wrapt Documentation

Release 1.12.1

Graham Dumpleton

Mar 09, 2020

Contents

A Python module for decorators, wrappers and monkey patching.

CHAPTER 1

Overview

The aim of the **wrapt** module is to provide a transparent object proxy for Python, which can be used as the basis for the construction of function wrappers and decorator functions.

An easy to use decorator factory is provided to make it simple to create your own decorators that will behave correctly in any situation they may be used.

```
import wrapt

@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass
```

In addition to the support for creating object proxies, function wrappers and decorators, the module also provides a post import hook mechanism and other utilities useful in performing monkey patching of code.

The **wrapt** module focuses very much on correctness. It therefore goes way beyond existing mechanisms such as `functools.wraps()` to ensure that decorators preserve introspectability, signatures, type checking abilities etc. The decorators that can be constructed using this module will work in far more scenarios than typical decorators and provide more predictable and consistent behaviour.

To ensure that the overhead is as minimal as possible, a C extension module is used for performance critical components. An automatic fallback to a pure Python implementation is also provided where a target system does not have a compiler to allow the C extension to be compiled.

CHAPTER 2

Documentation

2.1 Getting Started

To implement your decorator you need to first define a wrapper function. This will be called each time a decorated function is called. The wrapper function needs to take four positional arguments:

- `wrapped` - The wrapped function which in turns needs to be called by your wrapper function.
- `instance` - The object to which the wrapped function was bound when it was called.
- `args` - The list of positional arguments supplied when the decorated function was called.
- `kwargs` - The dictionary of keyword arguments supplied when the decorated function was called.

The wrapper function would do whatever it needs to, but would usually in turn call the wrapped function that is passed in via the `wrapped` argument.

The decorator `@wrapt.decorator` then needs to be applied to the wrapper function to convert it into a decorator which can in turn be applied to other functions.

```
import wrapt

@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass
```

If you wish to implement a decorator which accepts arguments, then wrap the definition of the decorator in a function closure. Any arguments supplied to the outer function when the decorator is applied, will be available to the inner wrapper when the wrapped function is called.

```
import wrapt
```

(continues on next page)

(continued from previous page)

```
def with_arguments(myarg1, myarg2):
    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return wrapper

@with_arguments(1, 2)
def function():
    pass
```

When applied to a normal function or static method, the wrapper function when called will be passed `None` as the `instance` argument.

When applied to an instance method, the wrapper function when called will be passed the instance of the class the method is being called on as the `instance` argument. This will be the case even when the instance method was called explicitly via the class and the instance passed as the first argument. That is, the instance will never be passed as part of `args`.

When applied to a class method, the wrapper function when called will be passed the class type as the `instance` argument.

When applied to a class, the wrapper function when called will be passed `None` as the `instance` argument. The `wrapped` argument in this case will be the class.

The above rules can be summarised with the following example.

```
import inspect

@wrapt.decorator
def universal(wrapped, instance, args, kwargs):
    if instance is None:
        if inspect.isclass(wrapped):
            # Decorator was applied to a class.
            return wrapped(*args, **kwargs)
        else:
            # Decorator was applied to a function or staticmethod.
            return wrapped(*args, **kwargs)
    else:
        if inspect.isclass(instance):
            # Decorator was applied to a classmethod.
            return wrapped(*args, **kwargs)
        else:
            # Decorator was applied to an instancemethod.
            return wrapped(*args, **kwargs)
```

Using these checks it is therefore possible to create a universal decorator that can be applied in all situations. It is no longer necessary to create different variants of decorators for normal functions and instance methods, or use additional wrappers to convert a function decorator into one that will work for instance methods.

In all cases, the wrapped function passed to the wrapper function is called in the same way, with `args` and `kwargs` being passed. The `instance` argument doesn't need to be used in calling the wrapped function.

2.2 Function Decorators

The **wrapt** module provides various components, but the main reason that it would be used is for creating decorators. This document covers the creation of decorators and all the information needed to cover what you can do within the

wrapper function linked to your decorator.

2.2.1 Creating Decorators

To implement your decorator you need to first define a wrapper function. This will be called each time a decorated function is called. The wrapper function needs to take four positional arguments:

- `wrapped` - The wrapped function which in turns needs to be called by your wrapper function.
- `instance` - The object to which the wrapped function was bound when it was called.
- `args` - The list of positional arguments supplied when the decorated function was called.
- `kwargs` - The dictionary of keyword arguments supplied when the decorated function was called.

The wrapper function would do whatever it needs to, but would usually in turn call the wrapped function that is passed in via the `wrapped` argument.

The decorator `@wrapt.decorator` then needs to be applied to the wrapper function to convert it into a decorator which can in turn be applied to other functions.

```
import wrapt

@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass
```

2.2.2 Decorators With Arguments

If you wish to implement a decorator which accepts arguments, then you can wrap the definition of the decorator in a function closure. Any arguments supplied to the outer function when the decorator is applied, will be available to the inner wrapper when the wrapped function is called.

```
import wrapt

def with_arguments(myarg1, myarg2):
    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)
    return wrapper

@with_arguments(1, 2)
def function():
    pass
```

If using Python 3, you can use the keyword arguments only syntax to force use of keyword arguments when the decorator is used.

```
import wrapt

def with_keyword_only_arguments(*, myarg1, myarg2):
    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
```

(continues on next page)

(continued from previous page)

```
        return wrapped(*args, **kwargs)
    return wrapper

@with_keyword_only_arguments(myarg1=1, myarg2=2)
def function():
    pass
```

An alternative approach to using a function closure to allow arguments is to use a class, where the wrapper function is the `__call__()` method of the class.

```
import wrapt

class with_arguments(object):

    def __init__(self, myarg1, myarg2):
        self.myarg1 = myarg1
        self.myarg2 = myarg2

    @wrapt.decorator
    def __call__(self, wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)

@with_arguments(1, 2)
def function():
    pass
```

In this case the wrapper function should also accept a `self` argument as is normal for instance methods of a class. The arguments to the decorator would then be accessed by the wrapper function from the class instance created when the decorator was applied to the target function, via the `self` argument.

Using a class in this way has the added benefit that other functions can be associated with the class providing for better encapsulation. The alternative would have been to have the class be separate and use it in conjunction with a function closure, where the class instance would have been created as a local variable within the outer function when called.

2.2.3 Decorators With Optional Arguments

Although opinion can be mixed about whether the pattern is a good one, if the decorator arguments all have default values, it is also possible to implement decorators which have optional arguments. This allows the decorator to be applied with or without the arguments, with the brackets being able to be dropped in the latter.

```
import wrapt

def with_optional_arguments(wrapped=None, myarg1=1, myarg2=2):
    if wrapped is None:
        return functools.partial(with_optional_arguments,
                myarg1=myarg1, myarg2=myarg2)

    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)

    return wrapper(wrapped)

@with_optional_arguments(myarg1=1, myarg2=2)
def function():
```

(continues on next page)

(continued from previous page)

```
    pass

@with_optional_arguments
def function():
    pass
```

For this to be used in this way, it is a requirement that the decorator arguments be supplied as keyword arguments.

If using Python 3, the requirement to use keyword only arguments can again be enforced using the keyword only argument syntax.

```
import wrapt

def with_optional_arguments(wrapped=None, *, myarg1=1, myarg2=2):
    if wrapped is None:
        return functools.partial(with_optional_arguments,
                myarg1=myarg1, myarg2=myarg2)

    @wrapt.decorator
    def wrapper(wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)

    return wrapper(wrapped)
```

2.2.4 Processing Function Arguments

The original set of positional arguments and keyword arguments supplied when the decorated function is called will be passed in the `args` and `kwargs` arguments.

Note that these are always passed as their own unique arguments and are not broken out and bound in any way to the decorator wrapper arguments. In other words, the decorator wrapper function signature must always be:

```
@wrapt.decorator
def my_decorator(wrapped, instance, args, kwargs): # CORRECT
    return wrapped(*args, **kwargs)
```

You cannot use:

```
@wrapt.decorator
def my_decorator(wrapped, instance, *args, **kwargs): # WRONG
    return wrapped(*args, **kwargs)
```

nor can you specify actual named arguments to which `args` and `kwargs` would be bound.

```
@wrapt.decorator
def my_decorator(wrapped, instance, arg1, arg2): # WRONG
    return wrapped(arg1, arg2)
```

Separate arguments are used and no binding performed to avoid the possibility of name collisions between the arguments passed to a decorated function when called, and the names used for the `wrapped` and `instance` arguments. This can happen for example were `wrapped` and `instance` also used as keyword arguments by the wrapped function.

If needing to modify certain arguments being supplied to the decorated function when called, you will thus need to trigger binding of the arguments yourself. This can be done using a nested function which in turn then calls the wrapped function:

```
@wrapt.decorator
def my_decorator(wrapped, instance, args, kwargs):
    def _execute(arg1, arg2, *_args, **_kwargs):

        # Do something with arg1 and arg2 and then pass the
        # modified values to the wrapped function. Use 'args'
        # and 'kwargs' on the nested function to mop up any
        # unexpected or non required arguments so they can
        # still be passed through to the wrapped function.

        return wrapped(arg1, arg2, *_args, **_kwargs)

    return _execute(*args, **kwargs)
```

If you do not need to modify the arguments being passed through to the wrapped function, but still need to extract them so as to log them or otherwise use them as input into some process you could instead use.

```
@wrapt.decorator
def my_decorator(wrapped, instance, args, kwargs):
    def _arguments(arg1, arg2, *args, **kwargs):
        return (arg1, arg2)

    arg1, arg2 = _arguments(*args, **kwargs)

    # Do something with arg1 and arg2 but still pass through
    # the original arguments to the wrapped function.

    return wrapped(*args, **kwargs)
```

You should not simply attempt to extract positional arguments from `args` directly because this will fail if those positional arguments were actually passed as keyword arguments, and so were passed in `kwargs` with `args` being an empty tuple.

Note that in either case, the argument names used in the decorated function would need to match the names mapped by the wrapper function. This is a restriction which would need to be documented for the specific decorator to ensure that users do not use arbitrary argument names which do not match.

2.2.5 Enabling/Disabling Decorators

A problem with using decorators is that once added into code, the actions of the wrapper function cannot be readily disabled. The use of the decorator would have to be removed from the code, or the specific wrapper function implemented in such a way as to check itself a flag indicating whether it should do what is required, or simply call the original wrapped function without doing anything.

