Zearin / **python_decorator_guide.md**

Last active 9 hours ago

The best explanation of Python decorators I've ever seen. (An archived answer from StackOverflow.)

python_decorator_guide.md

> ***NOTE:*** *This is a question I found on StackOverflow which I've archived here, because the answer is so effing phenomenal.*

Q: How can I make a chain of function decorators in Python?

If you are not into long explanations, see Paolo Bergantino's answer.

Decorator Basics

Python's functions are objects

To understand decorators, you must first understand that functions are objects in Python. This has important consequences. Let's see why with a simple example :

```
def shout(word='yes'):
    return word.capitalize() + '!'

print shout()
# outputs : 'Yes!'

# As an object, you can assign the function to a variable like any
# other object

scream = shout

# Notice we don't use parentheses: we are not calling the function, we are
# putting the function `shout` into the variable `scream`.
# It means you can then call `shout` from `scream`:

print scream()
# outputs : 'Yes!'

# More than that, it means you can remove the old name `shout`, and
# the function will still be accessible from `scream`

del shout
try:
    print shout()
except NameError as e:
    print e
    #outputs: "name 'shout' is not defined"

print scream()
# outputs: 'Yes!'
```

Okay! Keep this in mind. We'll circle back to it shortly.

Another interesting property of Python functions is they can be defined... inside another function!

```
def talk():

    # You can define a function on the fly in `talk` ...
    def whisper(word='yes'):
        return word.lower() + '...'

    # ... and use it right away!

    print whisper()

# You call `talk`, that defines `whisper` EVERY TIME you call it, then
# `whisper` is called in `talk`.
talk()
# outputs:
# "yes..."

# But `whisper` DOES NOT EXIST outside `talk`:

try:
    print whisper()
except NameError as e:
    print e
    #outputs : "name 'whisper' is not defined"*
    Python's functions are objects
```

Functions references

Okay, still here? Now the fun part...

You've seen that functions are objects. Therefore, functions:

- can be assigned to a variable
- can be defined in another function

That means that **a function can `return` another function**. Have a look! ☺️

```
def getTalk(kind='shout'):

    # We define functions on the fly
    def shout(word='yes'):
        return word.capitalize() + '!'

    def whisper(word='yes'):
        return word.lower() + '...'

    # Then we return one of them
    if kind == 'shout':
        # We don't use '()'. We are not calling the function;
        # instead, we're returning the function object
        return shout
    else:
        return whisper

# How do you use this strange beast?

# Get the function and assign it to a variable
talk = getTalk()

# You can see that `talk` is here a function object:
print talk
#outputs : <function shout at 0xb7ea817c>

# The object is the one returned by the function:
print talk()
#outputs : Yes!
```

```
# And you can even use it directly if you feel wild:
print getTalk('whisper')()
#outputs : yes...
```

But wait...there's more!

If you can `return` a function, you can pass one as a parameter:

```
def doSomethingBefore(func):
    print 'I do something before then I call the function you gave me'
    print func()

doSomethingBefore(scream)
#outputs:
#I do something before then I call the function you gave me
#Yes!
```

Well, you just have everything needed to understand decorators. You see, decorators are "wrappers", which means that **they let you execute code before and after the function they decorate** without modifying the function itself.

Handcrafted decorators

How you'd do it manually:

```
# A decorator is a function that expects ANOTHER function as parameter
def my_shiny_new_decorator(a_function_to_decorate):

    # Inside, the decorator defines a function on the fly: the wrapper.
    # This function is going to be wrapped around the original function
    # so it can execute code before and after it.
    def the_wrapper_around_the_original_function():

        # Put here the code you want to be executed BEFORE the original
        # function is called
        print 'Before the function runs'

        # Call the function here (using parentheses)
        a_function_to_decorate()

        # Put here the code you want to be executed AFTER the original
        # function is called
        print 'After the function runs'

    # At this point, `a_function_to_decorate` HAS NEVER BEEN EXECUTED.
    # We return the wrapper function we have just created.
    # The wrapper contains the function and the code to execute before
    # and after. It's ready to use!
    return the_wrapper_around_the_original_function

# Now imagine you create a function you don't want to ever touch again.
def a_stand_alone_function():
    print 'I am a stand alone function, don't you dare modify me'

a_stand_alone_function()
#outputs: I am a stand alone function, don't you dare modify me

# Well, you can decorate it to extend its behavior.
# Just pass it to the decorator, it will wrap it dynamically in
# any code you want and return you a new function ready to be used:

a_stand_alone_function_decorated = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function_decorated()
#outputs:
#Before the function runs
#I am a stand alone function, don't you dare modify me
```

```
#After the function runs
```

