1 year, 2 months ago

A guide to Python's function decorators

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)
```

tions

her functions.

```
# 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 perpended 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**

```
get_text = p_decorate(get_text)

print get_text("John")
# outputs "<p>lorem ipsum, John dolor sit amet</p>"
```

13 • Reply • Share ›

x01dev • 6 months 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 • Reply • Share ›

bob tongkon • 4 months ago

Great explanation of python decorator, really helped.
Thanks.

2 • Reply • Share ›

Michael Pelts • 3 months 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 • Reply • Share ›

Miten Mehta • 4 months ago

Nice explanation. You can try to add how the decorator generator is expanded. how the sytanx @ 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 • Reply • Share ›

Hugo Sousa • 4 months ago

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

1 • Reply • Share ›

jaco • 4 months ago

Answers that.
Thank you for well written explanation.

1 • Reply • Share ›

Alireza Ghaffari • a month ago

nice article!

• Reply • Share ›

measuring the memory each approach takes? i think you can in chrome dev tools