

How do I make function decorators and chain them together?

Asked 13 years, 7 months ago Modified 2 days ago Viewed 623k times

▲ How do I make two decorators in Python that would do the following?

3077



```
@make_bold
@make_italic
def say():
    return "Hello"
```

Calling `say()` should return:

```
"<b><i>Hello</i></b>"
```


[python](#) [function](#) [decorator](#) [python-decorators](#)

Share Edit Follow

edited 2 days ago

 [Martijn Pieters](#) ♦
998k 280 3926
3266

asked Apr 11, 2009 at 7:05

 [Imran](#)
84.4k 23 96 129

1 See also: [What does the "@" \(@\) symbol do in Python?](#) – [Karl Knechtel](#) Aug 13 at 9:54

19 Answers

Sorted by: Highest score (default) ▾

▲ If you are not into long explanations, see [Paolo Bergantino's answer](#).

4673



Decorator Basics



+250



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!'
```

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.**

```
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,
    # we are 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...

```

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

# 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: {0}, {1}".format(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 {0} {1}".format(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: {0},
{1}".format(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: https://stackoverflow.com/questions/13857/can-you-explain-closures-as-they-relate-to-python
        print("I am the decorator. Somehow you passed me arguments: {0},
{1}".format(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 know 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 {0} {1}".format(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 {0} {1}".format(function_arg1, function_arg2))

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

# Whooooot!

```

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 for 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("{0} {1}".format(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("{0} {1} {2}".format(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, macaroni, a gag, a banana bag, a tan, a tag,
a banana bag again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a
jar, sore hats, a peon, a canal: Panama!"))

#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, macaroni, a gag, a banana bag, a tan, a tag, a banana bag
again (or a camel), a crepe, pins, Spam, a rut, a Rolo, cash, a jar, sore hats,
a peon, a canal: Panama!',) {}
#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( niaga gab ananab a ,gat a ,nat a ,gab ananab a ,gag a
,inoracam ,elacrep ,epins ,spam ,arutaroloc a ,shajar ,soreh ,atsap ,eonac a
,nalp a ,nam A

```

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.

Share Edit Follow


edited Apr 26, 2021 at 21:49

community wiki
36 revs, 23 users 73%
e-satis

25 "You cannot un-decorate a function." - While normally true, it is possible to reach inside the closure in the function return by a decorator (i.e. via its `__closure__` attribute) to pull out the original undecorated function. One example usage is documented in [this answer](#) which covers how it is possible to inject a decorator function in a lower level in limited circumstances. – [metatoaster](#) Oct 22, 2015 at 0:04

10 While this is a great answer, I think it is a bit misleading in some ways. Python's `@decorator` syntax is probably most often used to replace a function with a wrapper closure (as the answer describes). But it can also replace the function with something else. The builtin `property` , `classmethod` and `staticmethod` decorators replace the function with a descriptor, for example. A decorator can also do something with a function, such as saving a reference to it in a registry of some sort, then return it, unmodified, without any wrapper. – [Blckknght](#) Apr 11, 2016 at 13:04 

The only thing missing is how to test a function that has been created as decorator (if that makes sense). – [thoroc](#) Oct 21, 2021 at 12:26

1 `functools.wraps` sets the attribute `__wrapped__` on the wrapper function, to allow retrieving the original wrapped function. That's more reliable than looking at closed-over variables. – [merwok](#) Dec 2, 2021 at 17:22 

Adding on to what Blckknght said, a decorator doesn't just return something else, it can return any object. – [Mous](#) May 24 at 21:48

▲ Check out [the documentation](#) to see how decorators work. Here is what you asked for:

3062



```
from functools import wraps

def makebold(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        return "<b>" + fn(*args, **kwargs) + "</b>"
    return wrapper

def makeitalic(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        return "<i>" + fn(*args, **kwargs) + "</i>"
    return wrapper

@makebold
@makeitalic
def hello():
    return "hello world"