Now, you probably want that every time you call `a_stand_alone_function`, `a_stand_alone_function_decorated` is called instead. That's easy, just overwrite `a_stand_alone_function` with the function returned by `my_shiny_new_decorator`:

```
a_stand_alone_function = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function()
#outputs:
#Before the function runs
#I am a stand alone function, don't you dare modify me
#After the function runs

# And guess what? That's EXACTLY what decorators do!
```

Decorators demystified

The previous example, using the decorator syntax:

```
@my_shiny_new_decorator
def another_stand_alone_function():
    print 'Leave me alone'

another_stand_alone_function()
#outputs:
#Before the function runs
#Leave me alone
#After the function runs
```

Yes, that's all, it's that simple. `@decorator` is just a shortcut to:

```
another_stand_alone_function = my_shiny_new_decorator(another_stand_alone_function)
```

Decorators are just a pythonic variant of the decorator design pattern. There are several classic design patterns embedded in Python to ease development (like iterators).

Of course, you can accumulate decorators:

```
def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "<\______/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print '#tomatoes#'
        func()
        print '~salad~'
    return wrapper

def sandwich(food='--ham--'):
    print food

sandwich()
#outputs: --ham--
sandwich = bread(ingredients(sandwich))
sandwich()
#outputs:
#</''''''\>
# #tomatoes#
# --ham--
```

```
# ~salad~
#<\_____/>
```

Using the Python decorator syntax:

```
@bread
@ingredients
def sandwich(food='--ham--'):
    print food

sandwich()
#outputs:
#</''''''\>
# #tomatoes#
# --ham--
# ~salad~
#<\_____/>
```

The order you set the decorators MATTERS:

```
@ingredients
@bread
def strange_sandwich(food='--ham--'):
    print food

strange_sandwich()
#outputs:
##tomatoes#
#</''''''\>
# --ham--
#<\_____/>
# ~salad~
```

Now: to answer the question...

As a conclusion, you can easily see how to answer the question:

```
# The decorator to make it bold
def makebold(fn):
    # The new function the decorator returns
    def wrapper():
        # Insertion of some code before and after
        return '<b>' + fn() + '</b>'
    return wrapper

# The decorator to make it italic
def makeitalic(fn):
    # The new function the decorator returns
    def wrapper():
        # Insertion of some code before and after
        return '<i>' + fn() + '</i>'
    return wrapper

@makebold
@makeitalic
def say():
    return 'hello'

print say()
#outputs: <b><i>hello</i></b>

# This is the exact equivalent to
def say():
```

```
    return 'hello'
say = makebold(makeitalic(say))

print say()
#outputs: <b><i>hello</i></b>
```

You can now just leave happy, or burn your brain a little bit more and see advanced uses of decorators.

Taking decorators to the next level

Passing arguments to the decorated function

```
# It's not black magic, you just have to let the wrapper
# pass the argument:

def a_decorator_passing_arguments(function_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2):
        print 'I got args! Look:', arg1, arg2
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments

# Since when you are calling the function returned by the decorator, you are
# calling the wrapper, passing arguments to the wrapper will let it pass them to
# the decorated function

@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print 'My name is', first_name, last_name

print_full_name('Peter', 'Venkman')
# outputs:
#I got args! Look: Peter Venkman
#My name is Peter Venkman
```

Decorating methods

One nifty thing about Python is that methods and functions are really the same. The only difference is that methods expect that their first argument is a reference to the current object (`self`).

That means you can build a decorator for methods the same way! Just remember to take `self` into consideration:

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3 # very friendly, decrease age even more :-)
        return method_to_decorate(self, lie)
    return wrapper


class Lucy(object):
    def __init__(self):
        self.age = 32

    @method_friendly_decorator
    def sayYourAge(self, lie):
        print 'I am {0}, what did you think?'.format(self.age + lie)

