

Decorators

What is a decorator?

A decorator is a way to **add behavior** around a function or method.

```
@somedecorator  
def some_function(x, y, z):  
    # ...
```

Once it is written, using a decorator is trivially easy.

Many successful libraries use decorators extensively: Flask, Django, Twisted, pytest, nose, SQLAlchemy, and more.

Writing decorators

Writing decorators is very challenging. But today, you'll learn how to do it!

What it lets you do:

- Add rich features to groups of functions and classes
- Untangle distinct, frustratingly intertwined concerns in your code
- Encapsulate code reuse patterns not otherwise possible
- Effectively extend Python syntax in certain limited but powerful ways
- Build easily reusable frameworks

Example: property

```
>>> class Person:
...     def __init__(self, first_name, last_name):
...         self.first_name = first_name
...         self.last_name = last_name
...
...     @property
...     def full_name(self):
...         return self.first_name + " " + self.last_name
...
>>> person = Person("John", "Smith")
>>> print(person.full_name)
John Smith
```

Example: Flask

```
@app.route("/")  
def hello():  
    return "<html><body>Hello World!</body></html>"
```

Example: thread locking

```
@withlock
def first_method_in_group(self, arg):
    ...
@withlock
def another_method_in_group(self, arg):
    ...
```

@ is a Shorthand

This:

```
@some_decorator  
def some_function(arg):  
    # blah blah
```

is equivalent to this:

```
def some_function(arg):  
    # blah blah  
some_function = some_decorator(some_function)
```

It's just a function

A decorator is **just a function**. That's all.

It is a function that takes exactly one argument, which is a function object.

And it returns a *different* function.

```
def some_function(arg):  
    # blah blah  
some_function = some_decorator(some_function)
```


Terminology

```
@some_decorator  
def some_function(arg):  
    # blah blah
```

- **decorator** - What comes after the @. It's a function.
- **bare function** - the one `def`'ed on the next line. The function being decorated.
- The result of decorating a function is the **decorated function**. It's what you actually call in your code.

Remember one thing

A decorator is just a normal, boring function.

It happens to be a function that takes exactly one argument, which is itself a function.

And when called, the decorator returns a *different* function.

Logging decorator

```
def printlog(func):  
    def wrapper(arg):  
        print("CALLING: " + func.__name__)  
        return func(arg)  
    return wrapper  
  
@printlog  
def f(n):  
    return n+2  
  
# Same as:  
def f(n):  
    return n+2  
f = printlog(f)
```

```
>>> print(f(3))  
CALLING: f  
5
```

Structure

```
def printlog(func):  
    def wrapper(arg):  
        print("CALLING: " + func.__name__)  
        return func(arg)  
    return wrapper
```

Body of printlog does just two things:

- Define a function called wrapper, and
- Return it.

That's all. Most decorators you create will follow this pattern.

Multiple Targets

Decorators are normally applied to many functions or methods.

```
@printlog
def f(n):
    return n+2

@printlog
def g(x):
    return 5 * x

@printlog
def h(arg):
    return 10 + arg
```

```
>>> print(f(3))
CALLING: f
5
>>> print(g(4))
CALLING: g
20
>>> print(h(5))
CALLING: h
15
```

Masking

```
def check_id(func):  
    def wrapper(arg):  
        print("ID of func: {}".format(id(func)))  
        return func(arg)  
    print("ID of wrapper: {}".format(id(wrapper)))  
    return wrapper
```

```
>>> @check_id  
... def f(x): return x * 3  
ID of wrapper: 4329698984  
>>>  
>>> f(2)  
ID of func: 4329698576  
6  
>>> id(f)  
4329698984
```

Practice syntax

Open a file named `decorators1.py`, and type this in:

```
def printlog(func):  
    def wrapper(arg):  
        print("CALLING: " + func.__name__)  
        return func(arg)  
    return wrapper  
@printlog  
def f(n):  
    return n+2  
  
print(f(3))
```

Run the script. Output should be:

```
CALLING: f  
5
```

Extra credit: Define & decorate new functions. Can you trigger interesting errors?

A shortcoming

```
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> baz(3, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wrapper() takes 1 positional argument but 2 were given
```

What went wrong?

Generalizing

```
# A MUCH BETTER printlog.  
def printlog(func):  
    def wrapper(*args, **kwargs):  
        print("CALLING: " + func.__name__)  
        return func(*args, **kwargs)  
    return wrapper
```

Rule of thumb: always define your `wrapper` function to accept `*args` and `**kwargs`, unless you have a specific reason not to.

