

# The Code Ship



Sailing through a sea of code

General<sup>5</sup>

Deployment<sup>1</sup>

Web Development<sup>9</sup>

Algorithms<sup>1</sup>

GNU/Linux<sup>2</sup>

Patterns<sup>1</sup>

Tools<sup>1</sup>



Subscribe in a reader



Subscribe via email

Blog by **Ayman Farhat**. Built with **Django**  
and hosted on **Digital Ocean**.

2 years, 10 months ago

## A guide to Python's function decorators

Fork me on GitHub

Python is rich with powerful features and expressive syntax. One of my favorites is decorators. In the context of design patterns, decorators dynamically alter the functionality of a function, method or class without having to directly use subclasses. This is ideal when you need to extend the functionality of functions that you don't want to modify. We can implement the decorator pattern anywhere, but Python facilitates the implementation by providing much more expressive features and syntax for that.

In this post I will be discussing Python's function decorators in depth, accompanied by a bunch of examples on the way to clear up the concepts. All examples are in Python 2.7 but the same concepts should apply to Python 3 with some change in the syntax.

Essentially, decorators work as wrappers, modifying the behavior of the code before and after a target function execution, without the need to modify the function itself, augmenting the original functionality, thus decorating it.

### What you need to know about functions

Before diving in, there are some prerequisites that should be clear. In Python, functions are first class citizens, they are objects and that means we can do a lot of useful stuff with them.

#### Assign functions to variables

```
def greet(name):
    return "hello " + name

greet_someone = greet
print greet_someone("John")

# Outputs: hello John
```

#### Define functions inside other functions

```
def greet(name):
    def get_message():
        return "Hello "

    result = get_message() + name
    return result

print greet("John")

# Outputs: Hello John
```

#### Functions can be passed as parameters to other functions

```
def greet(name):
    return "Hello " + name

def call_func(func):
    other_name = "John"
    return func(other_name)

print call_func(greet)

# Outputs: Hello John
```

## Functions can return other functions

In other words, functions generating other functions.

```
def compose_greet_func():
    def get_message():
        return "Hello there!"

    return get_message

greet = compose_greet_func()
print greet()

# Outputs: Hello there!
```

## Inner functions have access to the enclosing scope

More commonly known as a **closure**. A very powerful pattern that we will come across while building decorators. Another thing to note, Python only allows **read access to the outer scope** and not assignment. Notice how we modified the example above to read a "name" argument from the enclosing scope of the inner function and return the new function.

```
def compose_greet_func(name):
    def get_message():
        return "Hello there "+name+"!"

    return get_message

greet = compose_greet_func("John")
print greet()

# Outputs: Hello there John!
```

## Composition of Decorators

Function decorators are simply wrappers to existing functions. Putting the ideas mentioned above together, we can build a decorator. In this example let's consider a function that wraps the string output of another function by **p** tags.

```
def get_text(name):
    return "lorem ipsum, {0} dolor sit amet".format(name)

def p_decorate(func):
    def func_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return func_wrapper

my_get_text = p_decorate(get_text)

print my_get_text("John")

# <p>Outputs lorem ipsum, John dolor sit amet</p>
```

That was our first decorator. A function that takes another function as an argument, generates a new function, augmenting the work of the original function, and returning the generated function so we can use it anywhere. To have `get_text` itself be decorated by `p_decorate`, we just have to assign `get_text` to the result of `p_decorate`.

```

get_text = p_decorate(get_text)

print get_text("John")

# Outputs lorem ipsum, John dolor sit amet

```

Another thing to notice is that our decorated function takes a name argument. All what we had to do in the decorator is to let the wrapper of `get_text` pass that argument.

## Python's Decorator Syntax

Python makes creating and using decorators a bit cleaner and nicer for the programmer through some **syntactic sugar**. To decorate `get_text` we don't have to `get_text = p_decorator(get_text)`. There is a neat shortcut for that, which is to mention the name of the decorating function before the function to be decorated. The name of the decorator should be prepended with an `@` symbol.

```

def p_decorate(func):
    def func_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return func_wrapper

@p_decorate
def get_text(name):
    return "lorem ipsum, {0} dolor sit amet".format(name)

print get_text("John")

# Outputs <p>lorem ipsum, John dolor sit amet</p>

