

# Python Conquers The Universe

Adventures across space and time with the Python programming language

## Python Decorators

Posted on [2012/04/29](#)

In August 2009, I wrote a post titled [Introduction to Python Decorators](#). It was an attempt to explain Python decorators in a way that I (and I hoped, others) could grok.

Recently I had occasion to re-read that post. It wasn't a pleasant experience — it was pretty clear to me that the attempt had failed.

That failure — and two other things — have prompted me to try again.

- Matt Harrison has published an excellent e-book [Guide to: Learning Python Decorators](#).
- I now have a theory about why most explanations of decorators (mine included) fail, and some ideas about how better to structure an introduction to decorators.

There is an old saying to the effect that “Every stick has two ends, one by which it may be picked up, and one by which it may not.” I believe that most explanations of decorators fail because they pick up the stick by the wrong end.

In this post I will show you what the wrong end of the stick looks like, and point out why I think it is wrong. And I will show you what I think the right end of the stick looks like.

### The wrong way to explain decorators

Most explanations of Python decorators start with an example of a function to be decorated, like this:

```
def aFunction():
```

```
print("inside aFunction")
```

and then add a decoration line, which starts with an @ sign:

```
@myDecorator
def aFunction():
    print("inside aFunction")
```

At this point, the author of the introduction often defines a *decorator* as the line of code that begins with the “@”. (In my older post, I called such lines “annotation” lines. I now prefer the term “decoration” line.)

For instance, in 2008 Bruce Eckel [wrote on his Artima blog](#)

A function decorator is applied to a function definition by placing it on the line before that function definition begins.

and in 2004, Phillip Eby wrote in [an article in Dr. Dobb's Journal](#)

Decorators may appear before any function definition.... You can even stack multiple decorators on the same function definition, one per line.

Now there are two things wrong with this approach to explaining decorators. The first is that the explanation begins in the wrong place. It starts with an example of a function to be decorated and an decoration line, when it should begin with the decorator itself. The explanation should end, not start, with the decorated function and the decoration line. The decoration line is, after all, merely syntactic sugar — is not at all an essential element in the concept of a decorator.

The second is that the term “decorator” is used incorrectly (or ambiguously) to refer both to the decorator and to the decoration line. For example, in his *Dr. Dobb's Journal* article, after using the term “decorator” to refer to the decoration line, Phillip Eby goes on to define a “decorator” as a callable object.

But before you can do that, you first need to have some decorators to stack. A decorator is a callable object (like a function) that accepts one argument—the function being decorated.

So... it would seem that a decorator is both a callable object (like a function) **and** a single line of code that can appear before the line of code that begins a function definition. This is sort of like saying that an “address” is both a building (or apartment) at a specific location **and** a set of lines (written in pencil or ink) on the front of a mailing envelope. The ambiguity may be

almost invisible to someone familiar with decorators, but it is very confusing for a reader who is trying to learn about decorators from the ground up.

## The right way to explain decorators

So how **should** we explain decorators?

Well, we start with the decorator, not the function to be decorated.

### One

We start with the [basic notion of a function](#) — a function is something that generates a value based on the values of its arguments.

### Two

We note that in Python, functions are first-class objects, so they can be passed around like other values (strings, integers, objects, etc.).

### Three

We note that because functions are first-class objects in Python, we can write functions that both (a) accept function objects as argument values, and (b) return function objects as return values. For example, here is a function *foobar* that accepts a function object *original\_function* as an argument and returns a function object *new\_function* as a result.

```
def foobar(original_function):  
  
    # make a new function  
    def new_function():  
        # some code  
  
    return new_function
```

### Four

We define “decorator”.


*A **decorator** is a function (such as foobar in the above example) that takes a function object as an argument, and returns a function object as a return value.*

So there we have it — the definition of a decorator. Anything else that we say about decorators is a refinement of, or an expansion of, or an addition to, this definition of a decorator.