Generalized

This decorator is compatible with *any* Python function:

```
>>> @printlog
... def foo(x):
...     print(x + 2)
...
>>> @printlog
... def baz(x, y):
...     return x ** y
...
>>> foo(7)
CALLING: foo
9
>>> baz(3, 2)
CALLING: baz
9
```

Practice syntax

Open a file named `decorators2.py`, and type this in:

```
def printlog(func):  
    def wrapper(*args, **kwargs):  
        print("CALLING: " + func.__name__)  
        return func(*args, **kwargs)  
    return wrapper  
@printlog  
def g(a, b, c):  
    return a + b + c  
print(g(1, 2, 3))
```

Run the script. Output should be:

```
CALLING: g  
6
```

Why `*args` and `**kwargs`?

Two words: flexibility and power.

A decorator written to take arbitrary arguments can work with functions and methods written *years* later - code the original developer never could have anticipated.

This structure has proven very powerful and versatile.

```
# The prototypical form of Python decorators.  
def prototype_decorator(func):  
    def wrapper(*args, **kwargs):  
        return func(*args, **kwargs)  
    return wrapper
```

State in decorators

```
def history(func):  
    return_vals = set()  
    def wrapper(*args, **kwargs):  
        return_val = func(*args, **kwargs)  
        return_vals.add(return_val)  
        print('Return values: ' + str(sorted(return_vals)))  
        return return_val  
    return wrapper  
  
@history  
def foo(x):  
    return x + 2  
  
# Remember, same as:  
def foo(x):  
    return x + 2  
foo = history(foo)
```

History

```
>>> print(foo(3))
Return values: [5]
5
>>> print(foo(2))
Return values: [4, 5]
4
>>> print(foo(3))
Return values: [4, 5]
5
>>> print(foo(7))
Return values: [4, 5, 9]
9
```


Memoization

A function design pattern.

Given an expensive function f , you can cache its value.

```
def f(x, y, z):  
    # do something expensive  
  
cache = {}  
def cached_f(x, y, z):  
    # tuples can be dictionary keys.  
    key = (x, y, z)  
    if key not in cache:  
        cache[key] = f(x, y, z)  
    return cache[key]
```

This has been around for decades. It's still useful.

Lab: memoize

```
# Turn this:
cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]
# ... into this:
@memoize
def f(x, y, z):
    # ...
```

Lab file: decorators/memoize.py

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- and then do decorators/memoize_extra.py

HINT: In `memoize.py`, `wrapper` takes just `*args`, not `**kwargs`.

Stacking Decorators

You can stack decorators. Simply write them on separate lines.

```
@add2
@mult3
def foo(n):
    return n + 1
# That's shorthand for this:
foo = add2(mult3(foo))
```

add2 adds two, and mult3 multiplies by three:

```
def add2(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) + 2
    return wrapper
```

```
def mult3(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) * 3
    return wrapper
```

What will `foo(3)` return?

Stacking Order

The order of stacking matters.

```
>>> # shorthand for "foo = add2(mult3(foo))"
... @add2
... @mult3
... def foo(n):
...     return n + 1
>>> foo(3)
14
```

```
>>> # shorthand for "foo = mult3(add2(foo))"
... @mult3
... @add2
... def foo(n):
...     return n + 1
>>> foo(3)
18
```

Decorators That Take Arguments

Remember this:

```
@app.route("/")  
def hello():  
    return "<html><body>Hello World!</body></html>"
```

This is different from the decorators we've written so far, because it takes an argument. How do we do that?

Simpler example

Imagine a family of "adding" decorators.

```
def add2(func):  
    def wrapper(*args, **kwargs):  
        return func(*args, **kwargs) + 2  
    return wrapper  
  
def add4(func):  
    def wrapper(*args, **kwargs):  
        return func(*args, **kwargs) + 4  
    return wrapper  
  
@add2  
def foo(x):  
    return x ** 2  
  
@add4  
def bar(n):  
    return n * 2
```

DRY

There is literally only one character difference between `add2` and `add4`; it's very repetitive, and poorly maintainable.

Better:

```
@add(2)
def foo(x):
    return x ** 2

@add(4)
def bar(n):
    return n * 2
```

How do we do that?

Generating decorators

```
@add(2)
def foo(x):
    return x ** 2
```

`add` is actually *not* a decorator; it is a function that *returns* a decorator.

In other words, `add` is a function that returns another function. (Since the returned decorator is, itself, a function).

Nesting functions

Write a function called `add`, which creates and returns the decorator.

```
def add(increment):  
    def decorator(func):  
        def wrapper(*args, **kwargs):  
            return increment + func(*args, **kwargs)  
        return wrapper  
    return decorator
```


Using add()

These all mean the exact same thing:

```
# This...
@add(2)
def f(n):
    # ....

# ... is the same as this...
add2 = add(2)
@add2
def f(n):
    # ....

# ... and the same as this.
def f(n):
    # ....
f = add(2)(f)
```


Break it down...

```
def add(increment):  
    def decorator(func):  
        def wrapper(*args, **kwargs):  
            return increment + func(*args, **kwargs)  
        return wrapper  
    return decorator
```

- wrapper: just like in the other decorators
- decorator: What's applied to the bare function
- (Hint: we could say `add2 = add(2)`, then apply `add2` as a decorator)
- `add`: This is not a decorator. It's a function that returns a decorator.