@makebold
@makeitalic
def log(s):
```

```
return s

print hello()      # returns "<b><i>hello world</i></b>"
print hello.__name__ # with functools.wraps() this returns "hello"
print log('hello') # returns "<b><i>hello</i></b>"
```

Share Edit Follow

edited Feb 24, 2021 at 14:21



Clint Eastwood

4,826 2 30 27

answered Apr 11, 2009 at 7:16



Paolo Bergantino

474k 79 518 436

273 Consider using [functools.wraps](#) or, better yet, the [decorator module from PyPI](#): they preserve certain important metadata (such as `__name__` and, speaking about the decorator package, function signature). – [Marius Gedminas](#) Mar 11, 2011 at 2:30

34 `*args` and `**kwargs` should be added in the answer. Decorated function can have arguments, and they will be lost if not specified. – [Blusky](#) Apr 2, 2017 at 11:47

4 Although this answer has the great advantage of only using the `stdlib`, and works for this simple example where there are no *decorator* arguments nor *decorated function* arguments, it has 3 major limitations: (1) no simple support for optional decorator arguments (2) not signature-preserving (3) no simple way to extract a named argument from `*args`, `**kwargs`. An easy way to solve these 3 issues at once is to use `decopatch` as explained [here](#). You can also use `decorator` as already mentioned by Marius Gedminas, to solve points 2 and 3. – [smarie](#) Mar 11, 2019 at 15:28



Alternatively, you could write a factory function which return a decorator which wraps the return value of the decorated function in a tag passed to the factory function. For example:

154



```
from functools import wraps

def wrap_in_tag(tag):
    def factory(func):
        @wraps(func)
        def decorator():
            return '<%(tag)s>%(rv)s</%(tag)s>' % (
                {'tag': tag, 'rv': func()})
        return decorator
    return factory
```

This enables you to write:

```
@wrap_in_tag('b')
@wrap_in_tag('i')
def say():
    return 'hello'
```

or

```
makebold = wrap_in_tag('b')
makeitalic = wrap_in_tag('i')

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

Personally I would have written the decorator somewhat differently:

```
from functools import wraps

def wrap_in_tag(tag):
    def factory(func):
```



```
@wraps(func)
def decorator(val):
    return func('<%(tag)s>%(val)s</%(tag)s>' %
               {'tag': tag, 'val': val})
    return decorator
return factory
```

which would yield:

```
@wrap_in_tag('b')
@wrap_in_tag('i')
def say(val):
    return val
say('hello')
```

Don't forget the construction for which decorator syntax is a shorthand:

```
say = wrap_in_tag('b')(wrap_in_tag('i')(say))
```

Share Edit Follow

edited Apr 11, 2009 at 9:29

community wiki
2 revs
Trevor

6 In my opinion, it is better to avoid more than one decorator as far as possible. If i had to write a factory function i would code it with *kwargs like def wrap_in_tag(*kwargs) then @wrap_in_tag('b','i') – gunneysus Oct 29, 2013 at 22:29

Decorators are just syntactical sugar.

140 This

```
@decorator
def func():
    ...
```



expands to

```
def func():
    ...
func = decorator(func)
```

Share Edit Follow

edited Aug 26 at 2:08

Mateen Ulhaq
22.7k 16 88 128

answered Apr 11, 2009 at 8:00

Unknown
45.2k 26 137 181

3 This is so elegant, simple, easy to understand. 10000 upvotes for you, Sir Ockham. – eric Sep 2, 2017 at 17:01

5 Great and simple answer. Would like to add that when using @decorator() (instead of @decorator) it is syntactic sugar for func = decorator()(func). This is also common practice when you need to generate decorators "on the fly" – Omer Dagan Sep 28, 2017 at 11:00

@OmerDagan this is not syntactic sugar, but just regular python code. In the generator (after the @ sign) you can put a regular python expression that yields a decorator function. – leo848 Dec 16, 2021 at 8:07

In func = decorator(func), must the variable name be func which is also the original function name? Can var = decorator(func) also work? – Gathide May 28 at 4:36

Yes. The left hand side variable name and the right hand side variable name need not be the same; `decorator(func)` is assigned to `var` . If they are the same then `func` is overwritten. Else not. – [Anirban Mukherjee](#) Oct 6 at 1:51

