Python decorators

Powerful and not all that magical after all

Kenneth Nielsen[1]

[1]Center for Individual Nanoparticle Functionality (CINF)
Institute of Physics
Technical University of Denmark (DTU)

Python and beers presentation,
March 20, 2015

About me

- My name is Kenneth Nielsen and I work at CINF
- Free and open source software (FOSS) enthusiast
- Long time FOSS translator (and developer)
- BIG TIME Pythonista

Slides and files for examples

```
git clone https://github.com/KennethNielsen/presentations.git
```

`https://github.com/KennethNielsen/presentations`
`http://bit.ly/1FclzDR`

Decorators

```
@my_decorator
def my_function():
    pass
```

- Wraps (or decorates) Python functions
- Can be used to add functionality to functions
- Start-end, enter-exit, startup-teardown type code (or just one or the other)
- Bah! Lets talk about sandwiches

Sandwich machine

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')

...
```

Sandwich machine

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')
```

...

Kind of annoying, right?

Sandwich machine

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')
```

...

Kind of annoying, right? Because it's dry!

D

R

Y

Don't
Repeat
Yourself

Sandwich machine

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')

...
```

Sandwich machine

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')
```

. . .

Aha! We can fix that with a sub function.

Sandwich machine with sub function

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    blt()
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    blt()
    print('Apply top half of full wheat bun')

...

def blt():
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
```

Sandwich machine with sub function

```
def white_bread_blt():
    print('Apply bottom half of white bun')
    blt()
    print('Apply top half of white bun')

def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    blt()
    print('Apply top half of full wheat bun')

...

def blt():
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
```

But what if it was the other way around?

Sandwich machine with sub function

```
def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')

def full_wheat_ham_and_cheese():
    print('Apply bottom half of full wheat bun')
    print('Apply Ham')
    print('Apply Cheese')
    print('Apply top half of full wheat bun')

...
```

Sandwich machine with sub function

```
def full_wheat_blt():
    print('Apply bottom half of full wheat bun')
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    print('Apply top half of full wheat bun')

def full_wheat_ham_and_cheese():
    print('Apply bottom half of full wheat bun')
    print('Apply Ham')
    print('Apply Cheese')
    print('Apply top half of full wheat bun')
```

...

Could try with sub functions

Sandwich machine with sub function

```
def full_wheat_blt():
    full_wheat_bottom()
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    full_wheat_top()

def full_wheat_ham_and_cheese():
    full_wheat_bottom()
    print('Apply Ham')
    print('Apply Cheese')
    full_wheat_top()

def full_wheat_bottom():
    print('Apply bottom half of full wheat bun')
def full_wheat_top():
    print('Apply top half of full wheat bun')
```

Sandwich machine with sub function

```
def full_wheat_blt():
    full_wheat_bottom()
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')
    full_wheat_top()

def full_wheat_ham_and_cheese():
    full_wheat_bottom()
    print('Apply Ham')
    print('Apply Cheese')
    full_wheat_top()

def full_wheat_bottom():
    print('Apply bottom half of full wheat bun')
def full_wheat_top():
    print('Apply top half of full wheat bun')
```

Yuck!

Yay, decorators!

- Turns out, this is one of the problems that can be solved more elegantly with decorators
- But before we dive in, lets talk a little about functions and objects

Objects can be passed through functions and assigned other names

```
def pass_through(obj):
    return obj

my_list = ['Love Python and beers']
my_int = 47
my_other_int = my_int

my_new_list = pass_through(my_list)
my_new_int = pass_through(my_int)

my_list is my_new_list
my_int is my_new_int
my_int is my_other_int
# All returns True
```

Objects has methods and properties (even ints)

```
### Method on an int
my_int.bit_length
# <built-in method bit_length of int object at 0x155..>

my_int.bit_length()
# 6

### Property on an int
my_int.real
# 47

### Method on a list
my_list.append
# <built-in method append of list object at 0x7fe..>
```

Functions are objects too, part 1

```
pass_through
# <function pass_through at 0x7f706ec4f758>

### With methods
pass_through.__format__
# <built-in method __format__ of function object at 0x7f...>
pass_through.__format__('')
# '<function pass_through at 0x7f706ec4f758>'

### And properties
pass_through.__name__
# 'pass_through'
```

Functions are objects too, part 2

```
### and they can be assigned a different name
my_completely_unrelated_function = pass_through
print(my_completely_unrelated_function(9))
# 9

### so it does the same, and has the same attributes
my_completely_unrelated_function.__name__
# 'pass_through'

### because it's the same object
```

Functions are objects too, part 3

```
### Finally, because functions are objects, just like
### with any other mutable object, we can also create
### new attributes on them
pass_through.my_new_var = 47
print(pass_through.my_new_var)
# 47
```

AND functions can be passed through functions

```
def my_new_func():
    print('Python and beers')

what_a_ride = pass_through(my_new_func)
what_a_ride.__name__
# 'my_new_func'

what_a_ride()
# Python and beers
```

And now back to decorators!