## Five

We show what the internals of a decorator look like. Specifically, we show different ways that a decorator can use the *original\_function* in the creation of the *new\_function*. Here is a simple example.

```
def verbose(original_function):  
  
    # make a new function that prints a message when original_function  
    def new_function(*args, **kwargs):  
        print("Entering", original_function.__name__)  
        original_function(*args, **kwargs)  
        print("Exiting ", original_function.__name__)  
  
    return new_function
```



## Six

We show how to invoke a decorator — how we can pass into a decorator one function object (its input) and get back from it a different function object (its output). In the following example, we pass the *widget\_func* function object to the *verbose* decorator, and we get back a new function object to which we assign the name *talkative\_widget\_func*.

```
def widget_func():  
    # some code  
  
talkative_widget_func = verbose(widget_func)
```

## Seven

We point out that decorators are often used to add features to the *original\_function*. Or more precisely, decorators are often used to create a *new\_function* that does roughly what *original\_function* does, but also does things in addition to what *original\_function* does.

And we note that the output of a decorator is typically used to replace the *original function* that we passed in to the decorator as an argument. A typical use of decorators looks like this. (Note the change to line 4 from the previous example.)

```
def widget_func():  
    # some code  
  
widget_func = verbose(widget_func)
```

So for all practical purposes, in a typical use of a decorator we pass a function (*widget\_func*)

through a decorator (*verbose*) and get back an enhanced (or souped-up, or “decorated”) version of the function.

## Eight

We introduce Python’s “decoration syntax” that uses the “@” to create decoration lines. This feature is basically syntactic sugar that makes it possible to re-write our last example this way:

```
@verbose
def widget_func():
    # some code
```

The result of this example is exactly the same as the previous example — after it executes, we have a *widget\_func* that has all of the functionality of the original *widget\_func*, plus the functionality that was added by the *verbose* decorator.

Note that in **this** way of explaining decorators, the “@” and decoration syntax is one of the **last** things that we introduce, not one of the first.

And we absolutely do **not** refer to line 1 as a “decorator”. We might refer to line 1 as, say, a “decorator invocation line” or a “decoration line” or simply a “decoration”... whatever. But line 1 is **not** a “decorator”.

Line 1 is a line of code. A decorator is a function — a different animal altogether.

## Nine

Once we’ve nailed down these basics, there are a few advanced features to be covered.

- We explain that a decorator need not be a function (it can be any sort of callable, e.g. a class).
- We explain how decorators can be nested within other decorators.
- We explain how ~~decorators~~ decoration lines can be “stacked”. A better way to put it would be: we explain how decorators can be “chained”.
- We explain how additional arguments can be passed to decorators, and how decorators can use them.

## Ten — A decorators cookbook

The material that we've covered up to this point is what any basic Python programmer needs something productive with decorators. He (or she) commentary that describes / shows / explains how to accomplish specific tasks. (Ideally, such as about decorator gotchas and anti-patterns Cookbook" or perhaps "Python Decorator



## Follow "Python Conquers The Universe"

Get every new post delivered to your Inbox.

Join 230 other followers

- As far as I know, no such decorator
- The [Python Decorator Library](#) on It has its uses, but it does not have material of a true cookbook.
- Something similar to a descriptor can be generated by a [search of "descriptor"](#).



Build a website with WordPress.com

o Python  
der to be  
amples, and  
e used to  
s and warnings  
rthon Decorator

for examples.  
anatomy

ally organized,  
[ng on](#)

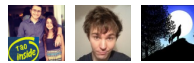
So that's it. I've described what I think is wrong (well, let's say suboptimal) about most introductions to decorators. And I've sketched out what I think is a better way to structure an introduction to decorators.

Now I can explain why I like Matt Harrison's e-book [Guide to: Learning Python Decorators](#). Matt's introduction is structured in the way that I think an introduction to decorators *should* be structured. It picks up the stick by the proper end.

The first two-thirds of the *Guide* hardly talk about decorators at all. Instead, Matt begins with a thorough discussion of how Python functions work. By the time the discussion gets to decorators, we have been given a strong understanding of the internal mechanics of functions. And since most decorators are functions (remember our definition of *decorator*), at that point it is relatively easy for Matt to explain the internal mechanics of decorators.

Which is just as it should be.

*Revised 2012-11-26 — replaced the word "annotation" with "decoration", following terminology ideas discussed in the comments.*

[About these ads](#)

10 bloggers like this.

---

## RELATED

[Introduction to Python Decorators](#)  
In "Decorators"

[enum in Python](#)  
In "Python features"

[Python's magic methods](#)  
In "Python features"

This entry was posted in [Decorators](#) by [Steve Ferg](#). Bookmark the [permalink](#) [<https://pythonconquerstheuniverse.wordpress.com/2012/04/29/python-decorators/>].

