PyCon-IE-2015-Decorators

October 24, 2015

1 Decorators: not that scary after all

1.1 PyCon Ireland 2015 - October 24th 2015 - Dublin, Ireland

1.1.1 By Paul Barry

Lecturer: Institute of Technology, Carlow, Ireland
Author: Head First Python, published by O'Reilly Media (update coming "real soon now")
URL: http://paulbarry.itcarlow.ie
Talk URL (as a PDF): http://paulbarry.itcarlow.ie/pyconie2015/decorators.pdf
or http://bit.ly/1NswdIt

1.2 Introduction

1.2.1 Why?

Because knowing how to effectively use more of Python's language features makes you a better Python programmer.

1.2.2 What is a decorator?

Programming tool to help you adjust the behaviour of pre-existing code (yours or others)
Most people consider such carry-on "scary".
Two types of decorator: function and class.
As we are mainly interested in understanding the basics, we're going to concentrate on function decorators in this talk.

1.2.3 How?

By live coding...

1.3 What you already know

But, before we get going, let's look at some decorators that you may have come across before:
Ever seen or used `@staticmethod`, `@classmethod`, or `@property`? Or maybe `@app.route` in Flask? More on this last one in a little bit...

1.4 What do you need to know about to understand decorators?

You need to be familiar with four things:

1. Functions.
2. Argument processing (including `*args`, `**kwargs`).
3. Functions taking the name of a function as an argument, then calling it.
4. Functions returning a function as a return value, which is then called.

None of this is hard.
Regardless, let's play at the shell for a bit to confirm that we are happy with all of this.

1.5 1. Functions

```
In [1]: def myfunc(a, b):
            return a * b

In [2]: myfunc(10, 20)

Out[2]: 200
```

1.6 2. Argument processing (including *args , **kwargs)

```
In [3]: def myfunc(a, *args):
            for b in args:
                a = a * b
            return a

In [4]: myfunc(10, 20)

Out[4]: 200

In [5]: myfunc(10, 20, 30, 40, 50)

Out[5]: 12000000

In [6]: 10*20*30*40*50

Out[6]: 12000000

In [7]: def myfunc(**kwargs):
            res = 1
            for k, v in kwargs.items():
                res = res * v
            return res

In [8]: myfunc(b=20, a=10, c=40, d=30)

Out[8]: 240000

In [9]: 20*10*40*30

Out[9]: 240000

In [10]: myfunc(b=20, a=10)

Out[10]: 200

In [11]: myfunc(first=1, last=1)

Out[11]: 1

In [12]: 1*1*1

Out[12]: 1
```

1.7 3. Functions taking the name of a function as an argument, then calling it

```
In [13]: def myfunc(func):
             func()

In [14]: def anotherfunc():
             print("I'm another func.")

In [15]: anotherfunc()

I'm another func.

In [16]: myfunc(anotherfunc)

I'm another func.
```

1.8 4. Functions returning a function as a return value, which is then called

```
In [17]: def myfunc():
             def innerfunc():
                 print("I'm the inner func.")
             return innerfunc

In [18]: f = myfunc()

In [19]: f()

I'm the inner func.
```

1.9 Using a decorator to wrap existing functionality

How do you know when to do this?

Good question...

Let's look at a case study using Flask (and we are deliberately going to KISS).

Here's some code for a very simple (minimalist) Flask app.

```
In [ ]: from flask import Flask

        app = Flask(__name__)

        @app.route('/')
        def hello():
            return 'Hello PyCon Ireland 2015!'

        if __name__ == '__main__':
            app.run(debug=True)
```

And it works!

Let's add some additional URLs.

```
In [ ]: from flask import Flask

        app = Flask(__name__)

        @app.route('/')
        def hello():
            return 'Hello PyCon Ireland 2015!'
```

```
@app.route('/page1')
def page1():
    return 'This is page 1.'

@app.route('/page2')
def page2():
    return 'This is page 2.'

@app.route('/page3')
def page3():
    return 'This is page 3.'

if __name__ == '__main__':
    app.run(debug=True)
```

And it works, too!

Let's imagine that we want to arrange that the `/page1`, `/page2`, and `/page3` URLs are to be made available to previously logged in users.

What do we need to do?

1. Provide a mechanism to indicate when someone is logged-in or not.
2. Check if a user is logged-in before letting them access any of the restricted URLs.

