Decorators

February 19, 2026

1 Closures

https://en.wikipedia.org/wiki/Closure_(computer_programming)

A **closure**, also **lexical closure** or **function closure**, is a technique for implementing *lexically scoped name binding* in a language with *first-class functions*. Operationally, a closure is a record **storing a function together with an environment**. The environment is a mapping associating each *free variable* of the function (*variables that are used locally, but defined in an enclosing scope*) with the value or reference to which the name was bound when the closure was created. Unlike a plain function, a closure allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

[1]:
```
def f(x):
    def g(y):
        return x + y
    return g  # Return a closure. x is a free variable, bound by the closure.

def h(x):
    return lambda y: x + y  # Return a closure.

my_closure = f(2)
print(my_closure(3))
```

```
5
```

2 Decorators

A decorator is an object that wraps a function, method or class. It can perform operations before and/or after making a call to the wrapped object.

2.0.1 Why we use decorators

In computer science there is a concept called **Aspect Oriented Programming**.

The basis of this is the Single Responsibility Principle (SRP), which states that each module, function or class should deal with **one** aspect of the problem we are solving. However, we often need to deal with things like logging, cacheing, authentication, and other matters that are not the purpose of an object - they pertain to other aspects of the solution and should not be part of the core function.

We can then use decorators to separate the functionality into two or more parts and make the decorating function replace all calls to the decorated function with calls to itself. The decorator handles the actual calls to the function it decorates.

Benefits:

- Cleaner, more readable code
- Multiple decorators to further divide the problem
- Separation of concerns
- Division of tasks (domain experts solve domain problems, computer scientists solve other)
- Standard decorators available in the standard library
- Reuse of code

2.0.2 A function based decorator

[2]:
```
import time
from functools import wraps

def timeit(target_function):

    @wraps(target_function)
    def wrapper(*args, **kwargs):
        before = time.time()
        result = target_function(*args, **kwargs)
        after = time.time()
        print(after-before)
        return result
    return wrapper

@timeit
def myfun():
    time.sleep(3)

#myfun = timeit(myfun)

myfun()
myfun
```

```
3.000415325164795
```

[2]:
```
<function __main__.timeit.<locals>.wrapper(*args, **kwargs)>
```

2.0.3 A class based decorator

This decorator does the same thing as the function based decorator above.

[4]:
```
import time
class timeit2:
    def __init__(self, function):
        self._function = function
```

```
    def __call__(self, *args, **kwargs):
        before = time.time()
        result = self._function(*args, **kwargs)
        after = time.time()
        print(after-before)
        return result

@timeit2
def myfun():
    time.sleep(3)

# myfun = timeit2(myfun)

myfun()
myfun
```

```
3.000054359436035
```

[4]:
```
<__main__.timeit2 at 0x7fb0fc161390>
```

2.0.4 Decorators with parameters

If you want to pass parameters to a decorator, there are a couple of different ways to do it. In this example, we just need one level of nesting, because we don't need to call the decorated function inside the closure, we can just return the reference to the function object of the decorated function.

We will have a real world example later, where we have double layers of nesting.

[3]:
```
# Decorators with parameters

def do_before(*args, **kwargs):
    print("Inside decorator")
    print(args, kwargs)
    def inner(func):
        print("Inside inner function")
        print("I like", kwargs['like'])
        if kwargs['like'] == 'Python':
            return lambda : None
        else:
            return func
    return inner

@do_before(like='Perl')
def my_func():
    print('Inside actual function')

my_func()
print(my_func)
```

```
Inside decorator
() {'like': 'Python'}
Inside inner function
I like Python
<function do_before.<locals>.inner.<locals>.<lambda> at 0x7fb6750c20c0>
```

2.0.5 A real world example

This decorator has been used for many years in a booking system to authenticate users in the realms (categories) of *user* and *admin*. It was originally written for Python 2.6 using *Twisted Web* and *Nevow*.

[5]:
```python
def authenticate(realm):
    """A method decorator that requires Basic http authentication before the
    render method of the resource can be accessed.

    It is only (known to be) usable for decorating render methods of
    Page objects.
    """
    def decorator(func):
        def wrapper(self, ctx, data):
            req = inevow.IRequest(ctx)
            auth = req.getHeader('authorization')
            authenticated = False
            if auth and auth.startswith('Basic '):
                name_pw = binascii.a2b_base64(auth[6:])
                with open('authentication.{}'.format(realm), 'r') as f:
                    for line in f:
                        if line.rstrip() == name_pw:
                            authenticated = True
                            break

                if authenticated:
                    # Let the resource know who was authenticated
                    self.username = name_pw.split(':')[0]
                    return func(self, ctx, data)

            req.setResponseCode(http.UNAUTHORIZED)
            req.setHeader('www-authenticate', 'Basic realm="{}"'.format(realm))
            return renderPage(T.h1['Authorization needed.'])

        wrapper.__name__ = func.__name__
        wrapper.__dict__ = func.__dict__
        wrapper.__doc__ = func.__doc__
        wrapper.__wrapped__ = func

        return wrapper
    return decorator
```