32 THOUGHTS ON "PYTHON DECORATORS"



**fmark**

on **2012/05/01 at 5:24 am** said:

I'm yet to read Matt Harrison's guide, but to date my favourite explanation of decorators comes in the form of a [stackoverflow answer](#)



Steve Ferg

on **2012/05/01 at 11:14 am** said:

I agree... that **is** a good explanation!



Matt Williams

on [2012/05/01 at 8:11 am](#) said:

What you refer to as “decorator line”/“decorator invocation line”/“annotation line” could simply be called a “decoration”. The decorator is the function doing the work whereas the decoration is simply the thing that the end users sees in the code.



Steve Ferg

on [2012/05/01 at 11:35 am](#) said:

+1

I like it! It is short and precise and easy to use. And the similarity between the words — “decorator” and “decoration” — reflects the close connection between decorators and decorations.

I think it is worth noting that there is some historical justification for the term “decoration”. [PEP 318](#) does contain the term “decoration”, although it seems to use the terms “decorator” and “decoration” interchangeably.

It might be argued that the similarity between the two words would make it too easy for someone (someone who is trying to learn about decorators) to overlook the important difference between the two terms.

But that possibility would make it imperative for anyone writing an introduction to decorators to explicitly point out the importance of the difference in terminology. And that would be a good thing, because it would force anyone writing an introduction to decorators to explicitly distinguish decorators and decorations, and to be very careful with his terminology. And that would be a Very Good Thing indeed.

I think that one implication of such terminological standardization would be that we should stop saying that lines that begin with `@` use “decorator syntax”. Instead, we should say that such lines use “**decoration syntax**”.





ncoghlan78

on [2012/05/01 at 9:03 am](#) said:

As one of the functools maintainers, I've definitely wrestled with these terminology problems. The terms I use myself are:

- decorator – as you defined it
- decorator expression – what you call a decorator line (\*please\* don't use “annotation”, since annotations already refer to something else in Python 3.x)
- decorator factory – functions like `functools.wraps` and `functools.lru_cache` that accept arguments in the decorator expression and then return the actual decorator to be applied

I believe the fact that many people incorrectly use the term “decorator” to refer to decorator factories like `functools.wraps` helps contribute to the confusion.



Steve Ferg

on [2012/05/01 at 11:54 am](#) said:

+1 on your objection to the term “annotation”.

My first post on this subject was in 2009, when Python 3 was less prominent than it is now. I used the term “annotation” in this post (1) partly for continuity with that earlier post, and (2) partly because now (with the growing use of Python 3) “annotation” is obviously a lousy name. In this post, “annotation” is so in-your-face obnoxious that it is bound to provoke comments with proposals for better terminology.

So thanks for your comment. It is nice to know that I'm not alone in wrestling with these terminological problems. I like your suggestion for “decorator expression”, but I think I like Matt Williams's suggestion of “decoration” (see above) even better.

But the important thing, of course, is that the Python community move toward using **some** standard term — *any term*, as long as it is different from “decorator” — so it will become easier to write clearly about the relationship between decorators and decorations (or decorator expressions).



kent37

on [2012/05/01 at 9:43 am](#) said:

That's pretty much how I explain decorators here:

<http://kentsjohnson.com/kk/00001.html>



Steve Ferg

on [2012/05/01 at 11:23 am](#) said:

Nice explanation. I have one comment, though.

You write that “a decorator is syntactic sugar”.

Since I've taken some pain to distinguish a decorator from a decorator line, I think I would prefer to say that *decoration syntax* (using the @) or a *decoration* or a *decorator line* is syntactic sugar.

Except for that reservation, I think your explanation is pretty good!



pagiannaros

on [2012/05/01 at 9:54 am](#) said:

As you're aiming for rigour it's probably worth changing your fourth step: decorators don't need to return functions. They can return whatever they want.



Steve Ferg

on [2012/05/01 at 3:05 pm](#) said:

This gets to the interesting question of “What (exactly) is a decorator?” or “What is it that makes a function a *decorator* function?”

Let’s do some thought experiments.

