method"].__wrapped__ is original_method
True
```

Traced `classmethod` and `staticmethod` functions are also replaced by Autologging, but in addition to creating a replacement function, Autologging also creates a replacement method descriptor. To access the original function of a classmethod or staticmethod, you must use the `__wrapped__` attribute *of the __func__ attribute* of the replacement classmethod or staticmethod. An example makes this clear:

```
>>> from autologging import traced
>>> class Example:
...     @classmethod
...     def class_method(cls):
...         return "OK"
...     @staticmethod
...     def static_method():
...         return "OK"
...
>>> original_classmethod = Example.__dict__["class_method"]
>>> original_staticmethod = Example.__dict__["static_method"]
>>> Example = traced(Example)
>>> Example.__dict__["class_method"].__func__.__wrapped__ is original_classmethod.__
↪func__
True
>>> Example.__dict__["static_method"].__func__.__wrapped__ is original_staticmethod.__
↪func__
True
```

Inheritance and subclassing with traced methods

Autologging is careful to not “break” assumptions about the types of methods, or how those methods are inherited or overridden.

A replacement tracing method (or method descriptor, in the case of classmethods and staticmethods) has the same type, name and signature as the original method:

```
>>> import inspect
>>> from types import FunctionType, MethodType
```

(continues on next page)

(continued from previous page)

```
>>> from autologging import traced
>>> @traced
... class Example:
...     @classmethod
...     def class_method(cls, arg, keyword=None):
...         return "OK"
...     @staticmethod
...     def static_method(arg, keyword=None):
...         return "OK"
...     def method(self, arg, keyword=None):
...         return "OK"
...
>>> type(Example.__dict__["class_method"]) is classmethod
True
>>> Example.class_method.__name__
'class_method'
>>> inspect.signature(Example.class_method)
<Signature (arg, keyword=None)>
>>> type(Example.__dict__["static_method"]) is staticmethod
True
>>> Example.static_method.__name__
'static_method'
>>> inspect.signature(Example.static_method)
<Signature (arg, keyword=None)>
>>> type(Example.__dict__["method"]) is FunctionType
True
>>> type(Example().method) is MethodType
True
>>> Example.method.__name__
'method'
>>> inspect.signature(Example().method)
<Signature (arg, keyword=None)>
```

1.5 Autologging 1.3.0 testing summary

1.5.1 CPython

Version	Platform
3.7.2	[GCC 8.2.1 20181215 (Red Hat 8.2.1-6)] on Linux 4.20.7-100.fc28.x86_64 x86_64
3.7.0	[Clang 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
	[MSC v.1914 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
3.6.8	[GCC 8.2.1 20181215 (Red Hat 8.2.1-6)] on Linux 4.20.7-100.fc28.x86_64 x86_64
3.6.6	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
	[MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
3.5.6	[GCC 8.1.1 20180712 (Red Hat 8.1.1-5)] on Linux 4.20.7-100.fc28.x86_64 x86_64
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
3.5.4	[MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
3.4.9	[GCC 8.1.1 20180712 (Red Hat 8.1.1-5)] on Linux 4.20.7-100.fc28.x86_64 x86_64
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
3.4.4	[MSC v.1600 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
2.7.15	[GCC 8.1.1 20180712 (Red Hat 8.1.1-5)] on Linux 4.20.7-100.fc28.x86_64 x86_64
	[MSC v.1500 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
	[GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)

1.5.2 PyPy

Version	Platform
3.5.3	[PyPy 6.0.0 with GCC 7.2.0] on Linux 4.20.7-100.fc28.x86_64 x86_64
	[PyPy 6.0.0 with GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
	[PyPy 5.10.1 with MSC v.1500 32 bit] on Windows-10-10.0.17134-SP0
2.7.13	[PyPy 6.0.0 with GCC 7.2.0] on Linux 4.20.7-100.fc28.x86_64 x86_64
	[PyPy 6.0.0 with GCC 4.2.1 Compatible Apple LLVM 9.1.0 (clang-902.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
	[PyPy 6.0.0 with MSC v.1500 32 bit] on Windows-10-10.0.17134-SP0

1.5.3 Stackless Python

Version	Platform
3.6.4	Stackless 3.1b3 060516 [GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on Linux 4.20.7-100.fc28.x86_64 x86_64
	Stackless 3.1b3 060516 [MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
	Stackless 3.1b3 060516 [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)
3.5.4	Stackless 3.1b3 060516 [MSC v.1900 64 bit (AMD64)] on Windows-10-10.0.17134-SP0
2.7.14	Stackless 3.1b3 060516 [GCC 6.4.1 20170727 (Red Hat 6.4.1-1)] on Linux 4.20.7-100.fc28.x86_64 x86_64
	Stackless 3.1b3 060516 [GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)

1.5.4 Jython

Version	Platform
2.7.1	[OpenJDK 64-Bit Server VM (Oracle Corporation)] (Java-11.0.1-OpenJDK_64-Bit_Server_VM) on Linux 4.20.7-100.fc28.x86_64 x86_64
	[Java HotSpot(TM) 64-Bit Server VM ("Oracle Corporation")] (Java-10.0.2-Java_HotSpot-TM-_64-Bit_Server_VM) on Windows-10-10.0.17134-SP0
	[Java HotSpot(TM) 64-Bit Server VM (Oracle Corporation)] (Java-9.0.4-Java_HotSpot-TM-_64-Bit_Server_VM) on Darwin 17.7.0 x86_64 (Mac OS X 10.13.6)

1.5.5 IronPython

Version	Platform
2.7.8	(IronPython 2.7.8 (2.7.8.0) on .NET 4.0.30319.42000 (64-bit)) on Windows-10-10.0.17134-SP0
	(IronPython 2.7.8 (2.7.8.0) on .NET 4.0.30319.42000 (32-bit)) on Windows-10-10.0.17134-SP0

CHAPTER 2

Download and Install

The easiest way to install Autologging is to use pip:

```
$ pip install Autologging
```

To install from source, clone or fork the repository:

```
$ git clone https://github.com/mzipay/Autologging.git
```

Alternatively, download and extract a source .zip or .tar.gz archive from https://github.com/mzipay/Autologging/releases or https://pypi.python.org/pypi/Autologging.

Run the test suite and install the `autologging` module (make sure you have setuptools installed!):

```
$ cd Autologging
$ python setup.py test
$ python setup.py install
```

You can also install from one of the available binary packages available at https://pypi.python.org/pypi/Autologging or https://sourceforge.net/projects/autologging/files/.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index