To make the task of enabling/disabling the actions of a wrapper function easier, such functionality is built in to `wrapt.decorator`. The feature operates at a couple of levels, but in all cases, the `enabled` option is used to `wrapt.decorator`. This must be supplied as a keyword argument and cannot be supplied as a positional argument.

In the first way in which this enabling feature can work, if it is supplied a boolean value, then it will immediately control whether a wrapper is applied around the function that the decorator was in turn applied to.

In other words, where the `enabled` option was `True`, then the decorator will still be applied to the target function and will operate as normal.

```
ENABLED = True
```

(continues on next page)

(continued from previous page)

```
@wrapt.decorator(enabled=ENABLED)
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass

>>> type(function)
<type 'FunctionWrapper'>
```

If however the `enabled` option was `False`, then no wrapper is added to the target function and the original function returned instead.

```
ENABLED = False

@wrapt.decorator(enabled=ENABLED)
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function():
    pass

>>> type(function)
<type 'function'>
```

In this scenario, as no wrapper is applied there is no runtime overhead at the point of call when the decorator had been disabled. This therefore provides a convenient way of globally disabling a specific decorator without having to remove all uses of the decorator, or have a special variant of the decorator function.

2.2.6 Dynamically Disabling Decorators

Supplying a boolean value for the `enabled` option when defining a decorator provides control over whether the decorator should be applied or not. This is therefore a global switch and once disabled it cannot be dynamically re-enabled at runtime while the process is executing. Similarly, once enabled it cannot be disabled.

An alternative to supplying a literal boolean, is to provide a callable for `enabled` which will yield a boolean value.

```
def _enabled():
    return True

@wrapt.decorator(enabled=_enabled)
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)
```

When a callable function is supplied in this way, the callable will be invoked each time the decorated function is called. If the callable returns `True`, indicating that the decorator is active, the wrapper function will then be called. If the callable returns `False` however, the wrapper function will be bypassed and the original wrapped function called directly.

If `enabled` is not `None`, nor a boolean, or a callable, then a boolean check will be done on the object supplied instead. This allows one to use a custom object which supports logical operations. If the custom object evaluates as `False` the wrapper function will again be bypassed.

2.2.7 Function Argument Specifications

To obtain the argument specification of a decorated function the standard `getargspec()` function from the `inspect` module can be used.

```
@wrapt.decorator
def my_decorator(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@my_decorator
def function(arg1, arg2):
    pass

>>> print(inspect.getargspec(function))
ArgSpec(args=['arg1', 'arg2'], varargs=None, keywords=None, defaults=None)
```

If using Python 3, the `getfullargspec()` or `signature()` functions from the `inspect` module can also be used.

In other words, applying a decorator created using `@wrapt.decorator` to a function is signature preserving and does not result in the loss of the original argument specification as would occur when more simplistic decorator patterns are used.

2.2.8 Wrapped Function Documentation

To obtain documentation for a decorated function which may be specified in a documentation string of the original wrapped function, the standard Python help system can be used.

```
@wrapt.decorator
def my_decorator(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@my_decorator
def function(arg1, arg2):
    """Function documentation."""
    pass

>>> help(function)
Help on function function in module __main__:

function(arg1, arg2)
    Function documentation.
```

Just the documentation string itself can still be obtained by accessing the `__doc__` attribute of the decorated function.

```
>>> print(function.__doc__)
Function documentation.
```

2.2.9 Wrapped Function Source Code

To obtain the source code of a decorated function the standard `getsource()` function from the `inspect` module can be used.

```
@wrapt.decorator
def my_decorator(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@my_decorator
def function(arg1, arg2):
    pass

>>> print(inspect.getsource(function))
@my_decorator
def function(arg1, arg2):
    pass
```

As with signatures, the use of the decorator does not prevent access to the original source code for the wrapped function.

2.2.10 Signature Changing Decorators

When using `inspect.getargspec()` the argument specification for the original wrapped function is returned. If however the decorator is a signature changing decorator, this is not going to be what is desired.

In this circumstance you can pass a dummy function to the decorator via the optional `adapter` argument. When this is done, the argument specification will be sourced from the prototype for this dummy function.

```
def _my_adapter_prototype(arg1, arg2): pass

@wrapt.decorator(adapter=_my_adapter_prototype)
def my_adapter(wrapped, instance, args, kwargs):
    """Adapter documentation."""

    def _execute(arg1, arg2, *_args, **_kwargs):

        # We actually multiply the first two arguments together
        # and pass that in as a single argument. The prototype
        # exposed by the decorator is thus different to that of
        # the wrapped function.

        return wrapped(arg1*arg2, *_args, **_kwargs)

    return _execute(*args, **kwargs)

@my_adapter
def function(arg):
    """Function documentation."""

    pass

>>> help(function)
Help on function function in module __main__:

function(arg1, arg2)
    Function documentation.
```

As it would not be accidental that you applied such a signature changing decorator to a function, it would normally be the case that such usage would be explained within the documentation for the wrapped function. As such, the documentation for the wrapped function is still what is used for the `__doc__` string and what would appear when using the Python help system. In the latter, the arguments required of the adapter would though instead appear.

If you need to generate the argument specification based on the function being wrapped dynamically, you can instead pass a tuple of the form which is returned by `inspect.getargspec()`, or a string of the form which is returned by `inspect.formatargspec()`. In these two cases the decorator will automatically compile a stub function to use as the adapter. This eliminates the need for a caller to generate the stub function if generating the signature on the fly.

```
def argspec_factory(wrapped):
    argspec = inspect.getargspec(wrapped)

    args = argspec.args[1:]
    defaults = argspec.defaults and argspec.defaults[-len(argspec.args):]

    return inspect.ArgSpec(args, argspec.varargs,
            argspec.keywords, defaults)

def session(wrapped):
    @wrapt.decorator(adapter=argspec_factory(wrapped))
    def _session(wrapped, instance, args, kwargs):
        with transaction() as session:
            return wrapped(session, *args, **kwargs)

    return _session(wrapped)
```

This mechanism and the original mechanism to pass a function, require that the adapter function has to be created in advance. If the adapter needs to be generated on demand for the specific function to be wrapped, then it is necessary to use a closure around the definition of the decorator as above, such that the generator can be passed in.

As a convenience, instead of using such a closure, you can instead use:

```
def argspec_factory(wrapped):
    argspec = inspect.getargspec(wrapped)

    args = argspec.args[1:]
    defaults = argspec.defaults and argspec.defaults[-len(argspec.args):]

    return inspect.ArgSpec(args, argspec.varargs,
            argspec.keywords, defaults)

@wrapt.decorator(adapter=wrapt.adapter_factory(argspec_factory))
def _session(wrapped, instance, args, kwargs):
    with transaction() as session:
        return wrapped(session, *args, **kwargs)
```

The result of `wrapt.adapter_factory()` will be recognised as indicating that the creation of the adapter is to be deferred until the decorator is being applied to a function. The factory function for generating the adapter function or specification on demand will be passed the function being wrapped by the decorator.

If wishing to create a library of routines for generating adapter functions or specifications dynamically, then you can do so by creating classes which derive from `wrapt.AdapterFactory` as that is the type which is recognised as indicating lazy evaluation of the adapter function. For example, `wrapt.adapter_factory()` is itself implemented as:

```
class DelegatedAdapterFactory(wrapt.AdapterFactory):
    def __init__(self, factory):
        super(DelegatedAdapterFactory, self).__init__()
        self.factory = factory
    def __call__(self, wrapped):
        return self.factory(wrapped)
```

(continues on next page)

(continued from previous page)

```
adapter_factory = DelegatedAdapterFactory
```

2.2.11 Decorating Functions

When applying a decorator to a normal function, the `instance` argument would always be `None`.

```
@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
def function(arg1, arg2):
    pass

function(1, 2)
```

2.2.12 Decorating Instance Methods

When applying a decorator to an instance method, the `instance` argument will be the instance of the class on which the instance method is called. That is, it would be the same as `self` passed as the first argument to the actual instance method.

```
@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

class Class(object):

    @pass_through
    def function_im(self, arg1, arg2):
        pass

c = Class()

c.function_im(1, 2)

Class.function_im(c, 1, 2)
```

Note that the `self` argument is only passed via `instance`, it is not passed as part of `args`. Only the arguments following on from the `self` argument will be a part of args.

When calling the wrapped function in the decorator wrapper function, the `instance` should never be passed explicitly though. This is because the instance is already bound to `wrapped` and will be passed automatically as the first argument to the original wrapped function.

This is even the situation where the instance method was called via the class type and the `self` pointer passed explicitly. This is the case as the decorator identifies this specific case and adjusts `instance` and `args` so that the decorator wrapper function does not see it as being any different to where it was called directly on the instance.

2.2.13 Decorating Class Methods

When applying a decorator to a class method, the `instance` argument will be the class type on which the class method is called. That is, it would be the same as `cls` passed as the first argument to the actual class method.

```
@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

class Class(object):

    @pass_through
    @classmethod
    def function_cm(cls, arg1, arg2):
        pass

Class.function_cm(1, 2)
```

Note that the `cls` argument is only passed via `instance`, it is not passed as part of `args`. Only the arguments following on from the `cls` argument will be a part of args.

When calling the wrapped function in the decorator wrapper function, the `instance` should never be passed explicitly though. This is because the instance is already bound to `wrapped` and will be passed automatically as the first argument to the original wrapped function.

Note that due to a bug in Python `classmethod.__get__()`, whereby it does not apply the descriptor protocol to the function wrapped by `@classmethod`, the above only applies where the decorator wraps the `@classmethod` decorator. If the decorator is placed inside of the `@classmethod` decorator, then `instance` will be `None` and the decorator wrapper function will see the call as being the same as a normal function. As a result, always place any decorator outside of the `@classmethod` decorator. Hopefully this issue in Python can be addressed in a future Python version.

2.2.14 Decorating Static Methods

When applying a decorator to a static method, the `instance` argument will be `None`. In other words, the decorator wrapper function will not be able to distinguish a call to a static method from a normal function.

```
@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

class Class(object):

    @pass_through
    @staticmethod
    def function_sm(arg1, arg2):
        pass

Class.function_sm(1, 2)
```

2.2.15 Decorating Classes

When applying a decorator to a class, the `instance` argument will be `None`. In order to distinguish this case from a normal function call, `inspect.isclass()` should be used on `wrapped` to determine if it is a class type.

```
@wrapt.decorator
def pass_through(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@pass_through
class Class(object):
    pass

c = Class()
```

Do note that whenever decorating a class, as you are replacing the aliased name for the class with a wrapper, it will complicate use of the class in cases where the original type is required.