Decorators are just syntactic sugar

```
@my_decorator
def my_function():
    pass
```

Is 100% equivalent to

```
def my_function():
    pass
my_function = my_decorator(my_function)
```

The essence of a decorator

```
@my_decorator
def my_function():
    pass
```

- A decorator is the function (my_decorator), the function it is applied to (my_function), is passed through, right after it is defined

```
def my_function():
    pass
my_function = my_decorator(my_function)
```

Enough talk, lets see one already!

A do absolutely nothing decorator (ex1.py)

```
def my_decorator(function):
    return function

@my_decorator
def my_function():
    print('python and beers')

my_function()
# python and beers
```

A needy decorator (ex2.py)

```
def my_decorator(function):
    def inner_function():
        print('Look at me, look at me, I\'m a decorator')
        function()
    return inner_function

@my_decorator
def my_function():
    print('python and beers')

my_function()
# Look at me, look at me, I'm a decorator
# python and beers
```

And now, for some actually useful decorators

A timer (ex3.py)

```
import time

def time_me(function):
    def inner_function(arg0):
        t0 = time.time()
        out = function(arg0)
        print('Duration', time.time() - t0, 's')
        return out
    return inner_function

@time_me
def square(n):
    return n ** 2

print(square(10))
# Duration 1.28746032715e-05 s
# 100
```

Decorators should be flexible

- In order for decorators to be really useful, they should be written to be flexible
- Take any combination of arguments
- Leave as little footprint on the function as possible (we won't go that much into that)

A flexible timer (ex4.py)

```
import time

def time_me(function):
    def inner_function(*args, **kwargs):
        t0 = time.time()
        out = function(*args, **kwargs)
        print('Duration', str(time.time() - t0), 's')
        return out
    return inner_function

@time_me
def square(n, repeat=1):
    for _ in range(repeat):
        out = n ** 2
    return out
```

A flexible timer (ex4.py)

```
print(square(10))
print()
print(square(10, 10**6))

# Duration 1.4066696167e-05 s
# 100
#
# Duration 0.081163883209 2 s
# 100
```

A flexible timer with cumulative sum (ex5.py), using attributes on the function

```
def time_me_cumulative(function):
    def inner_function(*args, **kwargs):
        t0 = time.time()
        out = function(*args, **kwargs)
        delta = time.time() - t0
        inner_function.cumsum += delta
        print("This run:", delta, "in total:",
              inner_function.cumsum)
        return out
    inner_function.cumsum = 0
    return inner_function
```

A flexible timer with cumulative sum (ex5.py), application

```
@time_me_cumulative
def square(n, repeat=1):
    for _ in range(repeat):
        out = n ** 2
    return out

square(10, 10**6)
square(10, 10**6)
#This run: 0.080968856811 5 in total: 0.080968856811 5
#This run: 0.079630851745 6 in total: 0.160599708557
```

Don't re-implement common tools

- At this point we are well on out way to implementing a full profiler
- If at any point you are thinking **"surely a lot of other people will have needed this before me, I wonder if there isn't a tool for it?"** you are most likely right!
- For advanced functionality use existing and commonly used implementations
- You are very likely to miss a few corner cases, that will seriously hurt you later

Have a look at `cProfile` in the standard library or `line_profiler` in pip

Debugging by print statements

- Often debugging is done by adding print statements at appropriate points
- Often on entry and exit from a function
- It can be cumbersome to add and remove all of these
- Sounds like a job for decorator

A call specification decorator (ex6.py), the decorator

```
def get_argstring(*args, **kwargs):
    """Returns str: arg0, arg1, kw0=kwarg0, kw1=kwarg1"""
    arglist = [str(arg) for arg in args]
    arglist += ['{}={}'.format(key, kwarg)
                for key, kwarg in kwargs.items()]
    return ', '.join(arglist)

def spec_me(function):
    def inner_function(*args, **kwargs):
        argstring = get_argstring(*args, **kwargs)
        print('+{}({})'.format(function.__name__,
                               argstring))
        out = function(*args, **kwargs)
        print('>{}'.format(out))
        return out
    return inner_function
```