Add support for sessions to our webapp, which lets us create the `/login`, `/logout`, and `/status` URLs.

```
In [ ]: from flask import Flask, session

        app = Flask(__name__)

        @app.route('/')
        def hello():
            return 'Hello PyCon Ireland 2015!'

        @app.route('/page1')
        def page1():
            return 'This is page 1.'

        @app.route('/page2')
        def page2():
            return 'This is page 2.'

        @app.route('/page3')
        def page3():
            return 'This is page 3.'

        @app.route('/login')
        def login():
            session['logged_in'] = True
            return 'You are now logged in.'

        @app.route('/logout')
        def logout():
            session.pop('logged_in')
            return 'You are now logged out.'
```

```
@app.route('/status')
def display_status():
    if 'logged_in' in session:
        return 'You are currently logged in.'
    return 'You are NOT logged in.'

app.secret_key = 'YouWillNeverGuess'

if __name__ == '__main__':
    app.run(debug=True)
```

Remember: we want to ensure only logged-in users see pages 1, 2, and 3.

What are our options here?

First "obvious" option is usually to do something like this, which is naive - it's waaaaay too much work, and it hides the real purpose of the `page1()` function (which is lost in the details of all this new code). This, coupled with the fact that we have to add boiler-plate code like this to each of our restricted URLs, makes this a poor strategy going forward:

```
In [ ]: @app.route('/page1')
        def page1():
            if 'logged_in' in session:
                return 'This is page 1.'
            return 'Please log in to continue."
```

1.10 A better approach is to create a decorator, then use it to wrap each "protected" function with the abstracted functionality

Let's start with the logic that we want to extract/abstract:

```
In [ ]: if 'logged_in' in session:
            # Call the decoratored function.
        return 'Please log in to continue.'
```

Instead of that comment, let's arrange to call any function (which takes any amount/type of arguments). When we call the function, arrange to return any results produced:

```
In [ ]: if 'logged_in' in session:
            return func(*args, *kwargs)
        return 'Please log in to continue.'
```

Now let's put this code into a function, which takes any amount/type of arguments:

```
In [ ]: def wrapped_function(*args, **kwargs):
            if 'logged_in' in session:
                return func(*args, *kwargs)
            return 'Please log in to continue.'
```

Let's put this function inside another which returns `wrapped_function` when invoked:

```
In [ ]: def check_loggedin():
            def wrapped_function(*args, **kwargs):
                if 'logged_in' in session:
                    return func(*args, *kwargs)
                return 'Please log in to continue.'
            return wrapped_function
```

We're nearly there.

The `check_loggedin` function needs to be told the name of the function to wrap, and it also needs to handle some stickly argument details. The `functools` library is your friend here:

```
In [ ]: from functools import wraps

        def check_loggedin(func):
            @wraps(func)
            def wrapped_function(*args, **kwargs):
                if 'logged_in' in session:
                    return func(*args, *kwargs)
                return 'Please log in to continue.'
            return wrapped_function
```

Now let's add this code to our webapp, decorate some functions, and see what happens:

```
In [ ]: app = Flask(__name__)

        from functools import wraps

        def check_loggedin(func):
            @wraps(func)
            def wrapped_function(*args, **kwargs):
                if 'logged_in' in session:
                    return func(*args, *kwargs)
                return 'Please log in to continue.'
            return wrapped_function

        @app.route('/')
        def hello():
            return 'Hello PyCon Ireland 2015!'

        @app.route('/page1')
        @check_loggedin
        def page1():
            return 'This is page 1.'

        @app.route('/page2')
        @check_loggedin
        def page2():
            return 'This is page 2.'

        @app.route('/page3')
        @check_loggedin
        def page3():
            return 'This is page 3.'

        @app.route('/login')
        def login():
            session['logged_in'] = True
            return 'You are now logged in.'

        @app.route('/logout')
        @check_loggedin
        def logout():
```

```
        session.pop('logged_in')
        return 'You are now logged out.'

    @app.route('/status')
    def display_status():
        if 'logged_in' in session:
            return 'You are currently logged in.'
        return 'You are NOT logged in.'

    app.secret_key = 'YouWillNeverGuess'

    if __name__ == '__main__':
        app.run(debug=True)
```

And there you have it: a nice, abstracted function decorator which keeps the login check away from your webapp's functions, but still lets you check that only logged in users get to see certain restricted pages.