In particular, if using `super()`, it is necessary to supply the original type and the wrapper cannot be used. It will therefore be necessary to use the `__wrapped__` attribute to get access to the original type, as in:

```
@pass_through
class Class(BaseClass):
    def __init__(self):
        super(Class.__wrapped__, self).__init__()
```

In this case one could also use:

```
@pass_through
class Class(BaseClass):
    def __init__(self):
        BaseClass.__init__(self)
```

but in general, use of `super()` in conjunction with the `__wrapped__` attribute to get access to the original type is still recommended.

If using Python 3, the issue can be avoided by simply using the new magic `super()` calling convention whereby the type and `self` argument are not required.

```
@pass_through
class Class(BaseClass):
    def __init__(self):
        super().__init__()
```

The need for the new magic `super()` in Python 3 was actually in part driven by this specific case where the class type can have a decorator applied.

2.2.16 Universal Decorators

A universal decorator is one that can be applied to different types of functions and can adjust automatically based on what is being decorated.

For example, the decorator may be able to be used on both a normal function and an instance method, thereby avoiding the need to create two separate decorators to be used in each case.

A universal decorator can be created by observing what has been stated above in relation to the expected values/types for `wrapped` and `instance` passed to the decorator wrapper function.

These rules can be summarised by the following.

```
import inspect
```

(continues on next page)

(continued from previous page)

```
@wrapt.decorator
def universal(wrapped, instance, args, kwargs):
    if instance is None:
        if inspect.isclass(wrapped):
            # Decorator was applied to a class.
            return wrapped(*args, **kwargs)
        else:
            # Decorator was applied to a function or staticmethod.
            return wrapped(*args, **kwargs)
    else:
        if inspect.isclass(instance):
            # Decorator was applied to a classmethod.
            return wrapped(*args, **kwargs)
        else:
            # Decorator was applied to an instancemethod.
            return wrapped(*args, **kwargs)
```

To be truly robust, if a universal decorator is being applied in a scenario it does not support, it should raise a runtime exception at the point it is called.

2.3 Proxies and Wrappers

Underlying the implementation of the decorators created by the **wrapt** module is a wrapper class which acts as a transparent object proxy. This document describes the object proxy and the various custom wrappers provided.

2.3.1 Object Proxy

The object proxy class is available as `wrapt.ObjectProxy`. The class would not normally be used directly, but as a base class to custom object proxies or wrappers which add behaviour which overrides that of the original object. When an object proxy is used, it will pass through any actions performed on the proxy through to the wrapped object.

```
>>> table = {}
>>> proxy = wrapt.ObjectProxy(table)
>>> proxy['key-1'] = 'value-1'
>>> proxy['key-2'] = 'value-2'

>>> proxy.keys()
['key-2', 'key-1']
>>> table.keys()
['key-2', 'key-1']

>>> isinstance(proxy, dict)
True

>>> dir(proxy)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'has_key',
'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
```

This ability for a proxy to stand in for the original goes as far as arithmetic operations, rich comparison and hashing.

```
>>> value = 1
>>> proxy = wrapt.ObjectProxy(value)

>>> proxy + 1
2

>>> int(proxy)
1
>>> hash(proxy)
1
>>> hash(value)
1

>>> proxy < 2
True
>>> proxy == 0
False
```

Do note however, that when wrapping an object proxy around a literal value, the original value is effectively copied into the proxy object and any operation which updates the value will only update the value held by the proxy object.

```
>>> value = 1
>>> proxy = wrapt.ObjectProxy(value)
>>> type(proxy)
<type 'ObjectProxy'>

>>> proxy += 1

>>> type(proxy)
<type 'ObjectProxy'>

>>> print(proxy)
2
>>> print(value)
1
```

Object wrappers may therefore have limited use in conjunction with literal values.

2.3.2 Type Comparison

The type of an instance of the object proxy will be `ObjectProxy`, or that of any derived class type if creating a custom object proxy.

```
>>> value = 1
>>> proxy = wrapt.ObjectProxy(value)
>>> type(proxy)
<type 'ObjectProxy'>

>>> class CustomProxy(wrapt.ObjectProxy):
...     pass

>>> proxy = CustomProxy(1)

>>> type(proxy)
<class '__main__.CustomProxy'>
```

Direct type comparisons in Python are generally frowned upon and allowance for 'duck typing' preferred. Instead of direct type comparison, the `isinstance()` function would therefore be used. Using `isinstance()`, comparison of the type of the object proxy will properly evaluate against the wrapped object.

```
>>> isinstance(proxy, int)
True
```

This works because the `__class__` attribute actually returns the class type for the wrapped object.

```
>>> proxy.__class__
<type 'int'>
```

Note that `isinstance()` will still also succeed if comparing to the `ObjectProxy` type. It is therefore still possible to use `isinstance()` to determine if an object is an object proxy.

```
>>> isinstance(proxy, wrapt.ObjectProxy)
True

>>> class CustomProxy(wrapt.ObjectProxy):
...     pass

>>> proxy = CustomProxy(1)

>>> isinstance(proxy, wrapt.ObjectProxy)
True
>>> isinstance(proxy, CustomProxy)
True
```

2.3.3 Custom Object Proxies

A custom proxy is where one creates a derived object proxy and overrides some specific behaviour of the proxy.

```
def function():
    print('executing', function.__name__)

class CallableWrapper(wrapt.ObjectProxy):

    def __call__(self, *args, **kwargs):
        print('entering', self.__wrapped__.__name__)
        try:
            return self.__wrapped__(*args, **kwargs)
        finally:
            print('exiting', self.__wrapped__.__name__)

>>> proxy = CallableWrapper(function)

>>> proxy()
('entering', 'function')
('executing', 'function')
('exiting', 'function')
```

Any method of the original wrapped object can be overridden, including special Python methods such as `__call__()`. If it is necessary to change what happens when a specific attribute of the wrapped object is accessed, then properties can be used.

If it is necessary to access the original wrapped object from within an overridden method or property, then `self.__wrapped__` is used.

2.3.4 Proxy Object Attributes

When an attempt is made to access an attribute from the proxy, the same named attribute would in normal circumstances be accessed from the wrapped object. When updating an attributes value, or deleting the attribute, that change will also be reflected in the wrapped object.

```
>>> proxy = CallableWrapper(function)

>>> hasattr(function, 'attribute')
False
>>> hasattr(proxy, 'attribute')
False

>>> proxy.attribute = 1

>>> hasattr(function, 'attribute')
True
>>> hasattr(proxy, 'attribute')
True

>>> function.attribute
1
>>> proxy.attribute
1
```

If an attribute was updated on the wrapped object directly, that change is still reflected in what is available via the proxy.

```
>>> function.attribute = 2

>>> function.attribute
2
>>> proxy.attribute
2
```

If creating a custom proxy and it needs to keep attributes of its own which should not be saved through to the wrapped object, those attributes should be prefixed with `_self_`.

```
def function():
    print('executing', function.__name__)

class CallableWrapper(wrapt.ObjectProxy):

    def __init__(self, wrapped, wrapper):
        super(CallableWrapper, self).__init__(wrapped)
        self._self_wrapper = wrapper

    def __call__(self, *args, **kwargs):
        return self._self_wrapper(self.__wrapped__, args, kwargs)

def wrapper(wrapped, args, kwargs):
      print('entering', wrapped.__name__)
      try:
          return wrapped(*args, **kwargs)
      finally:
          print('exiting', wrapped.__name__)
```

(continues on next page)

(continued from previous page)

```
>>> proxy = CallableWrapper(function, wrapper)

>>> proxy._self_wrapper
<function wrapper at 0x1005961b8>

>>> function._self_wrapper
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute '_self_wrapper'
```

If an attribute local to the proxy must be available under a name without this special prefix, then a `@property` can be used in the class definition.

```
class CustomProxy(wrapt.ObjectProxy):

    def __init__(self, wrapped):
        super(CustomProxy, self).__init__(wrapped)
        self._self_attribute = 1

    @property
    def attribute(self):
        return self._self_attribute

    @attribute.setter
    def attribute(self, value):
        self._self_attribute = value

    @attribute.deleter
    def attribute(self):
       del self._self_attribute

>>> proxy = CustomProxy(1)
>>> print proxy.attribute
1
>>> proxy.attribute = 2
>>> print proxy.attribute
2
>>> del proxy.attribute
>>> print proxy.attribute
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'attribute'
```

Alternatively, the attribute can be specified as a class attribute, with that then being overridden if necessary, with a specific value in the `__init__()` method of the class.

```
class CustomProxy(wrapt.ObjectProxy):
    attribute = None
    def __init__(self, wrapped):
        super(CustomProxy, self).__init__(wrapped)
        self.attribute = 1

>>> proxy = CustomProxy(1)
>>> print proxy.attribute
1
>>> proxy.attribute = 2
```

(continues on next page)

(continued from previous page)

```
>>> print proxy.attribute
2
>>> del proxy.attribute
>>> print proxy.attribute
None
```

Just be aware that although the attribute can be deleted from the instance of the custom proxy, lookup will then fallback to using the class attribute.

2.3.5 Function Wrappers

Although an `ObjectProxy` can be used to wrap a function, it doesn't do anything special in respect of bound methods. If attempting to use a custom object proxy to wrap instance methods, class methods or static methods, it would be necessary to override the appropriate descriptor protocol methods in order to be able to intercept and modify in any way the execution of the wrapped function.

```
class BoundCallableWrapper(wrapt.ObjectProxy):

    def __init__(self, wrapped, wrapper):
        super(BoundCallableWrapper, self).__init__(wrapped)
        self._self_wrapper = wrapper

    def __get__(self, instance, owner):
        return self

    def __call__(self, *args, **kwargs):
        return self._self_wrapper(self.__wrapped__, args, kwargs)

class CallableWrapper(wrapt.ObjectProxy):

    def __init__(self, wrapped, wrapper):
        super(CallableWrapper, self).__init__(wrapped)
        self._self_wrapper = wrapper

    def __get__(self, instance, owner):
        function = self.__wrapped__.__get__(instance, owner)
        return BoundCallableWrapper(function, self._self_wrapper)

    def __call__(self, *args, **kwargs):
        return self._self_wrapper(self.__wrapped__, args, kwargs)
```

The `CallableWrapper.__call__()` method would therefore be invoked when `CallableWrapper` is used around a regular function. The `BoundCallableWrapper.__call__()` would instead be what is invoked for a bound method, the instance of `BoundCallableWrapper` having being created when the original wrapped method was bound to the class instance.