### Thought experiment 1

I agree that in Python it certainly is possible to create a function that accepts a function object as an argument and returns something other than a callable.

Suppose, for example, that we write a function that accepts a function object as an argument and returns a string. Maybe something like this.

```
def toString(original_function):  
    return str( original_function() )
```

Now, with just this code available to us, would we want to say that *toString* is a decorator?

### Thought experiment 2

It is also true that we can use a decorator line (a decoration) to invoke such a function.

```
@toString  
def pi_gen():  
    numericPi = some_test_function for generating_p  
i_to_1000_decimal_places()  
    return numericPi
```

With this additional code available to us, would we want to say that *toString* is a decorator? Is being invoked by a decorator line (a decoration) what makes us call a function a “decoration”?

### Thought experiment 3

Suppose we have a program that defines and uses *toString*, but does **not** invoke it using a decorator line.

```
def toString(original_function):
    return str( original_function() )

def pi_gen():
    numericPi = some_test_function for generating_p
i_to_1000_decimal places()
    return numericPi

pi = pi_gen()
pi_as_string = toString(pi_gen)
number_of_decimals = len(pi_as_string) - 2 # don't
count the leading "3."
assert number_of_decimals == 1000
```

In this program, *toString* is certainly **capable** of being invoked via a “@toString” decoration, but in fact in this particular program it is not.

In this program, would we want to say that *toString* is a decorator?

My personal inclination is to say something like this.

We can classify a function as a “decorator” when it meets the following criteria.

- A. A decorator is a function that accepts a function object as an argument and returns a function object as a return value.
- B. We call such a function a “decorator” because it is used to “decorate” other functions.
- C. A function that is used to “decorate” other functions is a “decorator” regardless of how it is invoked in any particular program — regardless, that is, of whether a particular program invokes it via a decoration line or via a normal function call.
- D. A function that accepts a function object as an argument and returns something other than a callable is not a decorator — although it is certainly similar to a decorator in some respects.

So those are my personal intuitions about how we probably want to use the term “decorator”.

I like it because I can explicitly state what seem to me pretty clear, consistent, and intuitive rules for deciding whether or not to call a function a

“decorator”. And I like it because it makes it possible to classify a function as a decorator (or as a non-decorator) independently of the technique that might be used to invoke it in any particular program.

Others may of course prefer to use the word “decorator” differently. If they do, I think I would ask that they explicitly state their rules for using the word “decorator”. That would reduce the chances of miscommunication caused by mere differences in terminology.

Finally I think that a *Python Decorator Cookbook* would be an appropriate place to discuss all functions that we might want to invoke via a decoration line (@ line) — whether or not they return a callable. Maybe we should call it the *Python Decorator and Decoration Cookbook*.



matthewharrison

on **2012/05/01 at 12:25 pm** said:

Steve-

Thanks for your post. It is true, decorator is overloaded. Thanks for pointing that out, I think you are one of the first I recall doing that. I'm in the middle of reviewing the book as I'm preparing for POD and having a pdf/mobi/epub bundle again. You've given me some thoughts on some improvements I can make, so thanks!

A decorator cookbook sounds like an interesting idea. I actually don't write new decorators too often, but I use closures all the time. Decorators just sort of fall out once you have really grasped closures. Sadly I don't think a closure cookbook would be general enough to serve a wide audience. My closures are usually very specific to the problem at hand (changing interface, generating new functions, etc).



Steve Ferg

on **2012/05/01 at 12:51 pm** said:

Thanks!

RE: “Decorators just sort of fall out once you have really grasped closures.”  
Yeah, that’s one of the things that I learned from your book. It really helped me to put decorators into a broader context in which they made a lot more sense.

As you can see from my comments on earlier comments, I’m now inclined to follow Matt Williams’s suggestion (see above) and use the word “decoration” for what I previously called a “decorator line”.

And to replace the expression “decorator syntax” with “decoration syntax”. It is, after all, a specialized syntax for writing decorations, not decorators.

