# Jeff Knupp

PYTHON PROGRAMMER



# Everything I know about Python...



## Improve Your Python: Decorators Explained

I've previously written about "yield" and generators. In that article, I mention it's a topic that novices find confusing. The purpose and creation of **decorators** is another such topic (using them, however, is rather easy). In this post, you'll learn what decorators are, how they're created, and why they're so useful.

## A Brief Aside...

### Passing Functions

Before we get started, recall that *everything* in Python is an object that can be treated like a value (e.g. functions, classes, modules). You can bind names to these objects, pass them as arguments to functions, and return them from functions (among other things). The following code is an example of what I'm talking about:

```
def is_even(value):
    """Return True if *value* is even."""
    return (value % 2) == 0

def count_occurrences(target_list, predicate):
    """Return the number of times applying the callable *predicate* to a
    list element returns True."""
    return sum([1 for e in target_list if predicate(e)])

my_predicate = is_even
my_list = [2, 4, 6, 7, 9, 11]
result = count_occurrences(my_list, my_predicate)
print(result)
```

We've written a function that takes a list and another function (which happens to be a *predicate function*, meaning it returns True or False based on some property of the argument passed to it), and returns the number of times our predicate function holds true for an element in the list. While there are built-in functions to accomplish this, it's useful for illustrative purposes.

The magic is in the lines `my_predicate = is_even`. We bound the name `my_predicate` to the function itself (not the value returned when calling it) and can use it like any "normal" variable. Passing it to `count_occurrences` allows

`count_occurrences` to apply the function to the elements of the list, even though it doesn't "know" what `my_predicate` does. It just assumes it's a function that can be called with a single argument and returns True or False.

Hopefully, this is all old hat to you. If, however, this is the first time you've seen functions used in this manner, I recommend reading Drastically Improve Your Python: Understanding Python's Execution Model before continuing here.

## Returning Functions

We just saw that functions can be passed as arguments to other functions. They can also be *returned* from functions as the return value. The following demonstrates how that might be useful:

```
def surround_with(surrounding):
    """Return a function that takes a single argument and."""
    def surround_with_value(word):
        return '{}{}{}'.format(surrounding, word, surrounding)
    return surround_with_value

def transform_words(content, targets, transform):
    """Return a string based on *content* but with each occurrence
    of words in *targets* replaced with
    the result of applying *transform* to it."""
    result = ''
    for word in content.split():
        if word in targets:
            result += ' {}'.format(transform(word))
        else:
            result += ' {}'.format(word)
    return result

markdown_string = 'My name is Jeff Knupp and I like Python but I do not own a Python'
markdown_string_italicized = transform_words(markdown_string, ['Python', 'Jeff'],
        surround_with('*'))
print(markdown_string_italicized)
```

The purpose of the `transform_words` function is to search `content` for any occurrences of a word in the list of `targets` and apply the `transform` argument to them. In our example, we imagine we have a Markdown string and would like to italicize all occurrences of the words `Python` and `Jeff` (a word is italicized in Markdown when it is surrounded by asterisks).

Here we make use of the fact that functions can be returned as the result of calling a function. In the process, we create a *new* function that, when called, prepends and appends the given argument. We then pass that new function as an argument to `transform_words`, where it is applied to the words in our search list: (`['Python', 'Jeff']`).

You can think of `surround_with` as a little function "factory". It sits there waiting to create a function. You give it a value, and it gives you back a function that will surround a word argument with the value you gave it. Understanding what's happening here is crucial to understanding decorators. Our "function factory" doesn't *ever* return a "normal" value; it always returns a new function. Note that `surround_with` doesn't actually do the surrounding itself, it just creates a function that can do it whenever it's needed.

`surround_with_value` makes use of the fact that nested functions have access to names bound in the scope in which they were created. Therefore, `surround_with_value` doesn't need any special machinery to access `surrounding` (which would defeat the purpose). It simply "knows" it has access to it and uses it when required.

## Putting it all together

We've now seen that functions can both be sent as arguments to a function and returned as the result of a function. What if we made use of both of those facts together? Can we create a function that takes a function as a parameter and returns a function as the result. Would that be useful?

Indeed it would be. Imagine we were using a web framework and have models with lots of currency related fields like `price`, `cart_subtotal`, `savings` etc. Ideally, when we output these fields, we would always prepend a "$". If we could somehow mark functions that produce these values in a way that would do that for us, that would be great.

This is exactly what decorators do. The function below is used to show the `price` with `tax` applied:

```
class Product(db.Model):

    name = db.StringColumn
    price = db.FloatColumn

    def price_with_tax(self, tax_rate_percentage):
        """Return the price with *tax_rate_percentage* applied.
        *tax_rate_percentage* is the tax rate expressed as a float, like
        "7.0" for a 7% tax rate."""
        return price * (1 + (tax_rate_percentage * .01))
```

How can use the language to augment this function so that the return value has a "$" prepended? We create a `decorator` function, which has a useful shorthand notation: `@`. To create our `decorator`, we create a function which takes a function (the function to be decorated) and returns a new function (the original function with decoration applied). Here's how we would do that in our application:

```
def currency(f):
    def wrapper(*args, **kwargs):
        return '$' + str(f(*args, **kwargs))

    return wrapper
```

We include the '*args' and *kwargs' as parameters to the `wrapper` function to make it more flexible. Since we don't know the parameters the function we're wrapping may take (and `wrapper` needs to call that function), we accept all possible positional (`*args`) and keyword (`**args`) arguments as parameters and "forward" them to the function call.