This specific pattern is actually the basis of what is required to implement a robust function wrapper for use in implementing a decorator. Because it is a fundamental pattern, a predefined version is available as `wrapt.FunctionWrapper`.

As with the illustrative example above, `FunctionWrapper` class accepts two key arguments:

- `wrapped` - The function being wrapped.
- `wrapper` - A wrapper function to be called when the wrapped function is invoked.

Although in prior examples the wrapper function was shown as accepting three positional arguments of the wrapped function and the args and kwargs for when the wrapped function was called, when using `FunctionWrapper`, it is expected that the wrapper function accepts four arguments. These are:

- `wrapped` - The wrapped function which in turns needs to be called by your wrapper function.
- `instance` - The object to which the wrapped function was bound when it was called.
- `args` - The list of positional arguments supplied when the decorated function was called.
- `kwargs` - The dictionary of keyword arguments supplied when the decorated function was called.

When `FunctionWrapper` is applied to a normal function or static method, the wrapper function when called will be passed `None` as the `instance` argument.

When applied to an instance method, the wrapper function when called will be passed the instance of the class the method is being called on as the `instance` argument. This will be the case even when the instance method was called explicitly via the class and the instance passed as the first argument. That is, the instance will never be passed as part of `args`.

When applied to a class method, the wrapper function when called will be passed the class type as the `instance` argument.

When applied to a class, the wrapper function when called will be passed `None` as the `instance` argument. The `wrapped` argument in this case will be the class.

The above rules can be summarised with the following example.

```
import inspect

def wrapper(wrapped, instance, args, kwargs):
    if instance is None:
        if inspect.isclass(wrapped):
            # Decorator was applied to a class.
            return wrapped(*args, **kwargs)
        else:
            # Decorator was applied to a function or staticmethod.
            return wrapped(*args, **kwargs)
    else:
        if inspect.isclass(instance):
            # Decorator was applied to a classmethod.
            return wrapped(*args, **kwargs)
        else:
            # Decorator was applied to an instancemethod.
            return wrapped(*args, **kwargs)
```

Using these checks it is therefore possible to create a universal function wrapper that can be applied in all situations. It is no longer necessary to create different variants of function wrappers for normal functions and instance methods.

In all cases, the wrapped function passed to the wrapper function is called in the same way, with `args` and `kwargs` being passed. The `instance` argument doesn't need to be used in calling the wrapped function.

A simple decorator factory implementation which makes use of `FunctionWrapper` to delegate execution of the wrapped function to the wrapper function would be:

```
def function_wrapper(wrapper):
    @functools.wraps(wrapper)
    def _wrapper(wrapped):
        return FunctionWrapper(wrapped, wrapper)
    return _wrapper
```

It would be used like:

```
@function_wrapper
def wrapper(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@wrapper
def function():
    pass
```

This example of a simplified decorator factory is made available as `wrapt.function_wrapper`. Although it is usable in its own right, it is preferable that `wrapt.decorator` be used to create decorators as it provides additional features. The `@function_wrapper` decorator would generally be used more when performing monkey patching and needing to dynamically create function wrappers.

```
@function_wrapper
def wrapper(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

callback = wrapper(fetch_callback())
```

2.3.6 Custom Function Wrappers

If it is necessary to implement a custom function wrapper in order to override the behaviour of a wrapped function, it is possible to still derive from the `wrapt.FunctionWrapper` class. That binding of functions can occur, does however complicate this. This is because the bound function is a separate object implemented as a different type.

The type of the separate bound function wrapper is `wrapt.BoundFunctionWrapper`. If the behaviour for the bound function also needs to be overridden, a derived version of this class will also need to be created. The derived custom function wrapper will then need to indicate that this second type should be used when creating the bound function wrapper, rather than the default. This is done via the `__bound_function_wrapper__` attribute of the class.

```
class CustomBoundFunctionWrapper(wrapt.BoundFunctionWrapper):

    def __init__(self, *args, **kwargs):
        self._self_attribute = self._self_parent._self_attribute
        super(CustomBoundFunctionWrapper, self).__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self._self_attribute:
            ...
        return super(CustomBoundFunctionWrapper, self).__call__(*args, **kwargs)

class CustomFunctionWrapper(wrapt.FunctionWrapper):

    __bound_function_wrapper__ = CustomBoundFunctionWrapper

    def __init__(self, wrapped, wrapper, attribute):
        super(CustomFunctionWrapper, self).__init__(wrapped, wrapper)
        self._self_attribute = attribute
```

The set of arguments used to initialize an instance of a bound function wrapper object should be treated as a private implementation detail. This means that if a custom bound function wrapper needs to implement an `__init__()` method, it should pass through all arguments as `*args` and `**kwargs`. It should not use specific named parameters.

If the instance of the custom bound function wrapper needs to access any special attributes originally supplied to the custom function wrapper when created, it should use `self._self_parent` to access the parent object to retrieve

them.

Alternatively, it would be necessary to define the custom bound function wrapper type in a function closure and assign `__bound_function_wrapper__` dynamically against the instance of the custom bound function wrapper.

```
def custom_bound_function_wrapper(attribute):

    class CustomBoundFunctionWrapper(wrapt.BoundFunctionWrapper):

        def __init__(self, *args, **kwargs):
            self._self_attribute = attribute
            super(CustomBoundFunctionWrapper, self).__init__(*args, **kwargs)

        def __call__(self, *args, **kwargs):
            if self._self_attribute:
                ...
            return super(CustomBoundFunctionWrapper, self).__call__(*args, **kwargs)

class CustomFunctionWrapper(wrapt.FunctionWrapper):

    def __init__(self, wrapped, wrapper, attribute):
        super(CustomFunctionWrapper, self).__init__(wrapped, wrapper)
        self._self_attribute = attribute
        self.__bound_function_wrapper__ = custom_bound_function_wrapper(attribute)
```

2.4 Assorted Examples

This document provides various examples of decorators often described elsewhere, to exhibit what can be done with decorators using the **wrapt** module, for the purpose of comparison.

2.4.1 Thread Synchronization

Note: The final variant of the `synchronized` decorator described here is available within the **wrapt** package as `wrapt.synchronized`.

Synchronization decorators are a simplified way of adding thread locking to functions, methods, instances of classes or a class type. They work by associating a thread mutex with a specific context and when a function is called the lock is acquired prior to the call and then released once the function returns.

The simplest example of a decorator for synchronization is one where the lock is explicitly provided when the decorator is applied to a function. By being supplied explicitly, it is up to the user of the decorator to determine what context the lock applies to. For example, a lock may be applied to a single function, a group of functions, or a class.

As the lock needs to be supplied when the decorator is applied to the function we need to use a function closure as a means of supplying the argument to the decorator.

```
def synchronized(lock):
    @wrapt.decorator
    def _wrapper(wrapped, instance, args, kwargs):
        with lock:
            return wrapped(*args, **kwargs)
    return _wrapper
```

(continues on next page)

(continued from previous page)

```
import threading

lock = threading.RLock()

@synchronized(lock)
def function():
    pass

class Class(object):

    @synchronized(lock)
    def function(self):
        pass
```

Note that the recursive lock `threading.RLock` is used to ensure that recursive calls, or calls to another synchronized function associated with the same lock, doesn't cause a deadlock.

An alternative to requiring the lock be supplied when the decorator is applied to a function, is to associate a lock automatically with the wrapped function. That is, rather than require the lock be passed in explicitly, create one on demand and attach it to the wrapped function.

```
@wrapt.decorator
def synchronized(wrapped, instance, args, kwargs):
    # Retrieve the lock from the wrapped function.

    lock = vars(wrapped).get('_synchronized_lock', None)

    if lock is None:
        # There was no lock yet associated with the function so we
        # create one and associate it with the wrapped function.
        # We use ``dict.setdefault()`` as a means of ensuring that
        # only one thread gets to set the lock if multiple threads
        # do this at the same time. This may mean redundant lock
        # instances will get thrown away if there is a race to set
        # it, but all threads would still get back the same one lock.

        lock = vars(wrapped).setdefault('_synchronized_lock',
                threading.RLock())

    with lock:
        return wrapped(*args, **kwargs)

@synchronized
def function():
    pass
```

This avoids the need for an instance of a lock to be passed in explicitly when the decorator is being applied to a function, but it now means that all decorated methods of a class will have a distinct lock.

A more severe issue in both these approaches is that locks on each method work across all instances of the class where as what we really want is a lock per instance of a class for all methods of the class decorated with the `@synchronized` decorator.

To address this, we can use the fact that the decorator wrapper function is passed the `instance` and so can determine when the function is being invoked on an instance of a class and that it is not a normal function call. In this case we can associate the lock with the instance instead.

```
@wrapt.decorator
def synchronized(wrapped, instance, args, kwargs):
    # Use the instance as the context if function was bound.

    if instance is not None:
        context = vars(instance)
    else:
        context = vars(wrapped)

    # Retrieve the lock for the specific context.

    lock = context.get('_synchronized_lock', None)

    if lock is None:
        # There was no lock yet associated with the function so we
        # create one and associate it with the wrapped function.
        # We use ``dict.setdefault()`` as a means of ensuring that
        # only one thread gets to set the lock if multiple threads
        # do this at the same time. This may mean redundant lock
        # instances will get thrown away if there is a race to set
        # it, but all threads would still get back the same one lock.

        lock = context.setdefault('_synchronized_lock',
                threading.RLock())

    with lock:
        return wrapped(*args, **kwargs)

@synchronized
def function():
    pass
```

Now we actually have two scenarios that match for where `instance` is not `None`. One will be where an instance method is being called on a class, which is what we are targeting in this case. We will also have `instance` being a value other than `None` for the case where a class method is called. For this case `instance` will be a reference to the class type.

Having the lock being associated with the class type for class methods is entirely reasonable, but a problem presents. That is that `vars(instance)` where `instance` is a class type, actually returns a `dictproxy` and not a `dict`. As a `dictproxy` is effectively read only, it is not possible to associate the lock with it.