l = Lucy()
l.sayYourAge(-3)
#outputs: I am 26, what did you think?
```

If you're making general-purpose decorator--one you'll apply to any function or method, no matter its arguments--then just use `*args, **kwargs`:

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    # The wrapper accepts any arguments
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print 'Do I have args?:'
        print args
        print kwargs
        # Then you unpack the arguments, here *args, **kwargs
        # If you are not familiar with unpacking, check:
        # http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print 'Python is cool, no argument here.'

function_with_no_argument()
#outputs
#Do I have args?:
#()
#{}
#Python is cool, no argument here.

@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print a, b, c

function_with_arguments(1,2,3)
#outputs
#Do I have args?:
#(1, 2, 3)
#{}
#1 2 3

@a_decorator_passing_arbitrary_arguments
def function_with_named_arguments(a, b, c, platypus='Why not ?'):
    print 'Do {0}, {1} and {2} like platypus? {3}'.format(
    a, b, c, platypus)

function_with_named_arguments('Bill', 'Linus', 'Steve', platypus='Indeed!')
#outputs
#Do I have args ? :
#('Bill', 'Linus', 'Steve')
#{'platypus': 'Indeed!'}
#Do Bill, Linus and Steve like platypus? Indeed!


class Mary(object):
    def __init__(self):
        self.age = 31

    @a_decorator_passing_arbitrary_arguments
    def sayYourAge(self, lie=-3): # You can now add a default value
        print 'I am {0}, what did you think?'.format(self.age + lie)

m = Mary()
m.sayYourAge()
#outputs
# Do I have args?:
#(<__main__.Mary object at 0xb7d303ac>,)
#{}
#I am 28, what did you think?
```

Passing arguments to the decorator

Great, now what would you say about passing arguments to the decorator itself?

This can get somewhat twisted, since a decorator must accept a function as an argument. Therefore, you cannot pass the decorated function's arguments directly to the decorator.

Before rushing to the solution, let's write a little reminder:

```
# Decorators are ORDINARY functions
def my_decorator(func):
    print 'I am an ordinary function'
    def wrapper():
        print 'I am function returned by the decorator'
        func()
    return wrapper

# Therefore, you can call it without any '@'

def lazy_function():
    print 'zzzzzzzz'

decorated_function = my_decorator(lazy_function)
#outputs: I am an ordinary function

# It outputs 'I am an ordinary function', because that's just what you do:
# calling a function. Nothing magic.

@my_decorator
def lazy_function():
    print 'zzzzzzzz'

#outputs: I am an ordinary function
```

It's exactly the same: `my_decorator` is called. So when you `@my_decorator`, you are telling Python to call the function *labelled by the variable "`my_decorator`"*.

This is important! The label you give can point directly to the decorator—**or not**.

Let's get evil. ☺️

```
def decorator_maker():

    print 'I make decorators! I am executed only once: '+\
          'when you make me create a decorator.'

    def my_decorator(func):

        print 'I am a decorator! I am executed only when you decorate a function.'

        def wrapped():
            print ('I am the wrapper around the decorated function. '
                   'I am called when you call the decorated function. '
                   'As the wrapper, I return the RESULT of the decorated function.')
            return func()

        print 'As the decorator, I return the wrapped function.'

        return wrapped

    print 'As a decorator maker, I return a decorator'
    return my_decorator

# Let's create a decorator. It's just a new function after all.
new_decorator = decorator_maker()
#outputs:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator

# Then we decorate the function
```

```
def decorated_function():
    print 'I am the decorated function.'

decorated_function = new_decorator(decorated_function)
#outputs:
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function

# Let's call the function:
decorated_function()
#outputs:
#I am the wrapper around the decorated function. I am called when you call the decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

No surprise here.

Let's do EXACTLY the same thing, but skip all the pesky intermediate variables:

```
def decorated_function():
    print 'I am the decorated function.'
decorated_function = decorator_maker()(decorated_function)
#outputs:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function.

# Finally:
decorated_function()
#outputs:
#I am the wrapper around the decorated function. I am called when you call the decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