Closure

```
def add(increment):  
    def decorator(func):  
        def wrapper(*args, **kwargs):  
            return increment + func(*args, **kwargs)  
        return wrapper  
    return decorator
```

`increment` variable is encapsulated in the scope of the `add` function.

We can't access its value outside the decorator, in the calling context. But we don't need to.

Practice syntax

Create a file `decoratoradd.py`, and write in the following:

```
def add(increment):  
    def decorator(func):  
        def wrapper(*args, **kwargs):  
            return increment + func(*args, **kwargs)  
        return wrapper  
    return decorator  
  
@add(3)  
def f(n):  
    return n + 2  
  
print(f(4))
```

Output should be "9".

Extra credit: Create and use a `multiply` decorator.

Lab: The returns decorator

Runtime type checking:

```
# Raises TypeError if return value is not an int
@returns(int)
def f(x, y):
    if x > 3:
        return -1.5
    return x + y
```

(Hint: use `isinstance()`)

Lab file: `decorators/returns.py`

- In `labs/py3` for 3.x; `labs/py2` for 2.7
- When you are done, give a thumbs up...
- ... and then do `decorators/webframework.py`

Class-Based Decorators

So far, we've made each decorator by defining a function.

It turns out, you can also create one using a class.

Advantages:

- Can leverage inheritance, encapsulation, etc.
- Can sometimes be more readable for complex decorators

The call hook

Any object with a `__call__` method can be treated like a function.

```
class Prefixer:
    def __init__(self, prefix):
        self.prefix = prefix
    def __call__(self, message):
        return self.prefix + message
```

It's called a **callable**, meaning you can call it like a function:

```
>>> simonsays = Prefixer("Simon says: ")
>>> simonsays("Get up and dance!")
'Simon says: Get up and dance!'
```


The call hook

It's not a function! It's just callable like one.

```
>>> type(simonsays)
<class '__main__.Prefixer'>
```

When you call it like a function, this dispatches to the `__call__` method.

```
>>> simonsays("High five!")
'Simon says: High five!'
>>> simonsays.__call__("High five!")
'Simon says: High five!'
```

Important Note

It's possible to apply decorators to classes, just like you've applied them to functions.

This is a COMPLETELY DIFFERENT THING than class-based decorators.

@printlog as a function

As a reminder (the same code as before):

```
def printlog(func):  
    def wrapper(*args, **kwargs):  
        print("CALLING: " + func.__name__)  
        return func(*args, **kwargs)  
    return wrapper
```

```
>>> @printlog  
... def foo(x):  
...     print(x + 2)  
...  
>>> foo(7)  
CALLING: foo  
9
```

@PrintLog as a class

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print('CALLING: {}'.format(self.func.__name__))
        return self.func(*args, **kwargs)

# Compare to the function version (from last slide):
def printlog(func):
    def wrapper(*args, **kwargs):
        print("CALLING: " + func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

Works the same!

To use this:

```
>>> @printlog
... def foo_func(x):
...     print(x + 2)
...
>>> @PrintLog
... def foo_class(x):
...     print(x + 2)
...
>>> foo_func(7)
CALLING: foo_func
9
>>> foo_class(7)
CALLING: foo_class
9
```

Another look

```
class PrintLog:  
    def __init__(self, func):  
        self.func = func  
    # ...
```

Constructor takes one arg: the function being decorated. Remember, this:

```
@PrintLog  
def foo_class(x):  
    print(x+2)
```

is shorthand for this:

```
def foo_class(x):  
    print(x+2)  
foo_class = PrintLog(foo_class)
```

The wrapped "function" is actually a `PrintLog` object.

Another look

```
class PrintLog:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args, **kwargs):
        print('CALLING: {}'.format(self.func.__name__))
        return self.func(*args, **kwargs)
```

The function being decorated is stored as `self.func`.

`__call__` is, in essence, the wrapper function.

Uses

Some reasons to use class-based decorators instead of functions:

- 1) To leverage inheritance, or other OO features
- 2) To store state in the decorator (as object attributes)
- 3) You feel it's more readable. (Some people like one form better than the other.)

Lab: Classy Memoizing

```
# Turn this:
cache = {}
def cached_f(x, y, z):
    # tuples can be dictionary keys.
    key = (x, y, z)
    if key not in cache:
        cache[key] = f(x, y, z)
    return cache[key]
# ... into this (a Memoize class, instead of a memoize function):
@Memoize
def f(x, y, z):
    # ...
```

Lab file: `decorators/memoize_class.py`

- In `labs/py3` for 3.x; `labs/py2` for 2.7
- HINT: In `memoize_class.py`, wrapper takes just `*args`, not `**kwargs`.
- When you are done, give a thumbs up...
- ... then do `decorators/memoize_class_extra.py` (after `memoize_extra.py`)