A similar problem also occurs where `instance` is `None` but `wrapped` is a class type. That is, if the decorator was applied to a class. The result is that the above technique will not work in these two cases.

The only way that it is possible to add attributes to a class type is to use `setattr`, either explicitly or via direct attribute assignment. Although this allows us to add attributes to a class, there is no equivalent to `dict.setdefault()`, so we loose the ability to add the attribute which will hold the lock atomically.

To get around this problem, we need to use an intermediary meta lock which gates the attempt to associate a lock with a specific context. This meta lock itself still needs to be created somehow, so what we do now is use the `dict.setdefault()` trick against the decorator itself and use it as the place to store the meta lock.

```
@wrapt.decorator
def synchronized(wrapped, instance, args, kwargs):
    # Use the instance as the context if function was bound.

    if instance is not None:
        context = instance
```

(continues on next page)

(continued from previous page)

```
    else:
        context = wrapped

    # Retrieve the lock for the specific context.

    lock = vars(context).get('_synchronized_lock', None)

    if lock is None:
        # There is no existing lock defined for the context we
        # are dealing with so we need to create one. This needs
        # to be done in a way to guarantee there is only one
        # created, even if multiple threads try and create it at
        # the same time. We can't always use the setdefault()
        # method on the __dict__ for the context. This is the
        # case where the context is a class, as __dict__ is
        # actually a dictproxy. What we therefore do is use a
        # meta lock on this wrapper itself, to control the
        # creation and assignment of the lock attribute against
        # the context.

        meta_lock = vars(synchronized).setdefault(
                '_synchronized_meta_lock', threading.Lock())

        with meta_lock:
            # We need to check again for whether the lock we want
            # exists in case two threads were trying to create it
            # at the same time and were competing to create the
            # meta lock.

            lock = vars(context).get('_synchronized_lock', None)

            if lock is None:
                lock = threading.RLock()
                setattr(context, '_synchronized_lock', lock)

    with lock:
        return wrapped(*args, **kwargs)
```

This means lock creation is all automatic, with an appropriate lock created for the different contexts the decorator is used in.

```
@synchronized # lock bound to function1
def function1():
    pass

@synchronized # lock bound to function2
def function2():
    pass

@synchronized # lock bound to Class
class Class(object):

    @synchronized # lock bound to instance of Class
    def function_im(self):
        pass
```

(continues on next page)

(continued from previous page)

```
    @synchronized # lock bound to Class
    @classmethod
    def function_cm(cls):
        pass

    @synchronized # lock bound to function_sm
    @staticmethod
    def function_sm():
        pass
```

Specifically, when the decorator is used on a normal function or static method, a unique lock will be associated with each function. For the case of instance methods, the lock will be against the instance. Finally, for class methods and a decorator against an actual class, the lock will be against the class type.

One requirement with this approach though is that only the execution of a whole function can be synchronized. In Java where a similar mechanism exists, it is also possible to have synchronized statements. In Python one can emulate synchronized statements by using the 'with' statement in conjunction with a lock. The trick with that is that if using it within a method of a class, we want to be able to use the same lock as that which is being applied to synchronized methods of the class. In effect we want to be able to do the following.

```
class Class(object):

    @synchronized
    def function_im_1(self):
        pass

    def function_im_2(self):
        with synchronized(self):
            pass
```

In other words we want the decorator function to serve a dual role of being able to decorate a function to make it synchronized, but also return a context manager for the lock for a specific context so that it can be used with the 'with' statement.

Because of this dual requirement, we actually need to partly side step `wrapt.decorator` and drop down to using the underlying `FunctionWrapper` class that it uses to implement decorators. Specifically, we need to create a derived version of `FunctionWrapper` which converts it into a context manager, but at the same time can still be used as a decorator as before.

```
def synchronized(wrapped):
    def _synchronized_lock(context):
        # Attempt to retrieve the lock for the specific context.

        lock = vars(context).get('_synchronized_lock', None)

        if lock is None:
            # There is no existing lock defined for the context we
            # are dealing with so we need to create one. This needs
            # to be done in a way to guarantee there is only one
            # created, even if multiple threads try and create it at
            # the same time. We can't always use the setdefault()
            # method on the __dict__ for the context. This is the
            # case where the context is a class, as __dict__ is
            # actually a dictproxy. What we therefore do is use a
            # meta lock on this wrapper itself, to control the
            # creation and assignment of the lock attribute against
```

(continues on next page)

(continued from previous page)

```
            # the context.

            meta_lock = vars(synchronized).setdefault(
                    '_synchronized_meta_lock', Lock())

            with meta_lock:
                # We need to check again for whether the lock we want
                # exists in case two threads were trying to create it
                # at the same time and were competing to create the
                # meta lock.

                lock = vars(context).get('_synchronized_lock', None)

                if lock is None:
                    lock = RLock()
                    setattr(context, '_synchronized_lock', lock)

        return lock

    def _synchronized_wrapper(wrapped, instance, args, kwargs):
        # Execute the wrapped function while the lock for the
        # desired context is held. If instance is None then the
        # wrapped function is used as the context.

        with _synchronized_lock(instance or wrapped):
            return wrapped(*args, **kwargs)

    class _FinalDecorator(FunctionWrapper):

        def __enter__(self):
            self._self_lock = _synchronized_lock(self.__wrapped__)
            self._self_lock.acquire()
            return self._self_lock

        def __exit__(self, *args):
            self._self_lock.release()

    return _FinalDecorator(wrapped=wrapped, wrapper=_synchronized_wrapper)
```

When used in this way, the more typical use case would be to synchronize against the class instance, but if needing to synchronize with the work of a class method from an instance method, it could also be done against the class itself.

```
class Class(object):

    @synchronized
    @classmethod
    def function_cm(cls):
        pass

    def function_im(self):
        with synchronized(Class):
            pass
```

If wishing to have more than one normal function synchronize on the same object, then it is possible to have the synchronization be against a data structure which they all manipulate.

```
class Data(object):
    pass

data = Data()

def function_1():
    with synchronized(data):
        pass

def function_2():
    with synchronized(data):
        pass
```

In doing this you would be restricted to using a data structure to which new attributes can be added, such that the hidden lock can be added. This means for example, you could not do this with a dictionary. It also means you can't just decorate the whole function.

What would perhaps be better is to return back to having the `synchronized` decorator allow an actual lock object to be supplied when the decorator is being applied to a function. Being able to do this though would be optional and if not done the lock would be associated with the appropriate context of the wrapped function.

```
lock = threading.RLock()

@synchronized(lock)
def function_1():
    pass

@synchronized(lock)
def function_2():
    pass
```

This requires what the decorator accepts to be overloaded and so may be frowned on by some, but the implementation would be as follows.

```
def synchronized(wrapped):
    # Determine if being passed an object which is a synchronization
    # primitive. We can't check by type for Lock, RLock, Semaphore etc,
    # as the means of creating them isn't the type. Therefore use the
    # existence of acquire() and release() methods. This is more
    # extensible anyway as it allows custom synchronization mechanisms.

    if hasattr(wrapped, 'acquire') and hasattr(wrapped, 'release'):
        # We remember what the original lock is and then return a new
        # decorator which accesses and locks it. When returning the new
        # decorator we wrap it with an object proxy so we can override
        # the context manager methods in case it is being used to wrap
        # synchronized statements with a 'with' statement.

        lock = wrapped

        @decorator
        def _synchronized(wrapped, instance, args, kwargs):
            # Execute the wrapped function while the original supplied
            # lock is held.

            with lock:
                return wrapped(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        class _PartialDecorator(ObjectProxy):

            def __enter__(self):
                lock.acquire()
                return lock

            def __exit__(self, *args):
                lock.release()

        return _PartialDecorator(wrapped=_synchronized)

    # Following only apply when the lock is being created
    # automatically based on the context of what was supplied. In
    # this case we supply a final decorator, but need to use
    # FunctionWrapper directly as we want to derive from it to add
    # context manager methods in case it is being used to wrap
    # synchronized statements with a 'with' statement.

    def _synchronized_lock(context):
        # Attempt to retrieve the lock for the specific context.

        lock = vars(context).get('_synchronized_lock', None)

        if lock is None:
            # There is no existing lock defined for the context we
            # are dealing with so we need to create one. This needs
            # to be done in a way to guarantee there is only one
            # created, even if multiple threads try and create it at
            # the same time. We can't always use the setdefault()
            # method on the __dict__ for the context. This is the
            # case where the context is a class, as __dict__ is
            # actually a dictproxy. What we therefore do is use a
            # meta lock on this wrapper itself, to control the
            # creation and assignment of the lock attribute against
            # the context.

            meta_lock = vars(synchronized).setdefault(
                    '_synchronized_meta_lock', Lock())

            with meta_lock:
                # We need to check again for whether the lock we want
                # exists in case two threads were trying to create it
                # at the same time and were competing to create the
                # meta lock.

                lock = vars(context).get('_synchronized_lock', None)

                if lock is None:
                    lock = RLock()
                    setattr(context, '_synchronized_lock', lock)

        return lock

    def _synchronized_wrapper(wrapped, instance, args, kwargs):
        # Execute the wrapped function while the lock for the
        # desired context is held. If instance is None then the
```

(continues on next page)

(continued from previous page)

```
        # wrapped function is used as the context.

        with _synchronized_lock(instance or wrapped):
            return wrapped(*args, **kwargs)

    class _FinalDecorator(FunctionWrapper):

        def __enter__(self):
            self._self_lock = _synchronized_lock(self.__wrapped__)
            self._self_lock.acquire()
            return self._self_lock

        def __exit__(self, *args):
            self._self_lock.release()

    return _FinalDecorator(wrapped=wrapped, wrapper=_synchronized_wrapper)
```

As well as normal functions, this can be used with methods of classes as well. Because though the lock object has to be available at the time the class definition is being created, it can only be used to refer to a lock which is the same across the whole class, or one which is at global scope.

```
class Class(object):
    lock1 = threading.RLock()
    lock2 = threading.RLock()

    @synchronized(lock1)
    @classmethod
    def function_cm_1(cls):
        pass

    @synchronized(lock1)
    def function_im_1(self):
        pass

    @synchronized(lock2)
    @classmethod
    def function_cm_2(cls):
        pass
```

The alternative is to use `synchronized` as a context manager and pass the lock in at that time.

```
class Class(object):

    def __init__(self):
        self.lock1 = threading.RLock()

    def function_im(self):
        with synchronized(self.lock1):
            pass
```

This is actually the same as using the 'with' statement directly on the lock, but it you want to get carried away and have all the code look more or less uniform, it is possible.