It would be an interesting test and experiment to see what the effect would be of using this terminology in the *Guide*. I’m guessing that it would make things clearer. But you never know with experiments — sometimes they don’t turn out the way you expect them to!



fijiaron

on **2012/05/01 at 4:02 pm** said:

You should start by saying a decorator is a pattern. That will help by warning people that it’s just one way to do a common task. Next you should explain that python decorators are a specific syntax used to identify (but not necessarily implement) the pattern. In other words, “python decorators” are the little @ annotations that specify the function (not decorator) that is used to implement decorator pattern. You can then explain that the pattern is to “decorate” a function with additional functionality but not break its original contract (e.g. perform the original task and/or return the same value.) That’s about all that’s needed.

Then, except in rare cases where the decorator is already implemented and all people have to do is paste the @ call, they can accomplish the same thing in a simpler way (usually by adding a call to a logging function within their own function.)

Frankly, I think your original post did a better job, primarily because it has a real example.

I don’t know why people think if a term is used in a book they haven’t read that they

need to build a big mystique around it. Chances are the book is considered arcane because it's more obtuse than enlightening. A more generous phrasing would be that the knowledge gained from reading the books reflects better on the brainiac able to sift through the muddle than the profundity of the original author(s).



Steve Ferg

on [2012/05/02 at 12:09 am](#) said:

Yup, that might be yet another way of structuring an explanation of decorators. But I wonder how helpful it would be, especially when trying to help newbies understand decorators. It sort of reminds me of the old joke ...

You have a problem — you need to explain decorators.

You decide to start by saying that a decorator is a pattern.

Now you have two problems.



[Craig McQueen](#)

on [2012/05/01 at 7:16 pm](#) said:

It's also worth explaining (clearly) the difference between a decorator that takes parameters, and one that doesn't. I found that confusing at first.



matthewharrison

on [2012/05/03 at 9:07 am](#) said:

I'm convinced that once one understands closures, decorators come quickly. Parameterized decorators are just another closure around a normal decorator. Again, the key here is to understand closures and how they work



Lisin

on **2012/05/01 at 8:55 pm** said:

Being just a beginner who is learning python, I find decorators one of several concepts difficult to grasp. At first I thought that all it was is just like you would assign a number 5 to a variable `num = 5`, is the same with decorators that you assign a function to a variable by using `@num =` assign the below function and pass it to another function??

If my understanding is correct, I dont know how to actually write it or use it as all the examples I have seen so far are rather complex.



TheBlackCat

on **2012/05/03 at 10:18 am** said:

I would say there is one important step missing in the process: you need to explain why decorators are useful. A cookbook would help with that, but having a few practical examples showing exactly why someone would want to use decorators in their own code in the first place, rather than just modifying their functions, is really important, and something that all explanations of decorators that I have seen pretty much completely skip. At least personally it isn't really obvious why I would want to use a decorator.



Steve Ferg

on **2012/05/07 at 12:18 am** said:

I absolutely agree.

In the case of almost all (or all?) current introductions to decorators, once you've read the introduction you've got a solution looking for a problem. It is like being handed a tool without any explanation of what you can do with the tool. Imagine being handed a stick of dynamite without being given any idea of what you could do with it (blast out mines and tunnels and road cuts and canals, kill enemy troops, blow out oil-well fires, etc.). Pretty useless.

That's why I think a cookbook can't just be a list of recipes. I think the recipes have to be organized around problems. Basically, you have to treat each recipe like a design pattern, where each recipe starts with a problem



statement (“Pattern 1234 — You’re in situation such-and-such, and you need to do so-and-so. Here is how you can solve your problem using a decorator.”), and group the patterns into general categories.

In a collection of code examples, there is always a tension around whether to organize the examples around the coding techniques used (“Now we’re going to look at how to pass arguments to decorators.”) or around the kinds of problems to be solved (“Now we will look at various ways to use decorators to add logging capabilities to functions and classes in your programs.”). Most introductions to decorators are organized the first way. But now that we have such introductions, what we really need is a cookbook — or a **collection of decorator design patterns** — organized in the second way, around problems that can be solved (dealt with effectively) using decorators.



Uncle Roastie

on [2012/05/03 at 2:09 pm](#) said:

Bruce Eckel has an excellent description of decorators in “Python 3 Patterns & Idioms”:

<https://bitbucket.org/BruceEckel/python-3-patterns-idioms/downloads>



Raoul

on [2012/05/04 at 10:17 am](#) said:

A simple, functioning example was not given. Why not ?

What good is any explanation without a good, simple example ?



