
# Decorators With Arguments in Python

Posted on April 28, 2015 by **admin** in **Python** | 6 Comments

## Social Media

## Recent Posts

Complete Raspberry Pi Image for Internet of Things Project
Kalman Filters, In My Own Words
Altitude Climb in a Blimp
Blimp Autopilot Flight Test in the Wind
Successful RC Blimp Autopilot Flight Test

## True Story Follows

So I'm on the grind, straight tearing up the keyboard. You'd think I was playing Street Fighter II on my laptop using Ken and fighting against Guile in front of a 2 dimensional F-16 with some random dudes fist pumping, but no, I was typing into vim.

Anyway, in my conscious effort for code to be as clean as possible and ensure that a function does exactly one thing and no more, I encountered a case where I could either break this rule or avoid redundant network calls. Specifically, I had a function that validated input (with Django forms) and a separate function that actually used the input. In this case, the input was an address, and I was making a network call via an API to turn that into lat lon coordinates.

### Recent Comments

Tim Bramlett on Learning Swift as a Python Programmer
hotigeftas on GPS and Accelerometer Sensor Fusion with a Kalman Filter, a Practical Walkthrough (Golang)
Dacian Mujdar on How to Install OpenCV on Heroku (with ffmpeg support)
Galadima Martins on Video Tutorials Outlining Inertial Measurement Unit (IMU) implementation and Sensor Fusion with GPS
JDG on Decorators With Arguments in Python

### Archives

November 2017
September 2017
April 2017
March 2017
January 2017
December 2016
October 2016
September 2016

I could certainly just bend my self-imposed rule a little bit and make the function in question mutate a value to store the latitude and longitude, but this isn't straightforward, and now the operator of the said function would need to know somehow that this mutation is occuring. Whenever I'm the operator reading code like this, the rate of "wtf's per minute" spikes briefly.

So to get to the point, I could basically keep myself and everyone else happy by writing code in the style I wanted to and maintain efficiency by avoiding the network call using memoization. The standard way to memoize functions in python is just with a simple decorator, but a super simple caching mechanism can also create memory pressure and incur a trade-off that leads to a whole new set of problems.

So to make a long story short, I wanted to make a memoization decorator that took a parameter that set a cache size. And in order to do that, I needed to do a little bit of playing around to understand how decorators in python with parameters work.

## The Basic Python Decorator

Here's the most basic syntax for a Python Decorator:

```
def pass_thru(func_to_decorate):
    def new_func(*original_args, **original_kwargs):
        print "Function has been decorated.
Congratulations."
        # Do whatever else you want here
        return func_to_decorate(*original_args,
**original_kwargs)
    return new_func
```

```
@pass_thru
def print_args(*args):
    for arg in args:
        print arg


print_args(1, 2, 3)
```

This sort of syntax is really the equivalent of:

```
def pass_thru(func_to_decorate):
    def new_func(*original_args, **original_kwargs):
        print "Function has been decorated.
Congratulations."
        # Do whatever else you want here
        return func_to_decorate(*original_args,
**original_kwargs)
    return new_func


# Notice nothing here now
def print_args(*args):
    for arg in args:
        print arg


# Notice the change here
pass_thru(print_args)(1, 2, 3)
```

## Output

```
Function has been decorated.  Congratulations.
1
2
3
```

September 2013

August 2013

# Function Decorators with Arguments

I can also do some other crazy stuff to **return a function that will take a function to decorate.**

```
def decorator(arg1, arg2):

    def real_decorator(function):

        def wrapper(*args, **kwargs):
            print "Congratulations.  You decorated
a function that does something with %s and %s" %
(arg1, arg2)
            function(*args, **kwargs)
        return wrapper

    return real_decorator


@decorator("arg1", "arg2")
def print_args(*args):
    for arg in args:
        print arg
```

And this sort of syntax is equivalent to:

```
def decorator(arg1, arg2):

    def real_decorator(function):

        def wrapper(*args, **kwargs):
            print "Congratulations.  You decorated a
function that does something with %s and %s" % (arg1,
arg2)
            function(*args, **kwargs)
        return wrapper
```

```
    return real_decorator


# No more decorator here
def print_args(*args):
    for arg in args:
        print arg


# getting crazy down here
decorator("arg1", "arg2")(print_args)(1, 2, 3)
```

## Output

```
Congratulations.  You decorated a function that does
something with arg1 and arg2
1
2
3
```

# Class Based Decorators

If you want to maintain some sort of state and/or just make your code more confusing, you can also use class based decorators. So here's an example:

```
class ClassBasedDecorator(object):

    def __init__(self, func_to_decorate):
        print "INIT ClassBasedDecorator"
        self.func_to_decorate = func_to_decorate

    def __call__(self, *args, **kwargs):
        print "CALL ClassBasedDecorator"
        return self.func_to_decorate(*args, **kwargs)


@ClassBasedDecorator
```

```
    def print_moar_args(*args):
        for arg in args:
            print arg


    print_moar_args(1, 2, 3)
    print_moar_args(1, 2, 3)
```

## Output

```
    INIT ClassBasedDecorator
    CALL ClassBasedDecorator
    1
    2
    3
    CALL ClassBasedDecorator
    1
    2
    3
```

Notice how __call__ is executed each time the function is executed.