One benefit of being able to pass the lock in explicitly, is that you can override the default lock type used, which is `threading.RLock`. Any synchronization primitive can be supplied so long as it provides a `acquire()` and `release()` method. This includes being able to pass in your own custom class objects with such methods which do something appropriate.

```
semaphore = threading.Semaphore(2)

@synchronized(semaphore)
def function():
    pass
```

2.5 Decorator Benchmarks

The **wrapt** module ensures that your decorators will work in all situations. The implementation therefore does not take the shortcuts that people usually take with decorators of using function closures. Instead it implements the wrappers as a class, which also acts as a descriptor. Ensuring correctness though does come at an additional cost in runtime overhead. The following attempts to quantify what that overhead is and compare it to other solutions typically used.

Results were collected under MacOS X Mountain Lion on a 2012 model MacBook Pro, running with Python 2.7.

2.5.1 Undecorated Calls

These tests provide a baseline for comparing decorated functions against a normal undecorated function call.

Test Code:

```
def function1():
    pass

class Class(object):

    def function1(self):
        pass

    @classmethod
    def function1cm(cls):
        pass

    @staticmethod
    def function1sm():
        pass
```

Test Results:

```
$ python -m timeit -s 'import benchmarks' 'benchmarks.function1()'
10000000 loops, best of 3: 0.132 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function1()'
10000000 loops, best of 3: 0.143 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function1cm()'
1000000 loops, best of 3: 0.217 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function1cm()'
10000000 loops, best of 3: 0.159 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function1sm()'
1000000 loops, best of 3: 0.199 usec per loop
```

(continues on next page)

(continued from previous page)

```
$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function1sm()'
10000000 loops, best of 3: 0.13 usec per loop
```

Note that differences between calling the class and static methods via the class vs the instance are possibly more to do with needing to traverse the dotted path.

2.5.2 Function Closures

These tests provide results for decorated functions where the decorators are implemented using function closures.

Test Code:

```
def wrapper2(func):
    def _wrapper2(*args, **kwargs):
        return func(*args, **kwargs)
    return _wrapper2

@wrapper2
def function2():
    pass

class Class(object):

    @wrapper2
    def function2(self):
        pass

    @classmethod
    @wrapper2
    def function2cmi(cls):
        pass

    @staticmethod
    @wrapper2
    def function2smi():
        pass
```

Test Results:

```
$ python -m timeit -s 'import benchmarks' 'benchmarks.function2()'
1000000 loops, best of 3: 0.326 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function2()'
1000000 loops, best of 3: 0.382 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function2cmi()'
1000000 loops, best of 3: 0.46 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function2cmi()'
1000000 loops, best of 3: 0.384 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function2smi()'
1000000 loops, best of 3: 0.389 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function2smi()'
1000000 loops, best of 3: 0.319 usec per loop
```

Note that decorators implemented as function closures cannot be added around staticmethod and classmethod decorators and must be added inside of those decorators.

2.5.3 wrapt.decorator

These tests provides results for decorated functions where the decorators are implemented using the **wrapt** module. Separate results are provided for when using the C extension and when using the pure Python implementation.

Test Code:

```
@wrapt.decorator
def wrapper3(wrapped, instance, args, kwargs):
    return wrapped(*args, **kwargs)

@wrapper3
def function3():
    pass

class Class(object):

    @wrapper3
    def function3(self):
        pass

    @wrapper3
    @classmethod
    def function3cmo(cls):
        pass

    @classmethod
    @wrapper3
    def function3cmi(cls):
        pass

    @wrapper3
    @staticmethod
    def function3smo():
        pass

    @staticmethod
    @wrapper3
    def function3smi():
        pass
```

Test Results (C Extension):

```
$ python -m timeit -s 'import benchmarks' 'benchmarks.function3()'
1000000 loops, best of 3: 0.382 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3()'
1000000 loops, best of 3: 0.836 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3cmo()'
1000000 loops, best of 3: 1.11 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3cmo()'
1000000 loops, best of 3: 1.06 usec per loop
```

(continues on next page)

(continued from previous page)

```
$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3cmi()'
1000000 loops, best of 3: 0.535 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3cmi()'
1000000 loops, best of 3: 0.455 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3smo()'
1000000 loops, best of 3: 1.37 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3smo()'
1000000 loops, best of 3: 1.31 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3smi()'
1000000 loops, best of 3: 0.453 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3smi()'
1000000 loops, best of 3: 0.378 usec per loop
```

Note that results for where the decorator is inside that of the classmethod decorator is quite a bit less than that where it is outside. This due to a potential bug in Python whereby it doesn't apply the descriptor protocol to what the classmethod decorator wraps. Instead it is executing a straight function call, which has less overhead.

Test Results (Pure Python):

```
$ python -m timeit -s 'import benchmarks' 'benchmarks.function3()'
1000000 loops, best of 3: 0.771 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3()'
100000 loops, best of 3: 6.67 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3cmo()'
100000 loops, best of 3: 6.89 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3cmo()'
100000 loops, best of 3: 6.77 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3cmi()'
1000000 loops, best of 3: 0.911 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3cmi()'
1000000 loops, best of 3: 0.863 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3smo()'
100000 loops, best of 3: 7.26 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3smo()'
100000 loops, best of 3: 7.17 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function3smi()'
1000000 loops, best of 3: 0.835 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function3smi()'
1000000 loops, best of 3: 0.774 usec per loop
```

Note that results for where the decorator is inside that of the classmethod decorator is quite a bit less than that where it is outside. This due to a potential bug in Python whereby it doesn't apply the descriptor protocol to what the

classmethod decorator wraps. Instead it is executing a straight function call, which has less overhead.

2.5.4 decorator.decorator

These tests provides results for decorated functions where the decorators are implemented using the **decorator** module available from PyPi.

Test Code:

```
@decorator.decorator
def wrapper4(wrapped, *args, **kwargs):
    return wrapped(*args, **kwargs)

@wrapper4
def function4():
    pass

class Class(object):

    @wrapper4
    def function4(self):
        pass

    @classmethod
    @wrapper4
    def function4cmi(cls):
        pass

    @staticmethod
    @wrapper4
    def function4smi():
        pass
```

Test Results:

```
$ python -m timeit -s 'import benchmarks' 'benchmarks.function4()'
1000000 loops, best of 3: 0.465 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function4()'
1000000 loops, best of 3: 0.537 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function4cmi()'
1000000 loops, best of 3: 0.606 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function4cmi()'
1000000 loops, best of 3: 0.533 usec per loop

$ python -m timeit -s 'import benchmarks' 'benchmarks.Class.function4smi()'
1000000 loops, best of 3: 0.532 usec per loop

$ python -m timeit -s 'import benchmarks; c=benchmarks.Class()' 'c.function4smi()'
1000000 loops, best of 3: 0.456 usec per loop
```

Note that decorators implemented using the decorator module cannot be added around staticmethod and classmethod decorators and must be added inside of those decorators.

2.6 Running Unit Tests

Unit tests are located in the `tests` directory.

To test both the pure Python and C extension module based implementations, run the command:

```
tox
```

By default tests are run for Python 2.7, 3.4-3.7 and PyPy, with and without the C extensions.

```
py27-without-extensions
py34-without-extensions
py35-without-extensions
py36-without-extensions
py37-without-extensions

py27-with-extensions
py34-with-extensions
py35-with-extensions
py36-with-extensions
py37-with-extensions

pypy-without-extensions
```

If wishing to run tests for a specific Python combination you can run `tox` with the `-e` option.

```
tox -e py37-with-extensions
```

If adding more tests and you need to add a test which is Python 2 or Python 3 specific, then end the name of the Python code file as `_py2.py` or `_py3.py` appropriately.

For further options refer to the documentation for `tox`.

2.6.1 Coverage

Coverage is collected and sent to Coveralls when running the tests automatically in Travis CI. To collect and view coverage results locally, here's the sequence of commands:

```
export COVERAGE_CMD="coverage run -m"
export COVERAGE_DEP=coverage
tox
coverage combine
coverage html --ignore-errors
```

At this point there's a directly called `htmlcov` with the formatted results.

2.7 Release Notes

2.7.1 Version 1.12.1

Bugs Fixed

- Applying a function wrapper to a static method of a class using the `wrap_function_wrapper()` function, or wrapper for the same, wasn't being done correctly when the static method was the immediate child of the

target object. It was working when the name path had multiple name components. A failure would subsequently occur when the static method was called via an instance of the class, rather than the class.

2.7.2 Version 1.12.0

Features Changed

- Provided that you only want to support Python 3.7, when deriving from a base class which has a decorator applied to it, you no longer need to access the true type of the base class using `__wrapped__` in the inherited class list of the derived class.

Bugs Fixed

- When using the `synchronized` decorator on instance methods of a class, if the class declared special methods to override the result for when the class instance was tested as a boolean so that it returned `False` all the time, the synchronized method would fail when called.
- When using an adapter function to change the signature of the decorated function, `inspect.signature()` was returning the wrong signature when an instance method was inspected by accessing the method via the class type.

2.7.3 Version 1.11.2

Bugs Fixed

- Fix possible crash when garbage collection kicks in when invoking a destructor of wrapped object.

2.7.4 Version 1.11.1

Bugs Fixed

- Fixed memory leak in C extension variant of `PartialCallableObjectProxy` class introduced in 1.11.0, when it was being used to perform binding, when a call of an instance method was made through the class type, and the self object passed explicitly as first argument.
- The C extension variant of the `PartialCallableObjectProxy` class introduced in 1.11.0, which is a version of `functools.partial` which correctly handles binding when applied to methods of classes, couldn't be used when no positional arguments were supplied.
- When the C extension variant of `PartialCallableObjectProxy` was used and multiple positional arguments were supplied, the first argument would be replicated and used to all arguments, instead of correct values, when the partial was called.
- When the C extension variant of `PartialCallableObjectProxy` was used and keyword arguments were supplied, it would fail as was incorrectly using the positional arguments where the keyword arguments should have been used.

2.7.5 Version 1.11.0

Bugs Fixed

- When using arithmetic operations through a proxy object, checks about the types of arguments were not being performed correctly, which could result in an exception being raised to indicate that a proxy object had not been initialised when in fact the argument wasn't even an instance of a proxy object.