▲ And of course you can return lambdas as well from a decorator function:

74



```
def makebold(f):
    return lambda: "<b>" + f() + "</b>"
def makeitalic(f):
    return lambda: "<i>" + f() + "</i>"

@makebold
@makeitalic
def say():
    return "Hello"

print say()
```

Share Edit Follow

answered Oct 25, 2010 at 6:18



[Rune Kaagaard](#)
6,423 2 36 27

13 And one step further: `makebold = lambda f : lambda "" + f() + ""` – [Robφ](#) Mar 4, 2013 at 22:26

15 @Robφ: To be syntactically correct: `makebold = lambda f: lambda: "" + f() + ""` – [martineau](#) Dec 20, 2013 at 16:01

13 Late to the party, but I really would suggest `makebold = lambda f: lambda *a, **k: "" + f(*a, **k) + ""` – [seequ](#) Feb 6, 2015 at 18:19

1 This needs `functools.wraps` in order to not discard the docstring / signature / name of `say` – [Eric](#) Sep 17, 2018 at 6:58

Well, what matters is whether it's mentioned in your answer. Having `@wraps` somewhere else on this page isn't going to help me when I print `help(say)` and get *"Help on function <lambda>"* instead of *"Help on function say"*. – [Eric](#) Sep 18, 2018 at 18:02

▲ Python decorators add extra functionality to another function

66



An italics decorator could be like

```
def makeitalic(fn):
    def newFunc():
        return "<i>" + fn() + "</i>"
    return newFunc
```

Note that a function is defined inside a function. What it basically does is replace a function with the newly defined one. For example, I have this class

```
class foo:
    def bar(self):
        print "hi"
    def foobar(self):
        print "hi again"
```

Now say, I want both functions to print "---" after and before they are done. I could add a print "---" before and after each print statement. But because I don't like repeating myself, I will make a decorator

```
def addDashes(fn): # notice it takes a function as an argument
    def newFunction(self): # define a new function
        print "---"
```

```
fn(self) # call the original function
print "---"
return newFunction
# Return the newly defined function - it will "replace" the original
```

So now I can change my class to

```
class foo:
    @addDashes
    def bar(self):
        print "hi"

    @addDashes
    def foobar(self):
        print "hi again"
```

For more on decorators, check <http://www.ibm.com/developerworks/linux/library/l-cpdecor.html>

Share Edit Follow

answered Apr 11, 2009 at 7:19

 [Abhinav Gupta](#)

4,510 2 22 18


Note as elegant as the lambda functions proposed by @Rune Kaagaard – [rds](#) Aug 19, 2011 at 10:46

1 @Phoenix: The self argument is needed because the newFunction() defined in addDashes() was specifically designed to be a method decorator not a general function decorator. The self argument represents the class instance and is passed to class methods whether they use it or not -- see the section titled **Decorating methods** in @e-satis's answer. – [martineau](#) Jul 12, 2013 at 15:46




1 Print the output as well please. – [user1767754](#) May 26, 2015 at 17:38

Missing functools.wraps – [Eric](#) Sep 17, 2018 at 6:58

The link to the IBM website is out-of-date clickbait. Please update the link or delete it. It goes nowhere except into the Big Blue Linux Developer Hole. Thank you.
– [Rich Lysakowski PhD](#) Nov 27, 2021 at 3:26


44

You *could* make two separate decorators that do what you want as illustrated directly below. Note the use of *args, **kwargs in the declaration of the wrapped() function which supports the decorated function having multiple arguments (which isn't really necessary for the example say() function, but is included for generality).

