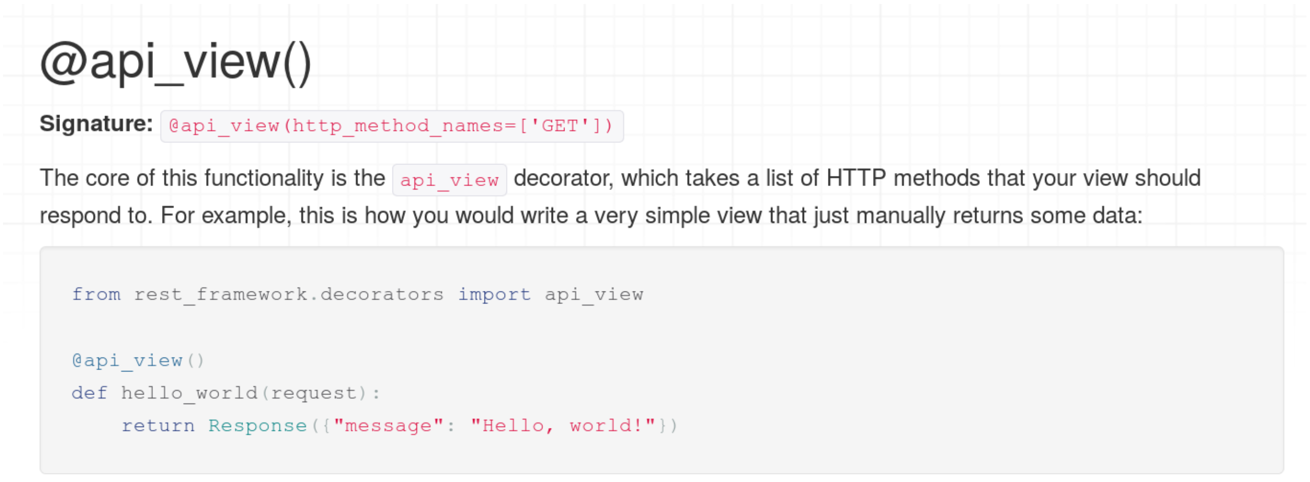
Walter Blair
603 Assignment 6

Decorator Pattern

Given that I've more or less forgotten what my thesis is about in the last few weeks, I figured I'd have some fun by branching out in this patterns assignment. The only place where I've heard of decorators is in regards to RESTful decorators like @POST that you can add to your code to facilitate a RESTful API. I've done this with Salesforce's Apex code in CSIS 659 and also in Python on a personal project. I found an example of this RESTful decorator pattern in the Django REST framework project (http://www.django-rest-framework.org/api-guide/views/). @api_view() is used as a decorator to add callout functionality in your Django code (Snippet 1). The example in Snippet 1 demonstrates the intent of a decorator pattern, which is to add additional responsibilities to the hello_world(request) function below. The example code below is simplified to the point of removing OOD ideas, because this is certainly possible in Python, but the idea is that whatever object likely holds this hello_world function in a real application would possess additional responsibilities, matching our OOD goals.



## @api_view()

**Signature:** `@api_view(http_method_names=['GET'])`

The core of this functionality is the `api_view` decorator, which takes a list of HTTP methods that your view should respond to. For example, this is how you would write a very simple view that just manually returns some data:

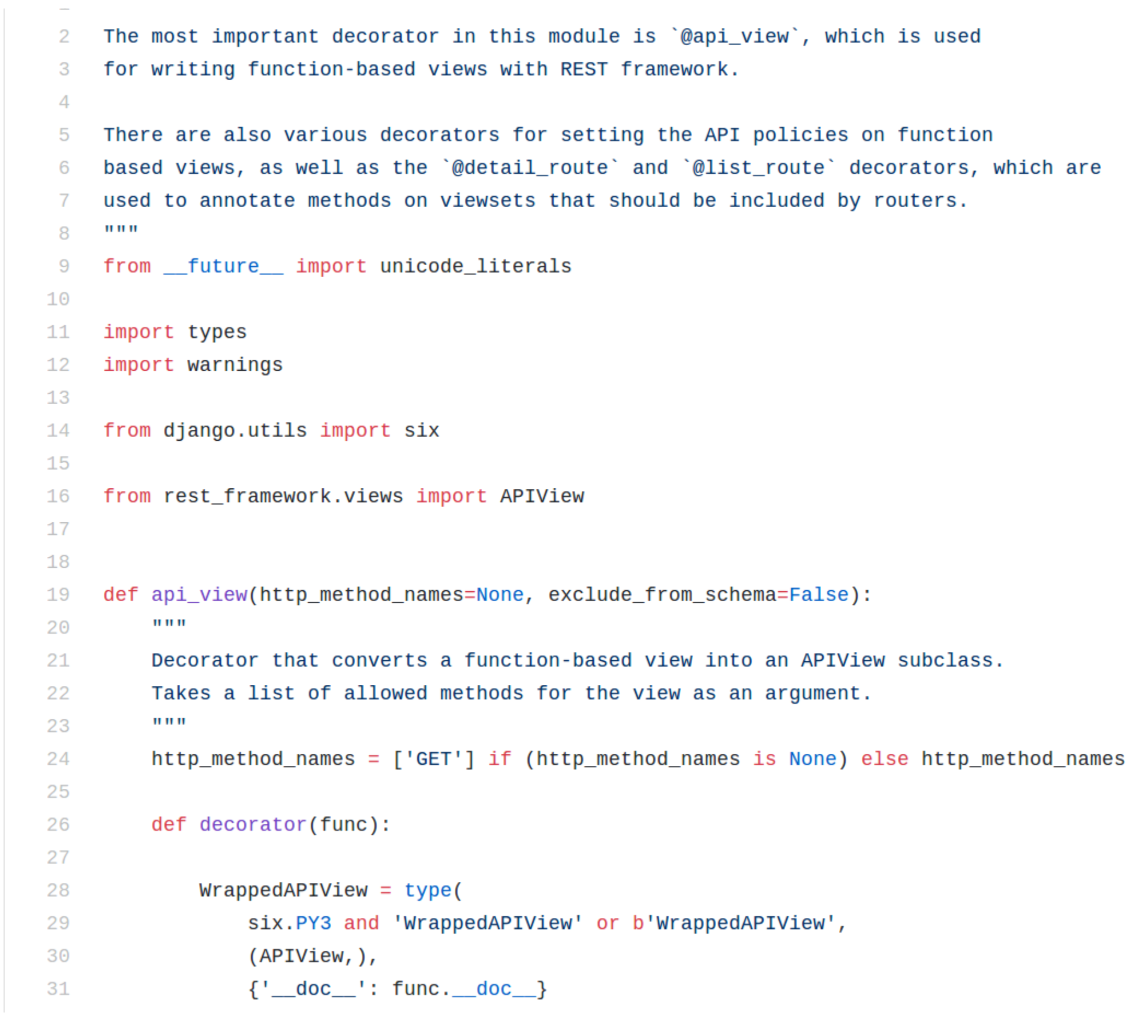
```
from rest_framework.decorators import api_view

@api_view()
def hello_world(request):
    return Response({"message": "Hello, world!"})
```

*Snippet 1. Docs explaining api_view() decorator*

Providing no args to @api_view() defaults to a GET callout, but the full RESTful experience can be achieved by passing CRUD args like 'POST'.

One thing I like about this example is that the project is so well documented and commented. The source code for api_view is very clear on how the decorator pattern is used (Snippet 2). Snippet 2 provides a nice picture of the components in this decorator pattern, despite the fact that Python is not strictly object-oriented. api_view can be seen as the component, because it provides an interface for the decorator functionality and the concrete component. api_view's function decorator() that accepts concrete callout methods would be the decorator, because this is where the desired http methods are specified for the concrete decorator. The hello_world() function is the concrete component, and the enhanced callout functions specified by decorator(args) are the concrete decorators. The decorator(args) function forwards requests to the APIView class to handle the added RESTful functionality.



```python
The most important decorator in this module is `@api_view`, which is used
for writing function-based views with REST framework.

There are also various decorators for setting the API policies on function
based views, as well as the `@detail_route` and `@list_route` decorators, which are
used to annotate methods on viewsets that should be included by routers.
"""
from __future__ import unicode_literals

import types
import warnings

from django.utils import six

from rest_framework.views import APIView


def api_view(http_method_names=None, exclude_from_schema=False):
    """
    Decorator that converts a function-based view into an APIView subclass.
    Takes a list of allowed methods for the view as an argument.
    """
    http_method_names = ['GET'] if (http_method_names is None) else http_method_names

    def decorator(func):

        WrappedAPIView = type(
            six.PY3 and 'WrappedAPIView' or b'WrappedAPIView',
            (APIView,),
            {'__doc__': func.__doc__}
```

*Snippet 2. Source code for api_view() decorator with thorough comments*

The big picture of RESTful services is that they are extremely flexible and easy to use compared to predecessors and alternatives like SOAP. When flexibility and ease of use is desired, decorators often fit the bill. Decorators are used to add certain functionality wherever they are needed without complicating the programmer's life with class inheritance. I'm not exactly sure how mixins like I see in Ruby and Scala compare to decorators, but I do know that they also allow classes to take certain specified functions from other classes without the usual inheritance baggage. Decorators win out in RESTful services over the Strategy pattern, because RESTful API classes are typically lightweight in the sense that they only do a small set of specific tasks, namely execute http callouts. If for some reason you found yourself with a heavy component class, you may want to use the Strategy pattern.