Because an incorrect cast in C level code was being performed and an attribute in memory checked on the basis of it being a type different to what it actually was, technically it may have resulted in a process crash if the size of the object was smaller than the type being casted to.

- The `__complex__()` special method wasn't implemented and using `complex()` on a proxy object would give wrong results or fail.
- When using the C extension, if an exception was raised when using inplace or, ie., `|=`, the error condition wasn't being correctly propagated back which would result in an exception showing up as wrong location in subsequent code.
- Type of `long` was used instead of `Py_hash_t` for Python 3.3+. This caused compiler warnings on Windows, which depending on what locale was set to, would cause pip to fail when installing the package.
- If calling `Class.instancemethod` and passing `self` explicitly, the ability to access `__name__` and `__module__` on the final bound method were not preserved. This was due to a `partial` being used for this special case, and it doesn't preserve introspection.
- Fixed typo in the getter property of `ObjectProxy` for accessing `__annotations__`. Appeared that it was still working as would fall back to using generic `__getattr__()` to access attribute on wrapped object.

Features Changed

- Dropped support for Python 2.6 and 3.3.
- If `copy.copy()` or `copy.deepcopy()` is used on an instance of the `ObjectProxy` class, a `NotImplementedError` exception is raised, with a message indicating that the object proxy must implement the `__copy__()` or `__deepcopy__()` method. This is in place of the default `TypeError` exception with message indicating a pickle error.
- If `pickle.dump()` or `pickle.dumps()` is used on an instance of the `ObjectProxy` class, a `NotImplementedError` exception is raised, with a message indicating that the object proxy must implement the `__reduce_ex__()` method. This is in place of the default `TypeError` exception with message indicating a pickle error.

2.7.6 Version 1.10.11

Bugs Fixed

- When wrapping a `@classmethod` in a class used as a base class, when the method was called via the derived class type, the base class type was being passed for the `cls` argument instead of the derived class type through which the call was made.

New Features

- The C extension can be disabled at runtime by setting the environment variable `WRAPT_DISABLE_EXTENSIONS`. This may be necessary where there is currently a difference in behaviour between pure Python implementation and C extension and the C extension isn't having the desired result.

2.7.7 Version 1.10.10

Features Changed

- Added back missing description and categorisations when releasing to PyPi.

2.7.8 Version 1.10.9

Bugs Fixed

- Code for `inspect.getargspec()` when using Python 2.6 was missing import of `sys` module.

2.7.9 Version 1.10.8

Bugs Fixed

- Ensure that `inspect.getargspec()` is only used with Python 2.6 where required, as function has been removed in Python 3.6.

2.7.10 Version 1.10.7

Bugs Fixed

- The mod operator '%' was being incorrectly proxied in Python variant of object proxy to the xor operator '^'.

2.7.11 Version 1.10.6

Bugs Fixed

- Registration of post import hook would fail with an exception if registered after another import hook for the same target module had been registered and the target module also imported.

New Features

- Support for testing with Travis CI added to repository.

2.7.12 Version 1.10.5

Bugs Fixed

- Post import hook discovery was not working correctly where multiple target modules were registered in the same entry point list. Only the callback for the last would be called regardless of the target module.
- If a `WeakFunctionProxy` wrapper was used around a method of a class which was decorated using a wrapt decorator, the decorator wasn't being invoked when the method was called via the weakref proxy.

Features Changed

- The `register_post_import_hook()` function, modelled after the function of the same name in PEP-369 has been extended to allow a string name to be supplied for the import hook. This needs to be of the form `module::function` and will result in an import hook proxy being used which will only load and call the function of the specified moduled when the import hook is required. This avoids needing to load the code needed to operate on the target module unless required.

2.7.13 Version 1.10.4

Bugs Fixed

- Fixup botched package version number from 1.10.3 release.

2.7.14 Version 1.10.3

Bugs Fixed

- Post import hook discovery from third party modules declared via `setuptools` entry points was failing due to typo in temporary variable name. Also added the `discover_post_import_hooks()` to the public API as was missing.

Features Changed

- To ensure parity between pure Python and C extension variants of the `ObjectProxy` class, allow the `__wrapped__` attribute to be set in a derived class when the `ObjectProxy.__init__()` method hasn't been called.

2.7.15 Version 1.10.2

Bugs Fixed

- When creating a derived `ObjectProxy`, if the base class `__init__()` method wasn't called and the `__wrapped__` attribute was accessed, in the pure Python implementation a recursive call of `__getattr__()` would occur and the maximum stack depth would be reached and an exception raised.
- When creating a derived `ObjectProxy`, if the base class `__init__()` method wasn't called, in the C extension implementation, if that instance was then used in a binary arithmetic operation the process would crash.

2.7.16 Version 1.10.1

Bugs Fixed

- When using `FunctionWrapper` around a method of an existing instance of a class, rather than on the type, then a memory leak could occur in two different scenarios.

  The first issue was that wrapping a method on an instance of a class was causing an unwanted reference to the class meaning that if the class type was transient, such as it is being created inside of a function call, the type object would leak.

  The second issue was that wrapping a method on an instance of a class and then calling the method was causing an unwanted reference to the instance meaning that if the instance was transient, it would leak.

  This was only occurring when the C extension component for the `wrapt` module was being used.

2.7.17 Version 1.10.0

New Features

- When specifying an adapter for a decorator, it is now possible to pass in, in addition to passing in a callable, a tuple of the form which is returned by `inspect.getargspec()`, or a string of the form which is returned by `inspect.formatargspec()`. In these two cases the decorator will automatically compile a stub function to use as the adapter. This eliminates the need for a caller to generate the stub function if generating the signature on the fly.

```
def argspec_factory(wrapped):
    argspec = inspect.getargspec(wrapped)

    args = argspec.args[1:]
    defaults = argspec.defaults and argspec.defaults[-len(argspec.args):]
```

(continues on next page)

(continued from previous page)

```
    return inspect.ArgSpec(args, argspec.varargs,
            argspec.keywords, defaults)

def session(wrapped):
    @wrapt.decorator(adapter=argspec_factory(wrapped))
    def _session(wrapped, instance, args, kwargs):
        with transaction() as session:
            return wrapped(session, *args, **kwargs)

    return _session(wrapped)
```

This mechanism and the original mechanism to pass a function, meant that the adapter function had to be created in advance. If the adapter needed to be generated on demand for the specific function to be wrapped, then it would have been necessary to use a closure around the definition of the decorator as above, such that the generator could be passed in.

As a convenience, instead of using such a closure, it is also now possible to write:

```
def argspec_factory(wrapped):
    argspec = inspect.getargspec(wrapped)

    args = argspec.args[1:]
    defaults = argspec.defaults and argspec.defaults[-len(argspec.args):]

    return inspect.ArgSpec(args, argspec.varargs,
            argspec.keywords, defaults)

@wrapt.decorator(adapter=wrapt.adapter_factory(argspec_factory))
def _session(wrapped, instance, args, kwargs):
    with transaction() as session:
        return wrapped(session, *args, **kwargs)
```

The result of `wrapt.adapter_factory()` will be recognised as indicating that the creation of the adapter is to be deferred until the decorator is being applied to a function. The factory function for generating the adapter function or specification on demand will be passed the function being wrapped by the decorator.

If wishing to create a library of routines for generating adapter functions or specifications dynamically, then you can do so by creating classes which derive from `wrapt.AdapterFactory` as that is the type which is recognised as indicating lazy evaluation of the adapter function. For example, `wrapt.adapter_factory()` is itself implemented as:

```
class DelegatedAdapterFactory(wrapt.AdapterFactory):
    def __init__(self, factory):
        super(DelegatedAdapterFactory, self).__init__()
        self.factory = factory
    def __call__(self, wrapped):
        return self.factory(wrapped)

adapter_factory = DelegatedAdapterFactory
```

Bugs Fixed

- The `inspect.signature()` function was only added in Python 3.3. Use fallback when doesn't exist and on Python 3.2 or earlier Python 3 versions.

  Note that testing is only performed for Python 3.3+, so it isn't actually known if the `wrapt` package works on Python 3.2.

2.7.18 Version 1.9.0

Features Changed

- When using `wrapt.wrap_object()`, it is now possible to pass an arbitrary object in addition to a module object, or a string name identifying a module. Similar for underlying `wrapt.resolve_path()` function.

Bugs Fixed

- It is necessary to proxy the special `__weakref__` attribute in the pure Python object proxy else using `inspect.getmembers()` on a decorator class will fail.
- The `FunctionWrapper` class was not passing through the instance correctly to the wrapper function when it was applied to a method of an existing instance of a class.
- The `FunctionWrapper` was not always working when applied around a method of a class type by accessing the method to be wrapped using `getattr()`. Instead it is necessary to access the original unbound method from the class `__dict__`. Updated the `FunctionWrapper` to work better in such situations, but also modify `resolve_path()` to always grab the class method from the class `__dict__` when wrapping methods using `wrapt.wrap_object()` so wrapping is more predictable. When doing monkey patching `wrapt.wrap_object()` should always be used to ensure correct operation.
- The `AttributeWrapper` class used internally to the function `wrap_object_attribute()` had wrongly named the `__delete__` method for the descriptor as `__del__`.

2.7.19 Version 1.8.0

Features Changed

- Previously using @wrapt.decorator on a class type didn't really yield anything which was practically useful. This is now changed and when applied to a class an instance of the class will be automatically created to be used as the decorator wrapper function. The requirement for this is that the __call__() method be specified in the style as would be done for the decorator wrapper function.

```
@wrapt.decorator
class mydecoratorclass(object):
    def __init__(self, arg=None):
        self.arg = arg
    def __call__(self, wrapped, instance, args, kwargs):
        return wrapped(*args, **kwargs)

@mydecoratorclass
def function():
    pass
```

If the resulting decorator class is to be used with no arguments, the __init__() method of the class must have all default arguments. These arguments can be optionally supplied though, by using keyword arguments to the resulting decorator when applied to the function to be decorated.

```
@mydecoratorclass(arg=1)
def function():
    pass
```

2.7.20 Version 1.7.0

New Features

- Provide wrapt.getcallargs() for determining how arguments mapped to a wrapped function. For Python 2.7 this is actually inspect.getcallargs() with a local copy being used in the case of Python 2.6.
- Added wrapt.wrap_object_attribute() as a way of wrapping or otherwise modifying the result of trying to access the attribute of an object instance. It works by adding a data descriptor with the same name as the attribute, to the class type, allowing reading of the attribute to be intercepted. It does not affect updates to or deletion of the attribute.