```

Now let's consider we wanted to decorate our `get_text` function by 2 other functions to wrap a `div` and `strong` tag around the string output.

```

def p_decorate(func):
    def func_wrapper(name):
        return "<p>{0}</p>".format(func(name))
    return func_wrapper

def strong_decorate(func):
    def func_wrapper(name):
        return "<strong>{0}</strong>".format(func(name))
    return func_wrapper

def div_decorate(func):
    def func_wrapper(name):
        return "<div>{0}</div>".format(func(name))
    return func_wrapper

```

With the basic approach, decorating `get_text` would be along the lines of

```

get_text = div_decorate(p_decorate(strong_decorate(get_text)))

```

With Python's decorator syntax, same thing can be achieved with much more expressive power.

```

@div_decorate
@p_decorate
@strong_decorate
def get_text(name):
    return "lorem ipsum, {0} dolor sit amet".format(name)

print get_text("John")

# Outputs <div><p><strong>lorem ipsum, John dolor sit amet</strong></p></div>

```

One important thing to notice here is that the order of setting our decorators matters. If the order was different in the example above, the output would have been different.

## Decorating Methods

In Python, methods are functions that expect their first parameter to be a reference to the current object. We can build decorators for methods the same way, while taking **self** into consideration in the wrapper function.

```

def p_decorate(func):
    def func_wrapper(self):
        return "<p>{0}</p>".format(func(self))
    return func_wrapper

class Person(object):
    def __init__(self):
        self.name = "John"
        self.family = "Doe"

    @p_decorate
    def get_fullname(self):
        return self.name+" "+self.family

my_person = Person()
print my_person.get_fullname()

```

A much better approach would be to make our decorator useful for functions and methods alike. This can be done by putting **\*args** and **\*\*kwargs** as parameters for the wrapper, then it can accept any arbitrary number of arguments and keyword arguments.

```

def p_decorate(func):
    def func_wrapper(*args, **kwargs):
        return "<p>{0}</p>".format(func(*args, **kwargs))
    return func_wrapper

class Person(object):
    def __init__(self):
        self.name = "John"
        self.family = "Doe"

    @p_decorate
    def get_fullname(self):
        return self.name+" "+self.family

my_person = Person()

print my_person.get_fullname()

```

## Passing arguments to decorators

Looking back at the example before the one above, you can notice how redundant the decorators in the example are. 3 decorators(`div_decorate`, `p_decorate`, `strong_decorate`) each with the same

functionality but wrapping the string with different tags. We can definitely do much better than that. Why not have a more general implementation for one that takes the tag to wrap with as a string? Yes please!

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator

@tags("p")
def get_text(name):
    return "Hello " + name

print get_text("John")

# Outputs <p>Hello John</p>
```

It took a bit more work in this case. Decorators expect to receive a function as an argument, that is why we will have to build a function that takes those extra arguments and generate our decorator on the fly. In the example above **tags**, is our decorator generator.

## Debugging decorated functions

At the end of the day decorators are just wrapping our functions, in case of debugging that can be problematic since the wrapper function does not carry the name, module and docstring of the original function. Based on the example above if we do:

```
print get_text.__name__
# Outputs func_wrapper
```

The output was expected to be **get\_text** yet, the attributes `__name__`, `__doc__`, and `__module__` of **get\_text** got overridden by those of the wrapper(func\_wrapper). Obviously we can re-set them within func\_wrapper but Python provides a much nicer way.

## Functools to the rescue

Fortunately Python (as of version 2.5) includes the **functools** module which contains **functools.wraps**. Wraps is a decorator for updating the attributes of the wrapping function(func\_wrapper) to those of the original function(get\_text). This is as simple as decorating func\_wrapper by `@wraps(func)`. Here is the updated example:

```

from functools import wraps

def tags(tag_name):
    def tags_decorator(func):
        @wraps(func)
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
        return func_wrapper
    return tags_decorator

