심화학습 Ch8-9

용어 정의

- Class-based Generic View - 현재 제공되고 있는 클래스 형태의 뷰 모음 (Django 1.3부터 제공)
- Function-based Generic View - 예전에 제공되었던 함수 형태의 뷰 모음 (Django 1.2까지 제공)
- Class-based View - 클래스 형태로 작성한 뷰
- Function-based View - 함수 형태로 작성한 뷰

URL, URL, URI 이야기

URL (Uniform Resource Locator)

리소스의 위치를 나타내는 것

`ex: https://8percent.kr`

URN (Uniform Resource Name)

리소스의 이름을 나타내는 것

`ex: urn:isbn:9780692915721`

URI (Uniform Resource Identifier)

리소스를 가리키는 식별자. URL, URN을 포함한다.

URL design

절대 쓰지 마시오!

Vignette-style - 12,123,5123,1.html 처럼 콤마를 여러 개 넣은 url 스타일

파일 확장자 url - `https://8percent.kr/guide.html`

CBV도 FBV와 다르지 않다

```python
@classonlymethod
def as_view(cls, **initkwargs):
    """
    Main entry point for a request-response process.
    """
    for key in initkwargs:
        if key in cls.http_method_names:
            raise TypeError("You tried to pass in the %s method name as a "
                            "keyword argument to %s(). Don't do that."
                            % (key, cls.__name__))
        if not hasattr(cls, key):
            raise TypeError("%s() received an invalid keyword %r. as_view "
                            "only accepts arguments that are already "
                            "attributes of the class." % (cls.__name__, key))

    def view(request, *args, **kwargs):
        self = cls(**initkwargs)
        if hasattr(self, 'get') and not hasattr(self, 'head'):
            self.head = self.get
        self.request = request
        self.args = args
        self.kwargs = kwargs
        return self.dispatch(request, *args, **kwargs)
    view.view_class = cls
    view.view_initkwargs = initkwargs

    # take name and docstring from class
    update_wrapper(view, cls, updated=())

    # and possible attributes set by decorators
    # like csrf_exempt from dispatch
    update_wrapper(view, cls.dispatch, assigned=())
    return view
```

Explicit is better than implicit (The Zen of Python)을 명심하자.

locals()를 쓰지 말아야 할 이유

context data에는 명시적으로 표시를 하도록 합니다.

```python
urlpatterns = patterns('', ...)

for app in settings.INSTALLED_APPS:
    if not app.startswith('django'):
        p = url('^%s/' % app, include('%s.urls') % app)
        urlpatterns += patterns('', p)
```

위처럼 쓰지 말고,

이렇게 쓰세요.

Decorator (장식자)

파이썬 내에서 모든 것들은 일급 객체. 함수도 일급 객체

일급 객체

일급 객체 - 위키백과, 우리 모두의 백과사전

조건

- 변수나 데이터 구조안에 담을 수 있다.
- 파라미터로 전달 할 수 있다.
- 반환값(return value)으로 사용할 수 있다.
- 할당에 사용된 이름과 관계없이 고유한 구별이 가능하다.
- 동적으로 프로퍼티 할당이 가능하다.

고계 함수 (Higher-order function)