# Class Based Decorator with Arguments

Now if we wanted to add arguments to the decorator **the structure of the class changes**, you'll note that **the function to decorate is now a parameter with the call method**.

```
    class ClassBasedDecoratorWithParams(object):

        def __init__(self, arg1, arg2):
            print "INIT ClassBasedDecoratorWithParams"
            print arg1
            print arg2
```

```
        def __call__(self, fn, *args, **kwargs):
            print "CALL ClassBasedDecoratorWithParams"

            def new_func(*args, **kwargs):
                print "Function has been decorated.
    Congratulations."
                return fn(*args, **kwargs)
            return new_func


    @ClassBasedDecoratorWithParams("arg1", "arg2")
    def print_args_again(*args):
        for arg in args:
            print arg


    print_args_again(1, 2, 3)
    print_args_again(1, 2, 3)
```

## Output

```
    INIT ClassBasedDecoratorWithParams
    arg1
    arg2
    CALL ClassBasedDecoratorWithParams
    Function has been decorated.  Congratulations.
    1
    2
    3
    Function has been decorated.  Congratulations.
    1
    2
    3
```

You can also see now that **__call__ is only called once**.

# My Memoization Decorator

And finally, we come to the reason for making an effort to understanding decorators: a simple class to memoize the results

of a function. For clarity's sake, I'm simply making it so that the results of a function with certain arguments are cached so that redundant calls don't execute potentially costly code.

```
from collections import deque


class Memoized(object):

    def __init__(self, cache_size=100):
        self.cache_size = cache_size
        self.call_args_queue = deque()
        self.call_args_to_result = {}

    def __call__(self, fn, *args, **kwargs):

        def new_func(*args, **kwargs):
            memoization_key =
self._convert_call_arguments_to_hash(args, kwargs)
            if memoization_key not in
self.call_args_to_result:
                result = fn(*args, **kwargs)

self._update_cache_key_with_value(memoization_key,
result)
                self._evict_cache_if_necessary()
            return
self.call_args_to_result[memoization_key]

        return new_func

    def _update_cache_key_with_value(self, key,
value):
        self.call_args_to_result[key] = value
        self.call_args_queue.append(key)

    def _evict_cache_if_necessary(self):
        if len(self.call_args_queue) >
self.cache_size:
            oldest_key =
self.call_args_queue.popleft()
            del self.call_args_to_result[oldest_key]

    @staticmethod
```

```
    def _convert_call_arguments_to_hash(args,
kwargs):
        return hash(str(args) + str(kwargs))


@Memoized(cache_size=5)
def get_not_so_random_number_with_max(max_value):
    import random
    return random.random() * max_value
```

## Output

Let's say I ran this:

```
print get_not_so_random_number_with_max(1)
print get_not_so_random_number_with_max(1)
print get_not_so_random_number_with_max(2)
print get_not_so_random_number_with_max(2)
print get_not_so_random_number_with_max(3)
print get_not_so_random_number_with_max(3)
print get_not_so_random_number_with_max(4)
print get_not_so_random_number_with_max(4)
print get_not_so_random_number_with_max(5)
print get_not_so_random_number_with_max(5)
print get_not_so_random_number_with_max(6)


print get_not_so_random_number_with_max(6)
print get_not_so_random_number_with_max(1)
```

If this works properly, a random number will not be generated a second time with a particular argument because of the cached result. But since I passed a cache size of 5, then after 5 function calls with different arguments, cache evictions will free up the memoized results and the function will be run again. So the actual output:

```
0.622446623043
```

```
0.622446623043
1.40971776532
1.40971776532
1.56628678698
1.56628678698
2.21064925087
2.21064925087
2.80447028359
2.80447028359
4.05414581689
4.05414581689
0.0251415894698
```

Sweet, it works (The last call with an argument of 1 does not match the initial two function calls).

To do just a bit more code analysis, a typical problem you might run into with memoizing results in Python is two-fold:

Since you can make function calls to python using any combination of arguments and keyword arguments (if keyword arguments are allowable for a function), it might be difficult to find a way to map all of the possible argument permutations to the same result. I dealt with this just by not mapping them to the same result at all. This shouldn't be a problem because code is likely to use the same combination of arguments (between arguments and keyword arguments) anyway, and this won't create problems with memory since we've already limited how large the cache can grow.

The other issue is that you can only hash immutable objects. So if a function parameter happens to be a list or a set or dictionary or anything else that can't be hashed, you need to account for that. I handled this just by casting everything as a string and hashing that value.

## The End

← Previous post Next post →