A call specification decorator (ex6.py), application

```
@spec_me
def square(n, repeat=1):
    for _ in range(repeat):
        out = n ** 2
    return out

square(10)
#+square(10)
#>100

square(10, repeat=2)
#+square(10, repeat=2)
#>100
```

A call spec decorator with levels (ex7.py)

Trick with global variable, left as an exercise

```
@spec_me
def square(n):
    return n ** 2

@spec_me
def speak_squares(n):
    return '{} squared is {}'.format(n, square(n))

speak_squares(10)

#+speak_squares(10)
#|+square(10)
#|>100
#>10 squared is 100
```

A function with memory

- Wait a minute!
- The ability to look at input and output and to save values on the functions
- That sounds like the foundation for a function with memory

A memory decorator (ex8.py)

```
def memory(function):
    cache = {}
    def inner_function(n):
        if n not in cache:
            cache[n] = function(n)
        return cache[n]
    return inner_function

### Same result as

def memory2(function):
    def inner_function(n):
        if n not in inner_function.cache:
            inner_function.cache[n] = function(n)
        return inner_function.cache[n]
    inner_function.cache = {}
    return inner_function
```

A memory decorator (ex8.py) in action

```
from ex3 import time_me

@time_me
@memory
def expensive(n):
    for n in range(10**7):
        out = n ** 2
    return out

out1 = expensive(8)
out2 = expensive(8)
print(out1 == out2)

# Duration 0.715964078903 s
# Duration 9.53674316406e-07 s
# True
```

Neat, but its not that simple

- While this works nicely for certain kinds of functions, that are a lot of corner cases and gotchas
- What if the input was a list, you can't use a mutable object as a dictionary key in `cache`. (Make a copy as a immutable type)
- What if the input was a big numpy array, do we really want to keep a copy of that in memory? (hashing)
- What if you wanted your function to remember across script runs? How do you save input and output?

These are all good questions. But remember.

Don't re-implement common tools

- If at any point you are thinking **"surely a lot of other people will have needed this before me, I wonder if there isn't a tool for it?"** you are most likely right!
- For advanced functionality use existing and commonly used implementations
- You are very likely to miss a few corner cases, that will seriously hurt you later

Have a look at `joblib`, which is in pip.

And then the mandatory Spiderman quote

With great power comes great responsibility.

- **Don't hide large amounts of code away**, that is central to the function of your code. It will make it more difficult to read.
- **Naming is very important for decorators**
- Caching is a great way to end up with some head-scratching bugs. Make sure to understand the limitations of a tool like `joblib`.
- **But, if in doubt, just remove the decorator**. It is just 1 line that toggles the added functionality off or on. That is one of the beauties of decorators.

And then the stuff we did not talk about

A decorator usually returns another function, which means that function "metadata", like its `__name__`, its docstring and its call specification is lost! (Plus your stack-trace takes a detour)

There are (ready to use) ways around this (stdlib `wraps`, pip `wrapt`), of which at least the former is part on any good on-line tutorial.

Decorators can also take arguments!

This was a little too much to cram into one Friday afternoon presentation, but it will be part of most online tutorials.

Credits

- Colton Myers: Decorators: A Powerful Weapon in your Python Arsenal - PyCon 2014 https://www.youtube.com/watch?v=9oyr0mocZTg
- Graham Dumpleton: Advanced methods for creating decorators - PyCon 2014 https://www.youtube.com/watch?v=7jGtDGxgwEY (ALL the gory details of how to make decorators 100% transparent)
- I have shamelessly stolen the sandwich example from a Youtube video I once watched, which I have been unable to find again. Credits to you, unnamed presenter.

That's it, any questions?

```
def full_wheat(filling):
    def full_wheat_sandwich():
        print('Apply bottom half of full wheat bun')
        filling()
        print('Apply top half of full wheat bun')
    return full_wheat_sandwich

@full_wheat
def full_wheat_blt():
    print('Apply Bacon')
    print('Apply Lettuce')
    print('Apply Tomatoes')

@full_wheat
def full_wheat_ham_and_cheese():
    print('Apply Ham')
    print('Apply Cheese')
```