For similar reasons, the functools.wraps decorator is used to change the meta attributes of the wrapped function to be those of the one being decorated. This makes error messages and embedded function documentation (func.__doc__) be those of the decorated function instead of wrapped() 's.

```
from functools import wraps

def makebold(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return "<b>" + fn(*args, **kwargs) + "</b>"
    return wrapped

def makeitalic(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return "<i>" + fn(*args, **kwargs) + "</i>"
    return wrapped

@makebold
@makeitalic
```

```
def say():
    return 'Hello'

print(say()) # -> <b><i>Hello</i></b>
```

Refinements

As you can see there's a lot of duplicate code in these two decorators. Given this similarity it would be better for you to instead make a generic one that was actually a *decorator factory*—in other words, a decorator function that makes other decorators. That way there would be less code repetition—and allow the [DRY](#) principle to be followed.

```
def html_deco(tag):
    def decorator(fn):
        @wraps(fn)
        def wrapped(*args, **kwargs):
            return '<%s>' % tag + fn(*args, **kwargs) + '</%s>' % tag
        return wrapped
    return decorator

@html_deco('b')
@html_deco('i')
def greet(whom=''):
    return 'Hello' + ( ' ' + whom) if whom else ''

print(greet('world')) # -> <b><i>Hello world</i></b>
```

To make the code more readable, you can assign a more descriptive name to the factory-generated decorators:

```
makebold = html_deco('b')
makeitalic = html_deco('i')

@makebold
@makeitalic
def greet(whom=''):
    return 'Hello' + ( ' ' + whom) if whom else ''

print(greet('world')) # -> <b><i>Hello world</i></b>
```

or even combine them like this:

```
makebolditalic = lambda fn: makebold(makeitalic(fn))

@makebolditalic
def greet(whom=''):
    return 'Hello' + ( ' ' + whom) if whom else ''

print(greet('world')) # -> <b><i>Hello world</i></b>
```

Efficiency

While the above examples do all work, the code generated involves a fair amount of overhead in the form of extraneous function calls when multiple decorators are applied at once. This may not matter, depending the exact usage (which might be I/O-bound, for instance).

If speed of the decorated function is important, the overhead can be kept to a single extra function call by writing a slightly different decorator factory-function which implements adding all the tags at once, so it can generate code that avoids the additional function calls incurred by using separate decorators for each tag.

This requires more code in the decorator itself, but this only runs when it's being applied to function definitions, not later when they themselves are called. This also applies when creating more readable names by using `lambda` functions as previously illustrated. Sample:

```
def multi_html_deco(*tags):
    start_tags, end_tags = [], []
    for tag in tags:
        start_tags.append('<%s>' % tag)
        end_tags.append('</%s>' % tag)
    start_tags = ''.join(start_tags)
    end_tags = ''.join(reversed(end_tags))

    def decorator(fn):
        @wraps(fn)
        def wrapped(*args, **kwargs):
            return start_tags + fn(*args, **kwargs) + end_tags
        return wrapped
    return decorator

makebolditalic = multi_html_deco('b', 'i')

@makebolditalic
def greet(whom=''):
    return 'Hello' + ( ' ' + whom) if whom else ''

print(greet('world')) # -> <b><i>Hello world</i></b>
```

Share Edit Follow

edited Oct 17, 2019 at 12:14

answered May 17, 2015 at 3:26



[martineau](#)

116k 25 161 289



Another way of doing the same thing:

23



```
class bol(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        return "<b>{}</b>".format(self.f())

class ita(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        return "<i>{}</i>".format(self.f())

@bol
@ita
def sayhi():
    return 'hi'
```

Or, more flexibly:

```
class sty(object):
    def __init__(self, tag):
        self.tag = tag
    def __call__(self, f):
        def newf():
            return "<{tag}>{res}</{tag}>".format(res=f(), tag=self.tag)
        return newf

@sty('b')
@sty('i')
```

```
def sayhi():  
    return 'hi'
```

Share Edit Follow

edited Oct 19, 2014 at 18:47



ROMANIA_engineer
52.4k 27 197 193

answered Dec 26, 2011 at 6:13



qed
21.7k 20 116 189

Needs functools.update_wrapper in order to keep sayhi.__name__ == "sayhi" – Eric Sep 17, 2018 at 6:57



22



You want the following function, when called:



```
@makebold  
@makeitalic  
def say():  
    return "Hello"
```