함수형 프로그래밍 - 위키백과, 우리 모두의 백과사전

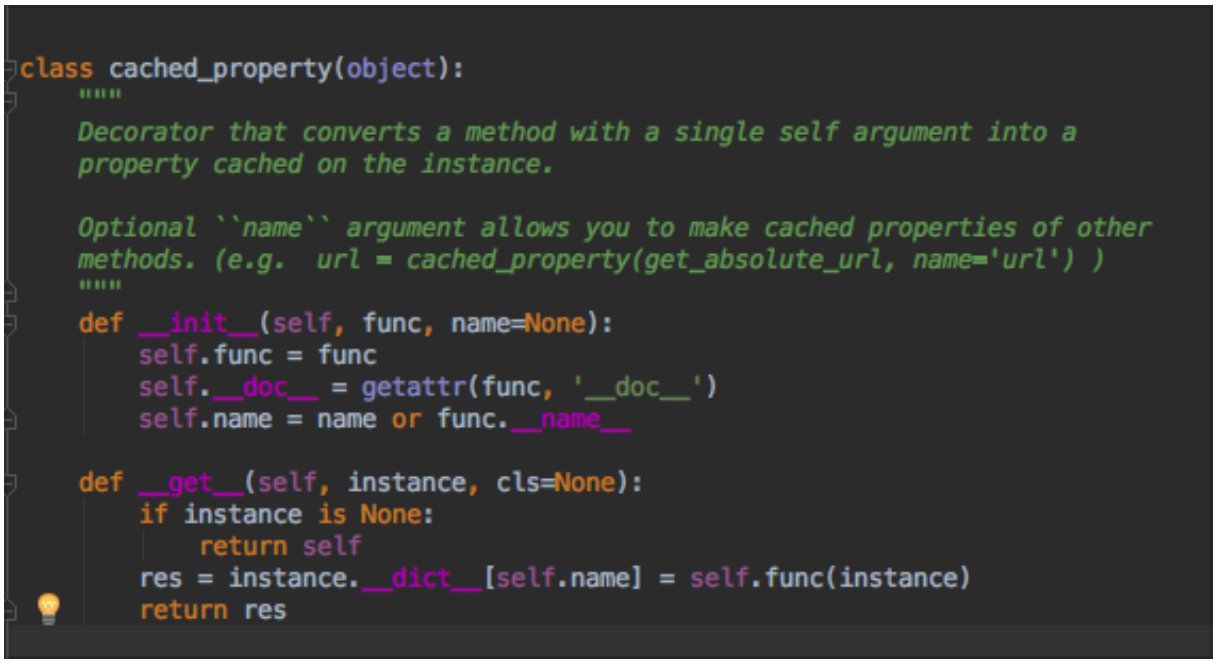
함수를 다루는 함수. 함수가 일급 객체라면, 함수의 인자로 함수를 전달하고, 함수의 결과값으로 함수가 사용 가능함

Decorator 문법

```
def echo(func):
    def wrapper():
        func()
        func()
    return wrapper

def hello():
    print('Hello')

# 첫번째 방법 - 같은 함수명으로 재할당
hello = echo(hello)

# 두번째 방법 - 골뱅이 쓰기
@echo
def hello():
    print('Hello')
```

클래스도 일급 객체. 따라서 클래스 decorator도 가능

`def __get__(self, instance, cls=None)` 을 쓰는 경우

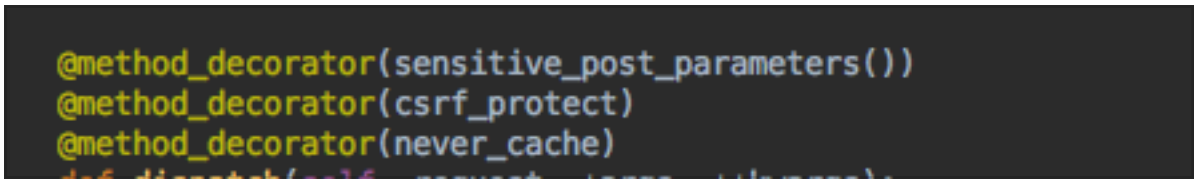


```python
class cached_property(object):
    """
    Decorator that converts a method with a single self argument into a
    property cached on the instance.

    Optional ``name`` argument allows you to make cached properties of other
    methods. (e.g.  url = cached_property(get_absolute_url, name='url') )
    """
    def __init__(self, func, name=None):
        self.func = func
        self.__doc__ = getattr(func, '__doc__')
        self.name = name or func.__name__

    def __get__(self, instance, cls=None):
        if instance is None:
            return self
        res = instance.__dict__[self.name] = self.func(instance)
        return res
```

메소드를 decorating할 때

`def __call__(self):`을 쓰는 경우
일반적인 용도

method_decorator는 무엇인가?



```python
@method_decorator(sensitive_post_parameters())
@method_decorator(csrf_protect)
@method_decorator(never_cache)
```