Let's make it *even shorter*:

```
@decorator_maker()
def decorated_function():
    print 'I am the decorated function.'
#outputs:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function.

#Eventually:
decorated_function()
#outputs:
#I am the wrapper around the decorated function. I am called when you call the decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

Hey, did you see that? We used a function call with the `@` syntax! :-)

So, back to decorators with arguments. If we can use functions to generate the decorator on the fly, we can pass arguments to that function, right?

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):

    print 'I make decorators! And I accept arguments:', decorator_arg1, decorator_arg2

    def my_decorator(func):
        # The ability to pass arguments here is a gift from closures.
        # If you are not comfortable with closures, you can assume it's ok,
```

```
        # or read: http://stackoverflow.com/questions/13857/can-you-explain-closures-as-they-relate-to-p
        print 'I am the decorator. Somehow you passed me arguments:', decorator_arg1, decorator_arg2

        # Don't confuse decorator arguments and function arguments!
        def wrapped(function_arg1, function_arg2):
            print ('I am the wrapper around the decorated function.\n'
                  'I can access all the variables\n'
                  '\t- from the decorator: {0} {1}\n'
                  '\t- from the function call: {2} {3}\n'
                  'Then I can pass them to the decorated function'
                  .format(decorator_arg1, decorator_arg2,
                          function_arg1, function_arg2))
            return func(function_arg1, function_arg2)

        return wrapped

    return my_decorator

@decorator_maker_with_arguments('Leonard', 'Sheldon')
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ('I am the decorated function and only knows about my arguments: {0}'
           ' {1}'.format(function_arg1, function_arg2))

decorated_function_with_arguments('Rajesh', 'Howard')
#outputs:
#I make decorators! And I accept arguments: Leonard Sheldon
#I am the decorator. Somehow you passed me arguments: Leonard Sheldon
#I am the wrapper around the decorated function.
#I can access all the variables
#       - from the decorator: Leonard Sheldon
#       - from the function call: Rajesh Howard
#Then I can pass them to the decorated function
#I am the decorated function and only knows about my arguments: Rajesh Howard
```

Here it is: a decorator with arguments. Arguments can be set as variable:

```
c1 = 'Penny'
c2 = 'Leslie'

@decorator_maker_with_arguments('Leonard', c1)
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ('I am the decorated function and only knows about my arguments:'
           ' {0} {1}'.format(function_arg1, function_arg2))

decorated_function_with_arguments(c2, 'Howard')
#outputs:
#I make decorators! And I accept arguments: Leonard Penny
#I am the decorator. Somehow you passed me arguments: Leonard Penny
#I am the wrapper around the decorated function.
#I can access all the variables
#       - from the decorator: Leonard Penny
#       - from the function call: Leslie Howard
#Then I can pass them to the decorated function
#I am the decorated function and only knows about my arguments: Leslie Howard
```

As you can see, you can pass arguments to the decorator like any function using this trick. You can even use `*args, **kwargs` if you wish. But remember decorators are called **only once**. Just when Python imports the script. You can't dynamically set the arguments afterwards. When you do `import x`, **the function is already decorated**, so you can't change anything.

Let's practice: decorating a decorator

Okay, as a bonus, I'll give you a snippet to make any decorator accept generically any argument. After all, in order to accept arguments, we created our decorator using another function.

We wrapped the decorator.

Anything else we saw recently that wrapped function?

Oh yes, decorators!

Let’s have some fun and write a decorator for the decorators:

```
def decorator_with_args(decorator_to_enhance):
    """
    This function is supposed to be used as a decorator.
    It must decorate an other function, that is intended to be used as a decorator.
    Take a cup of coffee.
    It will allow any decorator to accept an arbitrary number of arguments,
    saving you the headache to remember how to do that every time.
    """

    # We use the same trick we did to pass arguments
    def decorator_maker(*args, **kwargs):

        # We create on the fly a decorator that accepts only a function
        # but keeps the passed arguments from the maker.
        def decorator_wrapper(func):

            # We return the result of the original decorator, which, after all,
            # IS JUST AN ORDINARY FUNCTION (which returns a function).
            # Only pitfall: the decorator must have this specific signature or it won't work:
            return decorator_to_enhance(func, *args, **kwargs)

        return decorator_wrapper

    return decorator_maker
```

It can be used as follows:

```
# You create the function you will use as a decorator. And stick a decorator on it :-)
# Don't forget, the signature is `decorator(func, *args, **kwargs)`
@decorator_with_args
def decorated_decorator(func, *args, **kwargs):
    def wrapper(function_arg1, function_arg2):
        print 'Decorated with', args, kwargs
        return func(function_arg1, function_arg2)
    return wrapper

# Then you decorate the functions you wish with your brand new decorated decorator.

@decorated_decorator(42, 404, 1024)
def decorated_function(function_arg1, function_arg2):
    print 'Hello', function_arg1, function_arg2

decorated_function('Universe and', 'everything')
#outputs:
#Decorated with (42, 404, 1024) {}
#Hello Universe and everything

# Whoooot!
```

I know, the last time you had this feeling, it was after listening a guy saying: “before understanding recursion, you must first understand recursion”. But now, don’t you feel good about mastering this?

Best practices: decorators

- Decorators were introduced in Python 2.4, so be sure your code will be run on >= 2.4.
- Decorators slow down the function call. Keep that in mind.