2.0.6 Decorating a class

With a class decorator, we can change the behaviour of anything in the class. Remember that

```
@time_all_methods
class Foo:
```

just means

```
Foo = time_all_methods(Foo)
```

so

```
my_obj = Foo()    ->    my_obj = time_all_methods(Foo())
```

[4]:
```
import time
def time_this(original_function):
    print('Decorating')
    def new_function(*args,**kwargs):
        print('Starting timer')
        before = time.time()
        result = original_function(*args,**kwargs)
        after = time.time()
        print('Elapsed time = {0}'.format(after-before))
        return result
    return new_function

def time_all_methods(cls):
    class NewCls:
        def __init__(self,*args,**kwargs):
            self.wrapped_object = cls(*args,**kwargs)

        def __getattribute__(self, s):
            """
            this is called whenever any attribute of a NewCls object is␣
 ↪accessed. This function first tries to
            get the attribute off NewCls. If it fails then it tries to fetch␣
 ↪the attribute from self.oInstance (an
            instance of the decorated class). If it manages to fetch the␣
 ↪attribute from self.oInstance, and
            the attribute is an instance method then `time_this` is applied.
            """
            try:
                attr = super().__getattribute__(s)
            except AttributeError:
                pass
            else:
                return attr
            attr = self.wrapped_object.__getattribute__(s)
            if type(attr) == type(self.__init__): # it is an instance method
```

```
                return time_this(attr)                  # this is equivalent of␣
 ↪just decorating the method with time_this
            else:
                return attr
    return NewCls

@time_all_methods
class Foo:
    def a(self):
        print('Entering a')
        time.sleep(3)
        print('Exiting a')

my_instance = Foo()
my_instance.a()
```

```
Decorating
Starting timer
Entering a
Exiting a
Elapsed time = 3.000577211380005
```

2.0.7 Testing a decorator

1. Make a mock function with the right signature
2. Manually wrap it with your decorator, using monkey patching
3. Prepare any arguments that may be needed
4. Call the decorator
5. Make whatever assertions that are needed
    - Assertions on the result
    - Assertions on exceptions
    - Assertions on contents of the mock object

[7]:
```
import ipytest
ipytest.autoconfig()

__file__ = "Decorators.ipynb"
```

[8]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest
from unittest.mock import Mock

class TestLoginRequired():
    def test_no_user(self):
        func = Mock()
        decorated_func = login_required(func)
        request = prepare_request_without_user()
```

```
        response = decorated_func(request)

        # assert response is redirect
        assert not func.called

    def test_bad_user(self):
        func = Mock()
        decorated_func = login_required(func)
        request = prepare_request_with_non_authenticated_user()

        response = decorated_func(request)

        # assert response is redirect
        assert not func.called

    def test_ok(self):
        func = Mock(return_value='my response')
        decorated_func = login_required(func)
        request = prepare_request_with_ok_user()

        response = decorated_func(request)

        func.assert_called_with(request)
        assert response == 'my response'
```

```
FFF                                                                      [100%]
========================================== FAILURES ==========================================
_____________________________________ TestLoginRequired.test_no_user _____________________________________

self = <__main__.TestLoginRequired object at 0x7f64d99a0700>

    def test_no_user(self):
        func = Mock()
>       decorated_func = login_required(func)
E       NameError: name 'login_required' is not defined

/tmp/ipykernel_836662/726263122.py:7: NameError
_____________________________________ TestLoginRequired.test_bad_user _____________________________________

self = <__main__.TestLoginRequired object at 0x7f64d9c19fa0>

    def test_bad_user(self):
```

```
        func = Mock()
>       decorated_func = login_required(func)
E       NameError: name 'login_required' is not defined

/tmp/ipykernel_836662/726263122.py:17: NameError
_____________________________________ TestLoginRequired.test_ok

-------------------------------------

self = <__main__.TestLoginRequired object at 0x7f64d99a0250>

    def test_ok(self):
        func = Mock(return_value='my

response')
>       decorated_func = login_required(func)
E       NameError: name 'login_required' is not defined

/tmp/ipykernel_836662/726263122.py:27: NameError
==================================== short test summary info
=====================================
FAILED tmpyhcwff4w.py::TestLoginRequired::test_no_user - NameError: name
'login_required' is not …
FAILED tmpyhcwff4w.py::TestLoginRequired::test_bad_user - NameError: name
'login_required' is not…
FAILED tmpyhcwff4w.py::TestLoginRequired::test_ok - NameError: name
'login_required' is not defined
```

2.0.8 Testing functions with decorators

Sometimes you can test the function with the decorator in place, because the decorator is unobtrusive. This is not really orthodox unit testing, but it is simple to apply.

At other times, you really want to test the function without the decorator getting applied. In modern Python versions the function with the decorator applied, has an attribute **__wrapped__**, that contains a reference to the original function, if the wrapper function has been decorated with the @wraps decorator.

[9]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest
import time
from functools import wraps
def timeit(target_function):
    @wraps(target_function)
    def wrapper(*args, **kwargs):
        #args[0] += 1
        before = time.time()
        result = target_function(*args, **kwargs)
        after = time.time()
```

```
        print(after-before)
        return result
    return wrapper

@timeit
def myfun(x):
    return x + 1
#-------------------------------------------------
def test_myfun():
    wrapped_fun = myfun.__wrapped__
    n = 1
    result = wrapped_fun(n)

    assert result == n + 1
```

.

[100%]

2.0.9 Testing inner functions

Accessing inner functions for testing is hard, and most of the time you make do with writing tests for the top level function/method. However, it is possible, using some advanced introspection, fishing out the code object of the inner function from the outer function, creating a new function object and binding the necessary free variables, before calling the object.

The magic is in the *nested* module. I looked for the solution to this problem for years, before finding it in a low ranked response on Stack Overflow.

[10]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest
from nested import nested
from tested_code import f, C, m

def test_all():
    nested_g = nested(f, 'g', v1=8, v2=1)
    assert nested_g(2) == 15

    nested_h = nested(f, 'h')
    assert nested_h() == 16

    nested_k = nested(C.foo, 'k', self='mock')
    assert nested_k(5) == ['mock', 5]

    nested_n = nested(m, 'n', vm=1)
    nested_o = nested(nested_n, 'o', vm=1, an=2, vn=4)
    assert nested_o(8) == 31
```

```
.
[100%]
```