Bugs Fixed

- Need to explicitly proxy special methods __bytes__(), __reversed__() and __round__() as they are only looked up on the class type and not the instance, so can't rely on __getattr__() fallback.
- Raise more appropriate TypeError, with corresponding message, rather than IndexError, when a decorated instance or class method is called via the class but the required 1st argument of the instance or class is not supplied.

2.7.21 Version 1.6.0

Bugs Fixed

- The ObjectProxy class would return that the __call__() method existed even though the wrapped object didn't have one. Similarly, callable() would always return True even if the wrapped object was not callable.

  This resulted due to the existence of the __call__() method on the wrapper, required to support the possibility that the wrapped object may be called via the proxy object even if it may not turn out that the wrapped object was callable.

  Because checking for the existence of a __call__() method or using callable() can sometimes be used to indirectly infer the type of an object, this could cause issues. To ensure that this now doesn't occur, the ability to call a wrapped object via the proxy object has been removed from ObjectProxy. Instead, a new class CallableObjectProxy is now provided, with it being necessary to make a conscious choice as to which should be used based on whether the object to be wrapped is in fact callable.

  Note that neither before this change, or with the introduction of the class CallableObjectProxy, does the object proxy perform binding. If binding behaviour is required it still needs to be implemented explicitly to match the specific requirements of the use case. Alternatively, the FunctionWrapper class should be used which does implement binding, but also enforces a wrapper mechanism for manipulating what happens at the time of the call.

2.7.22 Version 1.5.1

Bugs Fixed

- Instance method locking for the synchronized decorator was not correctly locking on the instance but the class, if a synchronized class method had been called prior to the synchronized instance method.

2.7.23 Version 1.5.0

New Features

- Enhanced @wrapt.transient_function_wrapper so it can be applied to instance methods and class methods with the self/cls argument being supplied correctly. This allows instance and class methods to be used for this type of decorator, with the instance or class type being able to be used to hold any state required for the decorator.

Bugs Fixed

- If the wrong details for a function to be patched was given to the decorator @wrapt.transient_function_wrapper, the exception indicating this was being incorrectly swallowed up and mutating to a different more obscure error about local variable being access before being set.

2.7.24 Version 1.4.2

Bugs Fixed

- A process could crash if the C extension module was used and when using the ObjectProxy class a reference count cycle was created that required the Python garbage collector to kick in to break the cycle. This was occurring as the C extension had not implemented GC support in the ObjectProxy class correctly.

2.7.25 Version 1.4.1

Bugs Fixed

- Overriding __wrapped__ attribute directly on any wrapper more than once could cause corruption of memory due to incorrect reference count decrement.

2.7.26 Version 1.4.0

New Features

- Enhanced @wrapt.decorator and @wrapt.function_wrapper so they can be applied to instance methods and class methods with the self/cls argument being supplied correctly. This allows instance and class methods to be used as decorators, with the instance or class type being able to be used to hold any state required for the decorator.

Bugs Fixed

- Fixed process crash in extension when the wrapped object passed as first argument to FunctionWrapper did not have a tp_descr_get callback for the type at C code level. Now raised an AttributeError exception in line with what Python implementation does.

2.7.27 Version 1.3.1

Bugs Fixed

- The discover_post_import_hooks() function had not been added to the top level wrapt module.

2.7.28 Version 1.3.0

New Features

- Added a @transient_function_wrapper decorator for applying a wrapper function around a target function only for the life of a single function call. The decorator is useful for performing mocking or pass through data validation/modification when doing unit testing of packages.

2.7.29 Version 1.2.1

Bugs Fixed

- In C implementation, not dealing with unbound method type creation properly which would cause later problems when calling instance method via the class type in certain circumstances. Introduced problem in 1.2.0.

- Eliminated compiler warnings due to missing casts in C implementation.

2.7.30 Version 1.2.0

New Features

- Added an 'enabled' option to @decorator and FunctionWrapper which can be provided a boolean, or a function returning a boolean to allow the work of the decorator to be disabled dynamically. When a boolean, is used for @decorator, the wrapper will not even be applied if 'enabled' is False. If a function, then will be called prior to wrapper being called and if returns False, then original wrapped function called directly rather than the wrapper being called.
- Added in an implementation of a post import hook mechanism in line with that described in PEP 369.
- Added in helper functions specifically designed to assist in performing monkey patching of existing code.

Features Changed

- Collapsed functionality of _BoundMethodWrapper into _BoundFunctionWrapper and renamed the latter to BoundFunctionWrapper. If deriving from the FunctionWrapper class and needing to override the type of the bound wrapper, the class attribute `__bound_function_wrapper__` should be set in the derived FunctionWrapper class to the replacement type.

Bugs Fixed

- When creating a custom proxy by deriving from ObjectProxy and the custom proxy needed to override __getattr__(), it was not possible to called the base class ObjectProxy.__getattr__() when the C implementation of ObjectProxy was being used. The derived class __getattr__() could also get ignored.
- Using inspect.getargspec() now works correctly on bound methods when an adapter function can be provided to @decorator.

2.7.31 Version 1.1.3

New Features

- Added a _self_parent attribute to FunctionWrapper and bound variants. For the FunctionWrapper the value will always be None. In the case of the bound variants of the function wrapper, the attribute will refer back to the unbound FunctionWrapper instance. This can be used to get a back reference to the parent to access or cache data against the persistent function wrapper, the bound wrappers often being transient and only existing for the single call.

Improvements

- Use interned strings to optimise name comparisons in the setattro() method of the C implementation of the object proxy.

Bugs Fixed

- The pypy interpreter is missing operator.__index__() so proxying of that method in the object proxy would fail. This is a bug in pypy which is being addressed. Use operator.index() instead which pypy does provide and which also exists for CPython.
- The pure Python implementation allowed the __wrapped__ attribute to be deleted which could cause problems. Now raise a TypeError exception.
- The C implementation of the object proxy would crash if an attempt was made to delete the __wrapped__ attribute from the object proxy. Now raise a TypeError exception.

2.7.32 Version 1.1.2

Improvements

- Reduced performance overhead from previous versions. Most notable in the C implementation. Benchmark figures have been updated in documentation.

2.7.33 Version 1.1.1

Bugs Fixed

- Python object memory leak was occurring due to incorrect increment of object reference count in C implementation of object proxy when an instance method was called via the class and the instance passed in explicitly.
- In place operators in pure Python object proxy for __idiv__ and __itruediv__ were not replacing the wrapped object with the result of the operation on the wrapped object.
- In place operators in C implementation of Python object proxy were not replacing the wrapped object with the result of the operation on the wrapped object.

2.7.34 Version 1.1.0

New Features

- Added a synchronized decorator for performing thread mutex locking on functions, object instances or classes. This is the same decorator as covered as an example in the wrapt documentation.
- Added a WeakFunctionProxy class which can wrap references to instance methods as well as normal functions.
- Exposed from the C extension the classes _FunctionWrapperBase, _BoundFunctionWrapper and _BoundMethodWrapper so that it is possible to create new variants of FunctionWrapper in pure Python code.

Bugs Fixed

- When deriving from ObjectProxy, and the C extension variant was being used, if a derived class overrode __new__() and tried to access attributes of the ObjectProxy created using the base class __new__() before __init__() was called, then an exception would be raised indicating that the 'wrapper has not been initialised'.
- When deriving from ObjectProxy, and the C extension variant was being used, if a derived class __init__() attempted to update attributes, even the special '_self_' attributed before calling the base class __init__() method, then an exception would be raised indicating that the 'wrapper has not been initialised'.

2.7.35 Version 1.0.0

Initial release.

2.8 Known Issues

The following known issues exist.

2.8.1 @classmethod.__get__()

The Python `@classmethod` decorator assumes in the implementation of its `__get__()` method that the wrapped function is always a normal function. It doesn't entertain the idea that the wrapped function could actually be a descriptor, the result of a nested decorator. This is an issue because it means that the complete descriptor binding protocol is not performed on anything which is wrapped by the `@classmethod` decorator.

The consequence of this is that when `@classmethod` is used to wrap a decorator implemented using `@wrapt.decorator`, that `__get__()` isn't called on the latter. The result is that it is not possible in the latter to properly identify the decorator as being bound to a class method and it will instead be identified as being associated with a normal function, with the class type being passed as the first argument.

The behaviour of the Python `@classmethod` is arguably wrong and a fix to Python for this issue is being pursued (http://bugs.python.org/issue19072). The only solution is the recommendation that decorators implemented using `@wrapt.decorator` always be placed outside of `@classmethod` and never inside.

2.8.2 Using decorated class with super()

In the implementation of a decorated class, if needing to use a reference to the class type with super, it is necessary to access the original wrapped class and use it instead of the decorated class.

```
@mydecorator
class Derived(Base):

    def __init__(self):
        super(Derived.__wrapped__, self).__init__()
```

If using Python 3, one can simply use `super()` with no arguments and everything will work fine.

```
@mydecorator
class Derived(Base):

    def __init__(self):
        super().__init__()
```

2.8.3 Deriving from decorated class

If deriving from a decorated class, it is necessary to access the original wrapped class and use it as the base class.

```
@mydecorator
class Base(object):
    pass

class Derived(Base.__wrapped__):
    pass
```

In doing this, the functionality of any decorator on the base class is not inherited. If creation of a derived class needs to also be mediated via the decorator, the decorator would need to be applied to the derived class also.

In this case of trying to decorate a base class in a class hierarchy, it may turn out to be more appropriate to use a meta class instead of trying to decorate the base class.

Note that as of Python 3.7 and wrapt 1.12.0, accessing the true type of the base class using `__wrapped__` is not required. Such code though will not work for versions of Python older than Python 3.7.

CHAPTER 3

Presentations

Conference presentations related to the **wrapt** module:

- http://lanyrd.com/2013/kiwipycon/scpkbk

CHAPTER 4

Blog Posts

Blog posts related to the **wrapt** module:

- https://github.com/GrahamDumpleton/wrapt/tree/master/blog

CHAPTER 5

Installation

The **wrapt** module is available from PyPi at:

- https://pypi.python.org/pypi/wrapt

and can be installed using `pip`.

```
pip install wrapt
```

CHAPTER 6

Source Code

Full source code for the **wrapt** module, including documentation files and unit tests, can be obtained from github.

- https://github.com/GrahamDumpleton/wrapt