- **You cannot un-decorate a function.** (There *are* hacks to create decorators that can be removed, but nobody uses them.) So once a function is decorated, it's decorated *for all the code*.
- Decorators wrap functions, which can make them hard to debug. (This gets better from Python >= 2.5; see below.)

The `functools` module was introduced in Python 2.5. It includes the function `functools.wraps()`, which copies the name, module, and docstring of the decorated function to its wrapper.

(Fun fact: `functools.wraps()` is a decorator! ☺)

```
# For debugging, the stacktrace prints you the function __name__
def foo():
    print 'foo'

print foo.__name__
#outputs: foo

# With a decorator, it gets messy
def bar(func):
    def wrapper():
        print 'bar'
        return func()
    return wrapper

@bar
def foo():
    print 'foo'

print foo.__name__
#outputs: wrapper

# `functools` can help with that

import functools

def bar(func):
    # We say that `wrapper`, is wrapping `func`
    # and the magic begins
    @functools.wraps(func)
    def wrapper():
        print 'bar'
        return func()
    return wrapper

@bar
def foo():
    print 'foo'

print foo.__name__
#outputs: foo
```

How can the decorators be useful?

Now the big question: What can I use decorators for?

Seem cool and powerful, but a practical example would be great. Well, there are 1000 possibilities. Classic uses are extending a function behavior from an external lib (you can't modify it), or for debugging (you don't want to modify it because it's temporary).

You can use them to extend several functions in a DRY's way, like so:

```
def benchmark(func):
    """
    A decorator that prints the time a function takes
    to execute.
```

```
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print func.__name__, time.clock()-t
        return res
    return wrapper


def logging(func):
    """
    A decorator that logs the activity of the script.
    (it actually just prints it, but it could be logging!)
    """
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        print func.__name__, args, kwargs
        return res
    return wrapper


def counter(func):
    """
    A decorator that counts and prints the number of times a function has been executed
    """
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        res = func(*args, **kwargs)
        print '{0} has been used: {1}x'.format(func.__name__, wrapper.count)
        return res
    wrapper.count = 0
    return wrapper

@counter
@benchmark
@logging
def reverse_string(string):
    return str(reversed(string))

print reverse_string('Able was I ere I saw Elba')
print reverse_string('A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale,

#outputs:
#reverse_string ('Able was I ere I saw Elba',) {}
#wrapper 0.0
#wrapper has been used: 1x
#ablE was I ere I saw elbA
#reverse_string ('A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, maca
#wrapper 0.0
#wrapper has been used: 2x
#!amanaP :lanac a ,noep a ,stah eros ,raj a ,hsac ,oloR a ,tur a ,mapS ,snip ,eperc a ,)lemac a ro( niag
```

Of course the good thing with decorators is that you can use them right away on almost anything without rewriting. DRY, I said:

```
@counter
@benchmark
@logging
def get_random_futurama_quote():
    from urllib import urlopen
    result = urlopen('http://subfusion.net/cgi-bin/quote.pl?quote=futurama').read()
    try:
        value = result.split('<br><b><hr><br>')[1].split('<br><br><hr>')[0]
        return value.strip()
    except:
        return 'No, I’m ... doesn’t!'
```

```
print get_random_futurama_quote()
print get_random_futurama_quote()

#outputs:
#get_random_futurama_quote () {}
#wrapper 0.02
#wrapper has been used: 1x
#The laws of science be a harsh mistress.
#get_random_futurama_quote () {}
#wrapper 0.01
#wrapper has been used: 2x
#Curse you, merciful Poseidon!
```

Python itself provides several decorators: `property`, `staticmethod`, etc.

- Django uses decorators to manage caching and view permissions.
- Twisted to fake inlining asynchronous functions calls.

This really is a large playground.

jonahkaks commented on Aug 10

Impressive

noushadkhan01 commented on Aug 23

good job, it's extremely helpful.
Thank you!

jmsmistral commented on Sep 5

Very well explained!

96albert commented on Sep 8

Just awesome ...just thank you

TommyStarK commented on Sep 12

Amazing, thank you very much!

jmartinter commented on Oct 12

Super clear and helpful. Thanks!

huyxdong commented 22 days ago

Thanks bro! Awesome!!!

joetxca commented 17 days ago

Great explanation! Thank you.

soundmasteraj commented 14 days ago

Impressive in knowledge, more impressive in keeping an encouraging tone in the writing, 3x impressive to teach beyond the hard parts, while making the hardest parts as understandable as the other parts. (all the new Alexa SDK for Python now has decorators or Classes...I needed this!) TY 👍