To return:

```
<b><i>Hello</i></b>
```

Simple solution

To most simply do this, make decorators that return lambdas (anonymous functions) that close over the function (closures) and call it:

```
def makeitalic(fn):  
    return lambda: '<i>' + fn() + '</i>'  
  
def makebold(fn):  
    return lambda: '<b>' + fn() + '</b>'
```

Now use them as desired:

```
@makebold  
@makeitalic  
def say():  
    return 'Hello'
```

and now:

```
>>> say()  
'<b><i>Hello</i></b>'
```

Problems with the simple solution

But we seem to have nearly lost the original function.

```
>>> say
<function <lambda> at 0x4ACFA070>
```

To find it, we'd need to dig into the closure of each lambda, one of which is buried in the other:

```
>>> say.__closure__[0].cell_contents
<function <lambda> at 0x4ACFA030>
>>> say.__closure__[0].cell_contents.__closure__[0].cell_contents
<function say at 0x4ACFA730>
```

So if we put documentation on this function, or wanted to be able to decorate functions that take more than one argument, or we just wanted to know what function we were looking at in a debugging session, we need to do a bit more with our wrapper.

Full featured solution - overcoming most of these problems

We have the decorator `wraps` from the `functools` module in the standard library!

```
from functools import wraps

def makeitalic(fn):
    # must assign/update attributes from wrapped function to wrapper
    # __module__, __name__, __doc__, and __dict__ by default
    @wraps(fn) # explicitly give function whose attributes it is applying
    def wrapped(*args, **kwargs):
        return '<i>' + fn(*args, **kwargs) + '</i>'
    return wrapped

def makebold(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return '<b>' + fn(*args, **kwargs) + '</b>'
    return wrapped
```

It is unfortunate that there's still some boilerplate, but this is about as simple as we can make it.

In Python 3, you also get `__qualname__` and `__annotations__` assigned by default.

So now:

```
@makebold
@makeitalic
def say():
    """This function returns a bolded, italicized 'hello'"""
    return 'Hello'
```

And now:

```
>>> say
<function say at 0x14BB8F70>
>>> help(say)
Help on function say in module __main__:

say(*args, **kwargs)
    This function returns a bolded, italicized 'hello'
```

Conclusion

So we see that `wraps` makes the wrapping function do almost everything except tell us exactly what the function takes as arguments.

There are other modules that may attempt to tackle the problem, but the solution is not yet in the standard library.

Share Edit Follow

edited Dec 5, 2016 at 17:33

answered Dec 3, 2015 at 18:09



Russia Must Remove
Putin ♦

356k 85 395 329



A decorator takes the function definition and creates a new function that executes this function and transforms the result.

15



```
@deco
def do():
    ...
```

is equivalent to:

```
do = deco(do)
```

Example:

```
def deco(func):
    def inner(letter):
        return func(letter).upper() #upper
    return inner
```

This

```
@deco
def do(number):
    return chr(number) # number to letter
```

is equivalent to this

```
def do2(number):
    return chr(number)
```

```
do2 = deco(do2)
```

65 <=> 'a'

```
print(do(65))
print(do2(65))
>>> B
>>> B
```

To understand the decorator, it is important to notice, that decorator created a new function `do` which is inner that executes function and transforms the result.



This answer has long been answered, but I thought I would share my Decorator class which makes writing new decorators easy and compact.

11



```
from abc import ABCMeta, abstractclassmethod

class Decorator(metaclass=ABCMeta):
    """ Acts as a base class for all decorators """

    def __init__(self):
        self.method = None

    def __call__(self, method):
        self.method = method
        return self.call

    @abstractclassmethod
    def call(self, *args, **kwargs):
        return self.method(*args, **kwargs)
```

For one I think this makes the behavior of decorators very clear, but it also makes it easy to define new decorators very concisely. For the example listed above, you could then solve it as:

```
class MakeBold(Decorator):
    def call():
        return "<b>" + self.method() + "</b>"

class MakeItalic(Decorator):
    def call():
        return "<i>" + self.method() + "</i>"

@MakeBold()
@MakeItalic()
def say():
    return "Hello"
```