@tags("p")
def get_text(name):
    """returns some text"""
    return "Hello " + name

print get_text.__name__ # get_text
print get_text.__doc__ # returns some text
print get_text.__module__ # __main__

```

You can notice from the output that the attributes of `get_text` are the correct ones now.

## Where to use decorators

The examples in this post are pretty simple relative to how much you can do with decorators. They can give so much power and elegance to your program. In general, decorators are ideal for extending the behavior of functions that we don't want to modify. For a great list of useful decorators I suggest you check out the [Python Decorator Library](#)

## More reading resources

Here is a list of other resources worth checking out:

- - [What is a decorator?](#)
- - [Decorators I: Introduction to Python Decorators](#)
- - [Python Decorators II: Decorator Arguments](#)
- - [Python Decorators III: A Decorator-Based Build System](#)
- - [Guide to: Learning Python Decorators by Matt Harrison](#)

## Phew!

That was an introduction to the concept of decorators in Python. I hope that you found this post helpful, if you have any suggestions or questions please do share them in the comments. Happy coding!

python patterns decorators syntax functions

↑ [Back to top](#)

142

44 Comments    The Code Ship

 Login ▾

♥ Recommend 44     Share

Sort by Best ▾



Join the discussion...



skeller88 • 2 years ago



Great post! Really helped me.

Correction:

```
get_text = p_decorate(get_text)
```

```
print get_text("John")
```

```
# outputs "<p>lorem ipsum, John dolor sit amet</p>"
```

19 ^ | v • Reply • Share ›



**ITSecMedia** → skeller88 • a year ago

I've been missing the p in the output too ^^

^ | v • Reply • Share ›



**x01dev** • 2 years ago

Great article! If you want to start lower because you can't wrap your head around Python decorators, start from <http://x01dev.wordpress.com/20...> and then come back here to read the more-in-depth followup!

8 ^ | v • Reply • Share ›



**bob tongkon** • 2 years ago

Great explanation of python decorator, really helped.  
Thanks.

2 ^ | v • Reply • Share ›



**Hugo Sousa** • 2 years ago

Very good article. Almost one year working with Python and finally I really understand decorators and how useful they can be.

2 ^ | v • Reply • Share ›



**David** • a year ago

Hi, I converted this to a Jupyter notebook: <https://gist.github.com/Prooff...>

1 ^ | v • Reply • Share ›



**Michael Pelts** • 2 years ago

Decorators can also be implemented as classes, that provides a nice alternative for passing arguments with `__init__`. Check out this example:

<http://stackoverflow.com/a/337...>

1 ^ | v • Reply • Share ›



**David Chen** → Michael Pelts • a year ago

im guessing also as instance methods? for example, im looking at `@app.route` for flask and app would probably be a class instance.

^ | v • Reply • Share ›



**Miten Mehta** • 2 years ago

Nice explanation. You can try to add how the decorator generator is expanded. how the syntax `@` is expanded. if the arguments are passed then the `@` needs a decorator generator else it will take decorator. I am just guessing so for others benefit you can clarify.

```
get_text = tags("p")(get_text)
```

```
print get_text("John")
```

1 ^ | v • Reply • Share ›



**jaco** • 2 years ago

Answers that.

Thank you for well written explanation.

1 ^ | v • Reply • Share ›

**Chris Kavanagh** • 2 years ago

Very nice decorator tutorial!

1 ^ | v • Reply • Share ›

**Michael Abramovich** • a month ago

Amazing post!

^ | v • Reply • Share ›

**Rakesh Maddala** • 2 months ago

Great Introduction to Decorators. Thanks for posting.

^ | v • Reply • Share ›

**Logan** • 2 months agoFor more on decorators visit : <http://helpmeprep.webutu.com/>

^ | v • Reply • Share ›

**Opinions** • 2 months ago

Great introduction to decorators. Simple and clear.

^ | v • Reply • Share ›

**JmeHsieh** • 2 months ago

may i ask what's the colorscheme of your python codes here?

^ | v • Reply • Share ›

**JmeHsieh** • 2 months ago

may i ask what's the colorscheme of your python codes here?

^ | v • Reply • Share ›

**Louis-Mathieu Brochu** • 3 months ago

Thank you for this post. FYI there's small error in a phrase in the last paragraph of the section "Debugging decorated functions":

`"[...]the wrapper(func_wrapper."`

^ | v • Reply • Share ›

**Ajith Panneerselvam** • 6 months ago

Great effort! Explained in lucid manner.

^ | v • Reply • Share ›

**bing han** • 6 months ago

why hasn't correct the issue...It's been two years.

^ | v • Reply • Share ›

**Thiago Silvino** • 8 months ago

Very good explanation. Thanks very much!

^ | v • Reply • Share ›

**Ary Kleinerman** • 8 months ago

Well explained. Thanks for share this tutorial :)

^ | v • Reply • Share ›

**jcatanza** • 8 months ago

Excellent.

^ | v • Reply • Share ›

**Stephane St-Pierre** • 9 months ago





Great stuff!, What about \*args and \*\*kwargs in the scope of multi tag decorator as in :