Steve Ferg

on [2012/05/06 at 10:52 pm](#) said:

This post was not an attempt to explain decorators — it was a discussion of how such an explanation should be structured.

For an actual explanation of decorators, you may want to look at the text of my earlier post titled [Introduction to Python Decorators](#) and the links in that post.

For an explanation that has the structure that I recommend (and includes examples!) I recommend Matt Harrison's e-book [Guide to: Learning Python Decorators](#).

Raoul

on [2012/05/07 at 8:52 pm](#) said:

This seems to be a moot point such as “determining how many angels will fit on the head of a pin”. Why try to clearly explain what is clearly an obfuscation: Whatever decorators can do are better done by using simple functions :

```
def FuncInsteadOfDecorator( funcIn, *args, **  
kws ) :  
    funcOut = ...  
    return funcOut
```

Decorators are not “elegant” – they are unnecessarily confusing. They seem to be like a throwback to the dark days of in-line compiler directives. If a factory function is what is wanted, then write one.



**David McLean (@00Davo)**

on [2012/05/27 at 9:35 am](#) said:

Ah, but decorators are simple functions. In fact, that function you've provided as an example of not using decorators *is* a decorator.

You've most likely confused decorators themselves—functions which transform a function to a souped-up version—with Python's decoration syntactic sugar, provided to make decorator usage more convenient.

To expand through example, here's your function:

```
def funcInsteadOfDecorator( funcIn, *args, **kwargs ) :  
    funcOut = ...  
    return funcOut
```

This function is a decorator. It may be applied using regular function syntax:

```
def myFunc() :  
    do stuff in here  
  
myFunc = funcInsteadOfDecorator(myFunc)
```

Alternately, it may be applied using Python decoration syntax:

```
@funcInsteadOfDecorator  
def myFunc() :  
    do stuff in here
```

Both versions are equivalent, and both versions use only the simple function you've provided.



Octopusgrabbus

on **2012/07/30 at 12:10 pm** said:

I liked your explanation including what is wrong with traditional decorator explanations. It seems to me that decorators would be good for logging output before and/or after an original function performs its job, more like allowing the run-time code to change output on the fly.



**mark**

on **2012/08/17 at 10:10 am** said:

Great meta-explanation! Please do not name sections of your article after numbers though.



**Arun Mittal**

on **2012/11/26 at 3:13 pm** said:

This is awesome, i had been trying to understand decorators for long.  
This is the first article, which made me understand decorators in 9 points. Great Work!!!



**Suresh**

on **2012/11/27 at 3:49 am** said:

Rather than a cookbook of decorators, may be attempt at a classification of decorators – as to the various types (intention of usage).

1. To do some standard stuff on entry/exit of a function: Such decorators may be applied to a lots of different functions and is a way of enforcing the DRY principle.
2. To add some functionality to third party/standard library modules: These may be one off.



**Andrew Dalke**

on [2012/11/27 at 6:15 am](#) said:

You write “A decorator is a function (such as foobar in the above example) that takes a function object as an argument, and returns a function object as a return value.”

I believe there is a clearer way to describe this. “A decorator is a function which takes a function as an argument and returns a value – any value. This is often a new function, but it might be the same function, an integer, or even the Python None object. The return value is assigned to a variable with the same name as the function given in the function argument.”

For example, “@apply” “def data(): return 5” is a round-about way to write “data = 5”. I’ve seen it used in real code as a way to initialize a variable where the initialization uses several other helper variables which shouldn’t clutter the module namespace.



Fernando

on [2012/11/28 at 1:48 pm](#) said:

IMHO, step 1 should be why. What problem is solved? It’s only point seven that addresses it and it already comes with a solution. Which confused me for a moment since I didn’t realise decorators was not really a new feature.



participant3

on [2013/04/01 at 10:28 pm](#) said:

tl; dr... your old article was great and I totally got it from that. Thank you.



[Shuaib Mohammad \(@shuaib2m\)](#)

on **2013/04/12 at 4:25 am** said:

This is great Steve! I came across decorators while playing around with django and flask and was curious about what was happening behind the scenes. After some unhelpful explanations, I read Bruce Eckel's article and thought I was getting somewhere. But you really nailed it. I'm really impressed by the way you have deconstructed the concept and structured your explanation.

Comments are closed.

