View Decorators

Python has a really interesting feature called function decorators. This allow some really neat things for web applications. Because each view in Flask is a function decorators can be used to inject additional functionality to one or more functions. The `route()` decorator is the one you probably used already. But there are use cases for implementing your own decorator. For instance, imagine you have a view that should only be used by people that are logged in to. If a user goes to the site and is not logged in, they should be redirected to the login page. This is a good example of a use case where a decorator is an excellent solution.

Login Required Decorator

So let's implement such a decorator. A decorator is a function that returns a function. Pretty simple actually. The only thing you have to keep in mind when implementing something like this is to update the *__name__*, *__module__* and some other attributes of a function. This is often forgotten, but you don't have to do that by hand, there is a function for that that is used like a decorator (`functools.wraps()`).

This example assumes that the login page is called `'login'` and that the current user is stored as *g.user* and *None* if there is no-one logged in:

```python
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

So how would you use that decorator now? Apply it as innermost decorator to a view function. When applying further decorators, always remember that the `route()` decorator is the outermost:

```python
@app.route('/secret_page')
@login_required
def secret_page():
    pass
```

Caching Decorator

Imagine you have a view function that does an expensive calculation and because of that you would like to cache the generated results for a certain amount of time. A decorator would be nice for that. We're assuming you have set up a cache like mentioned in Caching.

Here an example cache function. It generates the cache key from a specific prefix (actually a format string) and the current path of the request. Notice that we are using a function that first creates the decorator that then decorates the function. Sounds awful? Unfortunately it is a little bit more complex, but the code should still be straightforward to read.

The decorated function will then work as follows

1. get the unique cache key for the current request base on the current path.
2. get the value for that key from the cache. If the cache returned something we will return that value.
3. otherwise the original function is called and the return value is stored in the cache for the timeout provided (by default 5 minutes).

Here the code:

```python
from functools import wraps
from flask import request

def cached(timeout=5 * 60, key='view/%s'):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            cache_key = key % request.path
            rv = cache.get(cache_key)
            if rv is not None:
                return rv
            rv = f(*args, **kwargs)
            cache.set(cache_key, rv, timeout=timeout)
            return rv
        return decorated_function
    return decorator
```

Notice that this assumes an instantiated *cache* object is available, see Caching for more information.

Templating Decorator

A common pattern invented by the TurboGears guys a while back is a templating decorator. The idea of that decorator is that you return a dictionary with the values passed to the template from the view function and the template is automatically rendered. With that, the following three examples do exactly the same:

```python
@app.route('/')
def index():
    return render_template('index.html', value=42)

@app.route('/')
@templated('index.html')
def index():
    return dict(value=42)

@app.route('/')
@templated()
def index():
    return dict(value=42)
```

As you can see, if no template name is provided it will use the endpoint of the URL map with dots converted to slashes + '.html'. Otherwise the provided template name is used. When the decorated function returns, the dictionary returned is passed to the template rendering function. If *None* is returned, an empty dictionary is assumed, if something else than a dictionary is returned we return it from the function unchanged. That way you can still use the redirect function or return simple strings.

Here the code for that decorator:

```python
from functools import wraps
from flask import request

def templated(template=None):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            template_name = template
            if template_name is None:
                template_name = request.endpoint \
                    .replace('.', '/') + '.html'
```

```
            ctx = f(*args, **kwargs)
            if ctx is None:
                ctx = {}
            elif not isinstance(ctx, dict):
                return ctx
            return render_template(template_name, **ctx)
        return decorated_function
    return decorator
```

Endpoint Decorator

When you want to use the werkzeug routing system for more flexibility you need to map the endpoint as defined in the **Rule** to a view function. This is possible with this decorator. For example:

```python
from flask import Flask
from werkzeug.routing import Rule

app = Flask(__name__)
app.url_map.add(Rule('/', endpoint='index'))

@app.endpoint('index')
def my_index():
    return "Hello world"
```