일반 함수에 적용하는 decorator를 메소드에 적합하도록 변경해주는 데코레이터

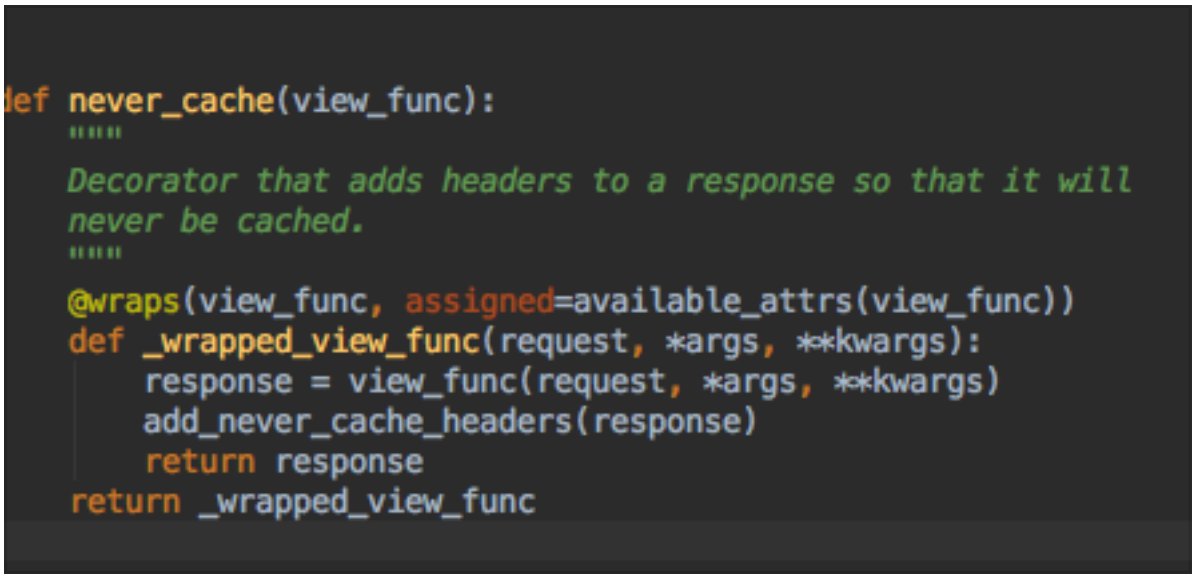
데코레이터 적용 순서는?

`never_cache` ➡ `csrf_protect` ➡ `sensitive_post_parameters`
즉 밑에서부터 위로 적용된다.

sensitive_post_parameters()는 왜 괄호가 있고, csrf_protect, never_cache는 없는가?

`sensitive_post_parameters()`는 인자를 받는 데코레이터이고, 나머지는 인자를 받지 않는다.

먼저 안 받는 경우,



```
def never_cache(view_func):
    """
    Decorator that adds headers to a response so that it will
    never be cached.
    """
    @wraps(view_func, assigned=available_attrs(view_func))
    def _wrapped_view_func(request, *args, **kwargs):
        response = view_func(request, *args, **kwargs)
        add_never_cache_headers(response)
        return response
    return _wrapped_view_func
```

```
dispatch = never_cache(dispatch)
```

인자를 받는 경우,

```python
def sensitive_post_parameters(*parameters):
    """
    Indicates which POST parameters used in the decorated view are sensitive,
    so that those parameters can later be treated in a special way, for example
    by hiding them when logging unhandled exceptions.

    Two forms are accepted:

    * with specified parameters:

        @sensitive_post_parameters('password', 'credit_card')
        def my_view(request):
            pw = request.POST['password']
            cc = request.POST['credit_card']
            ...

    * without any specified parameters, in which case it is assumed that
      all parameters are considered sensitive:

        @sensitive_post_parameters()
        def my_view(request)
            ...
    """
    def decorator(view):
        @functools.wraps(view)
        def sensitive_post_parameters_wrapper(request, *args, **kwargs):
            assert isinstance(request, HttpRequest), (
                "sensitive_post_parameters didn't receive an HttpRequest. "
                "If you are decorating a classmethod, be sure to use "
                "@method_decorator."
            )
            if parameters:
                request.sensitive_post_parameters = parameters
            else:
                request.sensitive_post_parameters = '__ALL__'
            return view(request, *args, **kwargs)
        return sensitive_post_parameters_wrapper
    return decorator
```

```
dispatch = sensitive_post_parameters()(dispatch)
```

1. sensitive*post*paramters() 를 호출하면 decorator 함수가 반환된다.
2. dispatch = decorator(dispatch) 로 동일한 형태임을 할 수 있다.

functools.wraps()를 안쓰게 되면 어떻게 될까?

> A side effect of using decorators is that the function that gets wrapped loses it's natural **name**, **doc**, and **module** attributes. The wraps function is used as a decorator that wraps the function that a decorator returns, restoring those three attributes to the values they would have if the wrapped function was not decorated. For instance: an*expensive*function's name (as seen by an*expensive*function.**name**) would have been 'wrapper' if we did not use the wraps decorator.