```
@tags("div")
@tags("p")
@tags("strong")
get_text("Elvis:)", "is", "alive!!!")
```

and what about the concept of temporary skipping a decorator call.

tx.

^ | v • Reply • Share ›



**Hunjae** • 10 months ago

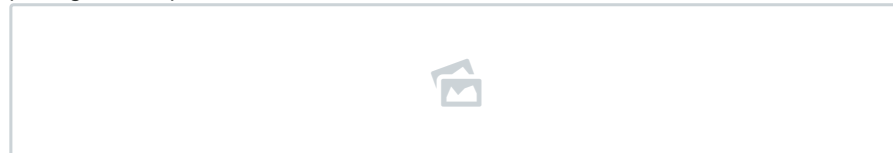
So so great.

^ | v • Reply • Share ›



**Wolf-Dieter Klotz** • a year ago

Best explanation f decorators on the web! Very clear! Thanks. A pity that you stopped posting more explanations to secrets in the sea of code.



^ | v • Reply • Share ›



**nitin** • a year ago

nicely written... and thnx ..

^ | v • Reply • Share ›



**Xeon** • a year ago

very useful tutorial. thanks

^ | v • Reply • Share ›



**thevjm** • a year ago

Thanks for the breakdown. I've been trying to wrap my head around the idea of decorators and this really helped.

^ | v • Reply • Share ›



**lbelin** • a year ago

How to comprehend the argument passing in the last example

^ | v • Reply • Share ›



**Chicken Little** • a year ago

Very good! Thanks

^ | v • Reply • Share ›



**ITSecMedia** • a year ago

You know how to explain things! Thanks for this article.

^ | v • Reply • Share ›



**jitendrakkk** • a year ago

Thank you very much.. :)

^ | v • Reply • Share ›



**whileloop** • a year ago

Thanks. Helped me a lot

^ | v • Reply • Share ›



**Assan Sanogo** • a year ago

**AWESOME**

^ | v • Reply • Share ›

**jobi carter** • a year ago

Excellent explanation of generators! Great use of code without too much fluff explanation. Thanks!

^ | v • Reply • Share ›

**mingwei hu** • a year ago

Great post! Very helpful for me to understand decorators in Python.

^ | v • Reply • Share ›

**Pablo Musumeci** • a year ago

Great article! Thank you for sharing all this.

^ | v • Reply • Share ›

**LinLiang** • 2 years ago

Great post! thanks for share

^ | v • Reply • Share ›

**metatron** • 2 years ago

Wonderful post! THX!

^ | v • Reply • Share ›

**Vadiraj Sagar** • 2 years ago

I still dont understand why do we have to create another function inside the decorator and return it. Why cant we do this:

`def p_decorate(func):` `return "``{0}``".format.get_text(name)`

is it because get\_text is already used in many places and we cant afford to change all those occurances of get\_text invocations?

^ | v • Reply • Share ›

**Giacomo** • 2 years ago

great!!

^ | v • Reply • Share ›

**fayyaz** • 2 years ago

well done and keep it up. great post. it really clear my doubts on decorators. Thanks alot

^ | v • Reply • Share ›

**Alireza Ghaffari** • 2 years ago

nice article!

^ | v • Reply • Share ›

## ALSO ON THE CODE SHIP

**A Guide to Python's function decorators**


8 comments • 3 years ago •

**Christopher Bier** — Thanks for the article!  
Your site looks really nice :) God bless!**The undefined vs null Pitfall in Javascript Den odefinierade vs null ...**


3 comments • 3 years ago •

**GG** — Great explanation, thanks.**Bootstrap your vanilla JS game with this project setup and gulp build****Common pitfalls when working with Javascript Arrays**




1 comment • 4 months ago •

 **jhice** — Hi. Your way to get the minimal loop with `update()` and `draw()` like in the majors game frameworks is great. I'll ...

6 comments • 3 years ago •

 **Tarjei Huse** — Yes, you missed something. `Array.isArray()` is perfectly usable on all modern browsers :)

---

 [Subscribe](#)  [Add Disqus to your site](#) [Add Disqus](#) [Add](#)  [Privacy](#)