You could also use it to do more complex tasks, like for instance a decorator which automatically makes the function get applied recursively to all arguments in an iterator:

```
class ApplyRecursive(Decorator):
    def __init__(self, *types):
        super().__init__()
        if not len(types):
            types = (dict, list, tuple, set)
        self._types = types

    def call(self, arg):
        if dict in self._types and isinstance(arg, dict):
            return {key: self.call(value) for key, value in arg.items()}

        if set in self._types and isinstance(arg, set):
            return set(self.call(value) for value in arg)

        if tuple in self._types and isinstance(arg, tuple):
            return tuple(self.call(value) for value in arg)

        if list in self._types and isinstance(arg, list):
            return list(self.call(value) for value in arg)

        return self.method(arg)
```

```
@ApplyRecursive(tuple, set, dict)
def double(arg):
    return 2*arg

print(double(1))
print(double({'a': 1, 'b': 2}))
print(double({1, 2, 3}))
print(double((1, 2, 3, 4)))
print(double([1, 2, 3, 4, 5]))
```

Which prints:

```
2
{'a': 2, 'b': 4}
{2, 4, 6}
(2, 4, 6, 8)
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

Notice that this example didn't include the `list` type in the instantiation of the decorator, so in the final print statement the method gets applied to the list itself, not the elements of the list.

Share Edit Follow

edited Nov 11, 2018 at 15:02

answered Nov 6, 2018 at 17:09



[halfer](#)
19.7k 17 92 179



[v4gil](#)
793 9 15



10



```
#decorator.py
def makeHtmlTag(tag, *args, **kwds):
    def real_decorator(fn):
        css_class = " class='{0}'".format(kwds["css_class"]) \
            if "css_class" in kwds else ""
        def wrapped(*args, **kwds):
            return "<"+tag+css_class+">" + fn(*args, **kwds) + "</"+tag+">"
        return wrapped
    # return decorator dont call it
    return real_decorator

@makeHtmlTag(tag="b", css_class="bold_css")
@makeHtmlTag(tag="i", css_class="italic_css")
def hello():
    return "hello world"

print hello()
```

You can also write decorator in Class

```
#class.py
class makeHtmlTagClass(object):
    def __init__(self, tag, css_class=""):
        self._tag = tag
        self._css_class = " class='{0}'".format(css_class) \
            if css_class != "" else ""

    def __call__(self, fn):
        def wrapped(*args, **kwargs):
            return "<" + self._tag + self._css_class+">" \
                + fn(*args, **kwargs) + "</" + self._tag + ">"
        return wrapped

@makeHtmlTagClass(tag="b", css_class="bold_css")
```

```
@makeHtmlTagClass(tag="i", css_class="italic_css")
def hello(name):
    return "Hello, {}".format(name)

print hello("Your name")
```

Share Edit Follow

edited Jul 24, 2017 at 14:14

answered Apr 3, 2014 at 9:43



martineau
116k 25 161 289



nickleefly
3,713 1 26 31

- 1 The reason to like a class here is that there is clearly related behavior, with two instances. You can actually get your two decorators by assigning the constructed classes to the names you wanted, rather than re-iterating the parameters. This is harder to do with a function. Adding it to the example would point out why this is not just redundant. – [Jon Jay Obermark](#) Aug 28, 2014 at 15:46



Here is a simple example of chaining decorators. Note the last line - it shows what is going on under the covers.

8



```
#####
#
#   decorators
#
#####

def bold(fn):
    def decorate():
        # surround with bold tags before calling original function
        return "<b>" + fn() + "</b>"
    return decorate

def uk(fn):
    def decorate():
        # swap month and day
        fields = fn().split('/')
        date = fields[1] + "/" + fields[0] + "/" + fields[2]
        return date
    return decorate

import datetime
def getDate():
    now = datetime.datetime.now()
    return "%d/%d/%d" % (now.day, now.month, now.year)

@bold
def getBoldDate():
    return getDate()

@uk
def getUkDate():
    return getDate()

@bold
@uk
def getBoldUkDate():
    return getDate()

print getDate()
print getBoldDate()
print getUkDate()
print getBoldUkDate()
# what is happening under the covers
print bold(uk(getDate))()
```

The output looks like:

```
17/6/2013
<b>17/6/2013</b>
6/17/2013
<b>6/17/2013</b>
<b>6/17/2013</b>
```

Share Edit Follow

answered Jun 17, 2013 at 4:43



[resigned](#)

1,024 1 10 11

Speaking of the counter example - as given above, the counter will be shared between all functions that use the decorator:

```
def counter(func):
    def wrapped(*args, **kws):
        print 'Called #%i' % wrapped.count
        wrapped.count += 1
        return func(*args, **kws)
    wrapped.count = 0
    return wrapped
```

That way, your decorator can be reused for different functions (or used to decorate the same function multiple times: `func_counter1 = counter(func); func_counter2 = counter(func)`), and the counter variable will remain private to each.

Share Edit Follow

answered Mar 2, 2012 at 21:47



[marqueeed](#)

1,063 10 12

Decorate functions with different number of arguments:

```
def frame_tests(fn):
    def wrapper(*args):
        print "\nStart: %s" %(fn.__name__)
        fn(*args)
        print "End: %s\n" %(fn.__name__)
    return wrapper

@frame_tests
def test_fn1():
    print "This is only a test!"

@frame_tests
def test_fn2(s1):
    print "This is only a test! %s" %(s1)

@frame_tests
def test_fn3(s1, s2):
    print "This is only a test! %s %s" %(s1, s2)

if __name__ == "__main__":
    test_fn1()
    test_fn2('OK!')
    test_fn3('OK!', 'Just a test!')
```

Result:

```
Start: test_fn1
This is only a test!
End: test_fn1

Start: test_fn2
This is only a test! OK!
End: test_fn2

Start: test_fn3
This is only a test! OK! Just a test!
End: test_fn3
```

Share Edit Follow

edited Jun 20, 2020 at 9:12



1 1

answered Apr 5, 2013 at 18:18



rabin utam

13k 3 20 15

-
- 1 This could easily be made even more versatile by also providing support for keyword arguments via `def wrapper(*args, **kwargs):` and `fn(*args, **kwargs)` .
– [martineau](#) May 17, 2015 at 2:33 ✎
-



7



[Paolo Bergantino's answer](#) has the great advantage of only using the `stdlib`, and works for this simple example where there are no *decorator* arguments nor *decorated function* arguments.

However it has 3 major limitations if you want to tackle more general cases:

- as already noted in several answers, you can not easily modify the code to **add optional decorator arguments**. For example creating a `makestyle(style='bold')` decorator is non-trivial.
- besides, wrappers created with `@functools.wraps` **do not preserve the signature**, so if bad arguments are provided they will start executing, and might raise a different kind of error than the usual `TypeError` .
- finally, it is quite difficult in wrappers created with `@functools.wraps` to **access an argument based on its name**. Indeed the argument can appear in `*args` , in `**kwargs` , or may not appear at all (if it is optional).

I wrote [decopatch](#) to solve the first issue, and wrote [makefun.wraps](#) to solve the other two. Note that `makefun` leverages the same trick than the famous [decorator](#) lib.

This is how you would create a decorator with arguments, returning truly signature-preserving wrappers:

```
from decopatch import function_decorator, DECORATED
from makefun import wraps

@function_decorator
def makestyle(st='b', fn=DECORATED):
    open_tag = "<%s>" % st
    close_tag = "</%s>" % st

    @wraps(fn)
    def wrapped(*args, **kwargs):
        return open_tag + fn(*args, **kwargs) + close_tag

    return wrapped
```

`decopatch` provides you with two other development styles that hide or show the various python concepts, depending on your preferences. The most compact style is the following:

```
from decopatch import function_decorator, WRAPPED, F_ARGS, F_KWARGS

@function_decorator
def makestyle(st='b', fn=WRAPPED, f_args=F_ARGS, f_kwargs=F_KWARGS):
    open_tag = "<%s>" % st
    close_tag = "</%s>" % st
    return open_tag + fn(*f_args, **f_kwargs) + close_tag
```

In both cases you can check that the decorator works as expected:

