

Decorators, unwrapped



Katie Silverio
@astrosilverio

I use decorators
but they are 🌟



I can show you
how they ⚙️!



Decorators, unwrapped

- * Formal definition of a decorator
- * Anatomy of a decorator
- * Why it works
- * Common tricks and gotchas

What *is* a decorator?

✨*examples*✨

What *is* a decorator?

```
@timer  
def my_function():  
    # code goes here
```

```
> my_function()  
my_function ran in 0.005s
```

```
> my_function()  
my_function ran in 0.008s
```

What *is* a decorator?

```
class MyClass(object):  
  
    @classmethod  
    def no_need_for_an_instance(cls):  
        # some code goes here
```

```
> MyClass.no_need_for_an_instance()  
some output
```

A decorator changes the behavior of a function without the user having to change the function code itself



Also a decorator can be used to decorate many functions!

Time this!

```
def my_function(arg):  
    # some code goes here  
    return my_return_value
```

```
> result = my_function(some_value)  
0.11s  
> result = my_function(some_other_value)  
1.01s
```


Timing, constructed

```
import time

def my_function(arg):
    start_time = time.time()
    # original body of original function
    end_time = time.time()
    print end_time - start_time
    return original_return_value
```

Timing, decorated

```
import my_timer
# how does it work? a mystery

@my_timer
def my_function(arg):
    # original body of function
    return original_return_value

@my_timer
def foo(bar):
    # foo the bar
    return baz
```

What *is* a function?

✨ *a first-class object* ✨

What *is* a first-class object?

- * FCOs can be assigned to variables
- * FCOs can be passed as arguments to other functions
- * FCOs can be returned from other functions
- * FCOs can be created within functions

Refactoring goals

- * Decouple timing code from function code
- * Make timing code reusable

Back to timing

```
import time
```

```
def timing_wrapper(func, func_arg):  
    start_time = time.time()  
    value_to_return = func(func_arg)  
    end_time = time.time()  
    print end_time - start_time  
    return value_to_return
```

```
> timing_wrapper(my_function, some_arg)
```

Refactoring goals

- * Decouple timing code from function code
- * Make timing code reusable
- * Wrap functions with timing code at definition time

Back to timing

```
import time
```

```
def my_timer(func):  
    # creates a new function  
    # that executes `func` AND  
    # performs timing code  
    return new_function
```

```
timed_function = my_timer(my_function)
```


Final refactor!

```
import time
```

```
def my_timer(func):
```

```
    def new_wrapped_function(func_arg):  
        start_time = time.time()  
        value_to_return = func(func_arg)  
        end_time = time.time()  
        print end_time - start_time  
        return value_to_return
```

```
    return new_wrapped_function
```

Final refactor!

```
import my_timer

def foo(bar):
    # foo the bar
    return baz

foo = my_timer(foo)
```

Back to decorators

From PEP 318:

```
def foo(self):  
    perform method operation  
foo = classmethod(foo)
```

Back to decorators

```
import my_timer

def foo(bar):
    # foo the bar

foo = my_timer(foo)
```

is equivalent to

```
import my_timer

@my_timer
def foo(bar):
    # foo the bar
```

A decorator replaces a function with the decorator's own return value, when called with the function it decorates



General decorator form

```
def my_decorator(func):  
  
    def new_function(*args, **kwargs):  
        # body probably contains some  
        # new code and probably a call  
        # to func and probably returns  
        # the return value of func  
  
    # probably  
    return new_function
```

"Probably"

```
def no_op(func):  
    return func
```

```
def sleight_of_hand(func):  
    def answer_is_42(*args, **kwargs):  
        return 42  
  
    return answer_is_42
```

```
def black_hole(func):  
    return None
```

Open Questions

- * How does the inner function have access to the wrapped function?
- * Why do many people use `@wraps`?
- * What order are stacked decorators applied in?
- * How does the `@` syntactic sugar work?

Scope

```
def my_function(some_arg):  
    print some_arg    # works
```

Scope

```
def my_function(some_arg):  
    print some_arg    # works  
    print locals()
```

```
> my_function(1)  
1  
{ 'some_arg': 1 }
```

Scope

```
def my_function(some_arg):  
    print some_arg    # works  
  
    def inner_function():  
        print some_arg    # works  
        print locals()  
  
    return inner_function
```

```
> my_function(42) ()  
42  
42  
{ 'some_arg': 42 }
```

Open Questions

- * How does the inner function have access to the wrapped function?
- * Why do many people use `@wraps`?
- * What order are stacked decorators applied in?
- * How does the `@` syntactic sugar work?

Timing decorator

```
import time
```

```
def my_timer(func):
```

```
    def new_wrapped_function(func_arg):  
        start_time = time.time()  
        value_to_return = func(func_arg)  
        end_time = time.time()  
        print end_time - start_time  
        return value_to_return
```

```
    return new_wrapped_function
```

Misdirection

```
import my_timer
```

```
@my_timer  
def foo(bar):  
    # foo the bar!  
    return baz
```

```
> foo.__name__  
new_wrapped_function    # ???
```



Enter @wraps!

- * Replaces the wrapper's `__name__` with the name of the wrapped function
- * Replaces the wrapper's `__doc__` with the docstring of the wrapped function
- * Does this through decorating the wrapper

Fix it with decorators

```
import time
from functools import wraps

def my_timer(func):
    @wraps
    def new_wrapped_function(func_arg):
        start_time = time.time()
        value_to_return = func(func_arg)
        end_time = time.time()
        print end_time - start_time
        return value_to_return

    return new_wrapped_function
```


Open Questions

- * How does the inner function have access to the wrapped function?
- * Why do many people use `@wraps`?
- * What order are stacked decorators applied in?
- * How does the `@` syntactic sugar work?

Decorators can stack!

```
import my_timer
import another_decorator

@another_decorator
@my_timer
def foo(bar):
    # do something!
    return baz
```

Remember

```
@my_timer  
def foo(bar):  
    # do something!
```

is equivalent to

```
foo = my_timer(foo)
```

What is happening?

```
foo = another_decorator(my_timer(foo))
```

or

```
foo = my_timer(another_decorator(foo))
```

Always wrap down

```
@another_decorator  
@my_timer  
def foo(bar):  
    # do something!
```

is equivalent to

```
foo = another_decorator(my_timer(foo))
```

More Questions

- * How does the @ syntactic sugar work?
- * How do decorators that take arguments work?
- * How do class decorators work?
- * How do you write a decorator that is not a function?
- * How does @wraps work?

How do decorators work?



Decorators, in brief

- * `@decorator` is really syntactic sugar for
`object = decorator(object)`
- * no other rules!

Further Reading

- * <http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/>
- * <https://www.thecodeship.com/patterns/guide-to-python-function-decorators/>
- * <https://www.python.org/dev/peps/pep-0318/>