The not so tricky explanation of…

Decorators

Easier than you think
No real magic
Super useful - DRY

First - let's do a primer on functions

- Don't worry - we will get to decorators too!

Functions are first class citizens

What does that mean?

- Functions are objects
- A function can be stored in a variable
- A function can be used as a function argument
- A function can be returned from another function

Functions are objects

```
>>> def some_function():
...     pass
...
>>> print isinstance(some_function, object)
True
```

A function can be stored in a variable

```
>>> def some_really_long_function_name(a, b, c):
...     print a + b + c
...
>>> add_all = some_really_long_function_name
>>> add_all(1, 2, 3)
6
```

A function can be used as function argument

```
>>> def function_executor(func):
...     func()
...
>>> def hello_world():
...     print "Hello World"
...
>>> function_executor(hello_world)
Hello World
```

A function can be returned from another function

```
>>> def func_creator():
...     def created_function():
...         print "created function"
...     return created_function
...
>>> created_func = func_creator()
>>> created_func()
created function
```

A couple more things we need to know

- A nested function has access to the local variables from the wrapper function.
- If the nested function uses these, it creates a closure.

Local variables are available

```
>>> def func_creator():
...     n = 10
...     def created_function():
...         print n
...     return created_function
...
>>> created_func = func_creator()
>>> created_func()
10
```

And they create a closure

```
>>> def func_creator():
...     n = 10
...     def created_function():
...         print n
...     return created_function
...
>>> created_func = func_creator()
>>> created_func()
10
>>> def func_creator(n=10):
...     def created_function():
...         print n
...     return created_function
...
>>> created_func_20 = func_creator(20)
>>> created_func_40 = func_creator(40)
>>> created_func_20()
20
>>> created_func_40()
40
>>>
```

Let's review! Then an example

- A function can be used as a function argument
- A function can be returned from another function
- When a nested function uses local variables, it creates a closure

Wrapping a function

- times_five takes a function as an argument
- It returns a function fn_times_five
- It creates a closure around the function passed in

```
>>> def times_five(fn):
...     def fn_times_five():
...         print fn() * 5
...     return fn_times_five
...
>>> def ten():
...     return 10
...
>>> ten_times_five = times_five(ten)
>>> ten_times_five()
50
```

Wrapping a function with arguments

- times_five takes a function as an argument
- It returns a function fn_times_five
- It creates a closure and returns a function that TAKES ARGUMENTS!!!

```
>>> def times_five(fn):
...     def fn_times_five(*args, **kwargs):
...         print fn(*args, **kwargs) * 5
...     return fn_times_five
...
>>> def square(x=1):
...     return x * x
...
>>> square_times_five = times_five(square)
>>> square_times_five(4)
80
```

Is Everyone Still With Me?

- Once we understand all of the previous slides, we can now learn about what a decorator is.
- Haha - tricked you.
- We already learned what they are!

WHAT????

```
>>> def times_five(fn):
...     def fn_times_five(*args, **kwargs):
...         print fn(*args, **kwargs) * 5
...     return fn_times_five
...
>>> def square(x=1):
...     return x * x
...
>>> square_times_five = times_five(square)
>>> square_times_five(4)
80
```

- Remember this?
- This is a decorator. It's just the hard way to create a decorator.
- Let's look at the easy way instead.

How the code looks using decorator syntax

```
>>> def times_five(fn):
...     def fn_times_five(*args, **kwargs):
...         print fn(*args, **kwargs) * 5
...     return fn_times_five
...
>>> def square(x=1):
...     return x * x
...
>>> square_times_five = times_five(square)
>>> square_times_five(4)
80
```

```
>>> def times_five(fn):
...     def fn_times_five(*args, **kwargs):
...         print fn(*args, **kwargs) * 5
...     return fn_times_five
...
>>> @times_five
... def square(x=1):
...     return x * x
...
>>> square(4)
80
```

- The decorator is created using the @decorator statement
- It passes the next defined function to the decorator
- It’s sorta magical

Wait, what if my decorator needs arguments too?

- I am glad you asked.
- Instead of a singly nested function, now we will have two nested functions.

```
>>> def my_decorator(name):
...     def wrapped_outter(fn):
...         def wrapped_inner(*args, **kwargs):
...             return fn(*args, **kwargs) + name
...         return wrapped_inner
...     return wrapped_outter
...
>>> @my_decorator("Sean")
... def greeting():
...     return "Nice to meet you "
...
>>> print greeting()
Nice to meet you Sean
```

Real world examples

- Remember Flask and Bottle and the @route decorator
- @route(“/path/to/thing”, methods=[“GET”, “POST”])

```
def route(self, rule, **options):
    def decorator(f):
        endpoint = options.pop('endpoint', None)
        self.add_url_rule(rule, endpoint, f, **options)
        return f
    return decorator
```

Real world examples

- Build your own timer for any function…

```
>>> import time
>>> def timer(fn):
...     def wrapped(*args, **kwargs):
...         t_start = time.time()
...         value = fn(*args, **kwargs)
...         t_end = time.time()
...         print t_end - t_start
...         return value
...     return wrapped
...
>>> @timer
... def sleep_a_bit(x=1):
...     time.sleep(x)
...     return "Done"
...
>>> print sleep_a_bit()
1.0000269413
Done
>>> sleep_a_bit(5)
5.00082087517
Done
```

functools.wraps

- Decorated functions lose information you might care about
- __name__ and __doc__ are not preserved
- You can use @functool.wraps(fn)
- Also look at package `wrapt` on PyPI or Github

Decorator Classes

```
import time
import functools

class Timer(object):

    def __init__(self, wrapped):
        self.wrapped = wrapped
        functools.update_wrapper(self, self.wrapped)

    def __call__(self, *args, **kwargs):
        start = time.time()
        return_val = self.wrapped(*args, **kwargs)
        end = time.time()
        self.time = end - start
        return return_val

@Timer
def sleep(t=5):
    """Sleep for a bit"""
    time.sleep(t)
    print "Hello"

sleep(1)
print sleep.time
print sleep.__name__
print sleep.__doc__
```

```
Hello
1.00113415718
sleep
Sleep for a bit
```

Questions