With `currency` defined, we can now use the `decorator` notation to decorate our `price_with_tax` function, like so:

```
class Product(db.Model):

    name = db.StringColumn
    price = db.FloatColumn
```

```
    @currency
    def price_with_tax(self, tax_rate_percentage):
        """Return the price with *tax_rate_percentage* applied.
        *tax_rate_percentage* is the tax rate expressed as a float, like "7.0"
        for a 7% tax rate."""
        return price * (1 + (tax_rate_percentage * .01))
```

Now, to other code, it seems as though `price_with_tax` is a function that returns the price with tax prepended by a dollar sign. Notice, however, that we didn't change any code in `price_with_tax` itself to achieve this. We simply "decorated" the function with a `decorator`, giving it additional functionality.

### Brief aside

One problem (easily solved) is that wrapping `price_with_tax` with `currency` changes its `.__name__` and `.__doc__` to that of `currency`, which is certainly not what we want. The `functools` modules contains a useful tool, `wraps`, which will restore these values to what we would expect them to be. It is used like so:

```
from functools import wraps

def currency(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        return '$' + str(f(*args, **kwargs))

    return wrapper
```

## Raw Power

This notion of wrapping a function with additional functionality without changing the wrapped function is *extremely* powerful and useful. Much can be done with `decorators` that would otherwise require lots of boilerplate code or simply wouldn't be possible. They also act as a convenient way for frameworks and libraries to provide functionality. Flask uses `decorators` as a means for adding new endpoints to the web application, as in this example from the documentation:

```
@app.route('/')
def hello_world():
    return 'Hello World!'
```

Notice that decorators (being functions themselves) can take arguments. I'll save decorator arguments, along with class decorators, for the next article in this series.

# In Closing

Today we learned how decorators can be used to manipulate the language (much like C macros) *using* the language we're manipulating (i.e. Python). This has very powerful implications, which we'll explore in the next article. For now, however, you should have a solid grasp on how the vast majority of decorators are created and used. More importantly, you should now understand how they work and when they're useful.

Posted on Nov 29, 2013 by Jeff Knupp



« Supercharge Your Python Developers

Subscribe

11 Comments jeffknupp.com 1 Login

Recommend 7 Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS

Name

**Sikolia Wycliffe** • a year ago

If your name was Jeff Knupp and you liked python and owned a python. What would that make you?

3 • Reply • Share ›

**RandomOne** • 2 years ago

Great article!
I can't say I have a full understanding of the subject now, but it's much more than I had before reading this post and after reading other explanations about the subject.

1 • Reply • Share ›

**Jason Haas** • 3 years ago

fantastic article! This article on decorators and your post on generators were especially informative!

1 • Reply • Share ›

**Diego Córdova Nieto** • 13 days ago

Hi, Jeff! I'm a novice with Python but I'm doing something with it. And I think decorators can be very useful in order to optimize the solution that I've done. However I do not understand how I can implement a decorator in this case. I only got two functions and I want to embed both of them into another (the decorator). Can I send you the code? It's relatively simple and it has no more than 80 lines of code. Please let me know if you can help me! :) Thanks, and great article by the way!

Reply • Share ›

**João S** • 4 months ago

Great article Jeff, I am glad I found your blog!

Reply • Share ›

**Viet Cao Minh** • a year ago

Great article!

Reply • Share ›

**Vũ Nguyễn** • a year ago

May I ask, why did you write " str(f(*args, **kwargs)) " ?

what does the f() do?

Reply • Share ›

**Thang Duong** ➚ Vũ Nguyễn • a year ago

It depends. In this particular example, f() is price_with_tax(), and it will
return price * (1 + (tax_rate_percentage * .01))

So basically, f() is a decorated function passed as argument to the decorator

Reply • Share ›

**Iryna Tyshko** • 2 years ago

Applying @currency decorator to price_with_tax function changes its return type from int to string.
Here you could redefine and decorate __str__ function to output the value prefixed with '$'.

Reply • Share ›

**Joaquín Bermúdez** • 4 years ago

Another great article. Thanks Jeff. Just two things:
1 - As Julien Tayon said, be careful with floating point and currencies.
2 - There's a typo in the Flask url (the actual url is not loading right now anyway, what a funny coincidence).



**julien tayon** • 4 years ago

If you talk about improving python:
Please use Decimal when presenting monetary values.

```
>>> 38.1 * .2
7.620000000000001
```

Floating points are introducing inaccuracies that can cumulate.