```
@makestyle
@makestyle('i')
def hello(who):
    return "hello %s" % who

assert hello('world') == '<b><i>hello world</i></b>'
```

Please refer to the [documentation](#) for details.

Share Edit Follow

answered Mar 11, 2019 at 15:24



[smarie](#)

4,054 21 35



I add a case when you need to add custom parameters in decorator, pass it to final function and then work it with.

0

the very decorators:



```
def jwt_or_redirect(fn):
    @wraps(fn)
    def decorator(*args, **kwargs):
        ...
        return fn(*args, **kwargs)
    return decorator

def jwt_refresh(fn):
    @wraps(fn)
    def decorator(*args, **kwargs):
        ...
        new_kwargs = {'refreshed_jwt': 'xxxxx-xxxxxx'}
        new_kwargs.update(kwargs)
        return fn(*args, **new_kwargs)
    return decorator
```

and the final function:

```
@app.route('/')
@jwt_or_redirect
@jwt_refresh
def home_page(*args, **kwargs):
    return kwargs['refreched_jwt']
```

Share Edit Follow

answered Apr 24, 2021 at 21:42



[Alexey Nikonov](#)

4,273 3 33 57



With `make_bold()` and `make_italic()` below:

0

▼

🔖

🔄

```
def make_bold(func):
    def core(*args, **kwargs):
        result = func(*args, **kwargs)
        return "<b>" + result + "</b>"
    return core

def make_italic(func):
    def core(*args, **kwargs):
        result = func(*args, **kwargs)
        return "<i>" + result + "</i>"
    return core
```

You can use them as decorators with `say()` as shown below:

```
@make_bold
@make_italic
def say():
    return "Hello"

print(say())
```

Output:

<i>Hello</i>

And of course, you can directly use `make_bold()` and `make_italic()` without decorators as shown below:

```
def say():
    return "Hello"

f1 = make_italic(say)
f2 = make_bold(f1)
result = f2()
print(result)
```

In short:

```
def say():
    return "Hello"

result = make_bold(make_italic(say))()
print(result)
```

Output:

<i>Hello</i>

Share Edit Follow

edited Nov 12 at 13:50

answered Nov 12 at 13:18



[Kai - Kazuya Ito](#)
12k 7 72 92



Yet another example of nested decorators for plotting an image:

0



```
import matplotlib.pyplot as plt

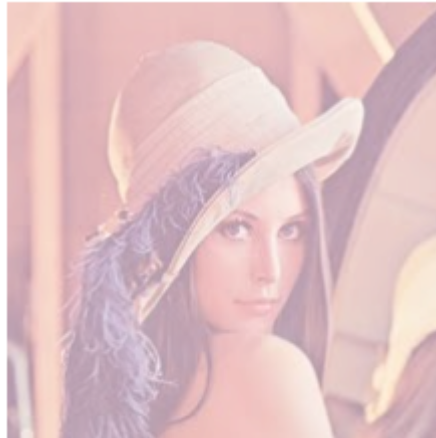
def remove_axis(func):
    def inner(img, alpha):
        plt.axis('off')
        func(img, alpha)
    return inner

def plot_gray(func):
    def inner(img, alpha):
        plt.gray()
        func(img, alpha)
    return inner

@remove_axis
@plot_gray
def plot_image(img, alpha):
    plt.imshow(img, alpha=alpha)
    plt.show()
```

Now, let's show a color image first without axis labels using the nested decorators:

```
plot_image(plt.imread('lena_color.jpg'), 0.4)
```



Next, let's show a gray scale image without axis labels using the nested decorators `remove_axis` and `plot_gray` (we need to `cmap='gray'` , otherwise the default colormap is `viridis` , so a grayscale image is by default not displayed in black and white shades, unless explicitly specified)

```
plot_image(plt.imread('lena_bw.jpg'), 0.8)
```



The above function call reduces down to the following nested call

```
remove_axis(plot_gray(plot_image))(img, alpha)
```


Share Edit Follow

edited Mar 21 at 6:02

answered Mar 20 at 11:13



Sandipan Dey

20.5k 2 44 58
