Uso de decoradores en python

4

Hola.

¿Usáis decoradores en vuestro código?

En caso afirmativo, ¿cómo los usáis (ejemplo, contexto,...)?

1

En general, no entiendo muy bien su aplicación directa, cuando veo código por ahí no veo mucho uso de decoradores y quería saber cómo lo usáis. La pregunta está hecha desde la ignorancia más absoluta.

Gracias.

Saludos.

decoradores decorators

preguntado **11 May '11, 23:33**

kikocorreoso
1.2k ● 10 ● 20 ● 36
Aceptadas: 31%

Para mi la aplicación mas directa es la generación de logs. Pero por ejemplo también lo he usado para decorar funciones que quiero saber cuanto tardan en ejecutarse.

monobot (10 Jun '13, 10:26)

3 Respuestas:

respuestas más antiguas | respuestas más recientes | respuestas populares

9

Los *decoradores* son bastante recientes y necesitan algo de tiempo para que se incorporen a la programación habitual.

Se han usado decoradores desde hace mucho para crear *métodos de clase* (decorador `methodclass`) y *métodos estáticos* (decorador `staticmethod`), aunque no era siempre tan simple como la sintaxis `@decorador` que se usa hoy en día.

Otro uso que se ve mucho es el decorador para crear **propiedades** (decorador `property`):

```
def MiClase(object):
    def __init__(self,value):
        self._value=value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value=value
```

Más en general, el caso de uso en **django** y otros frameworks web son casos de **Programación Orientada a Aspecto** donde se mantiene constante alguna característica transversal común como sería, por ejemplo, cuando quieres que todas las vistas tengan la misma cabecera y el mismo pié de página.

De modo similar, también se pueden usar generadores para **Diseño por Contrato**. Es la empleada por **zope.interface**, aunque no emplee la sintaxis de `@decorador` por compatibilidad. Un caso de uso frecuente sería la comprobación del *interface* (`secuencia`, `fichero`, etc), o simplemente para comprobar el tipo de argumento de entrada:

decoradores ×3

decorators ×1

pregunta realizada: **11 May '11, 23:33**

pregunta vista: **2,023 veces**

última modificación: **01 Jul '13, 23:31**

Preguntas relacionadas

¿Cual es la manera más eficiente de emplear la función property?

¿Cual es la manera más idonea de medir el rendimiento del codigo?

```python
from functools import wraps

def check_type(typ):
    def check_func(f):
        @wraps(f)
        def f_decorado(elem):
            if not isinstance(elem, typ):
                raise TypeError("%s no es de tipo %s"%(repr(elem),repr(typ)))
            return f(elem)
        return f_decorado
    return check_func

@check_type((int,long))
def mifuncion(seq):
    #...Más código...
```

Otro uso, no muy frecuente, es la *currificación* de funciones, que proviene del lenguaje **Curry** donde se pueden fijar algunos argumento de una función. En python se puede *currificar* funciones con `functools.partial`:

```python
from functools import partial

addtwo=partial(int.__add__,2)
print addtwo(3)
```

No usa la sintáxis `@decorator`, pero sigue siendo un decorador de `int.__add__`

Este caso de *currificación* es el que siempre se pone como ejemplo. Tal vez no se vea claro su utilidad, así que pongo otro ejemplo más ilustrativo:

```python
from functools import partial

display=partial(str.format,HOME="/home/chema",USER="chema")
print display("El usuario {USER} está en {HOME}")

display.keywords["USER"]="pepe"
print display("Ahora eres el usuario {USER}")
```

enlace permanente

respondido **12 May '11, 14:44**

chemacortes
4.0k ●5 ●49 ●57
Aceptadas: 55%

¡Qué bueno el último ejemplo!

Juanlu001 (01 Jul '13, 23:31)

5

Primero la teoría básica, que es sencilla pero hay que entenderla bien: un decorador no es más que una función que recibe una función (a decorar) como parámetro y devuelve otra función, que denominaremos como **decorada**. El decorador puede hacer todo tipo de cosas con la función que recibe, pero lo habitual es ejecutar código antes y/o después de la función original. Se ve más fácil con un ejemplo:

```
1  def decorador(funcion):
2      def funcion_decorada(*args, **kwargs):
3          print "Antes de llamar a la funcion %s" % funcion.__name__
4          funcion(*args, **kwargs)
5          print "Despues de llamar a la funcion %s" % funcion.__name__
6      return funcion_decorada
7
8  @decorador
9  def mi_funcion(arg1, arg2):
10     print arg1, arg2
```

La sintaxis `@decorador` es equivalente a:

```
1  def mi_funcion(arg1, arg2):
2      print arg1, arg2
3  mi_funcion = decorador(mi_funcion)
```

Aquí se ve claramente lo que decía el principio, que un decorador no es más que una función que recibe una función como parámetro (aunque puede recibir más parámetros) y devuelve otra función.

Ahora al llamar a mi_funcion el resultado sería:

```
>>> mi_funcion('La respuesta es', '42')
Antes de llamar a la funcion funcion_decorada
La respuesta es 42
Despues de llamar a la funcion funcion_decorada
```

¡Un momento! ¡¿Cómo que funcion_decorada?! ¡¿Pero no es mi_funcion?! **¡No!** mi_funcion dejó de existir cuando se le aplicó el decorador, ¡se sustituyó por la función decorada! Cuando es importante que el nombre (y otros detalles) de la función decorada coincidan con los de la función original, se usa el decorador de Python `functools.wraps`:

```
from functools import wraps
def decorador(funcion):
    @wraps(funcion)
    def funcion_decorada(*args, **kwargs):
        # ... el resto de código permanece igual
```

Ahora sí se mantiene el nombre original (pero hay que recordar que sigue siendo una función diferente a la definida originalmente):

```
>>> mi_funcion('La respuesta es', '42')
Antes de llamar a la funcion mi_funcion
La respuesta es 42
Despues de llamar a la funcion mi_funcion
```

Sobre usos reales de los decoradores pues hay muchos y muy útiles. Un ejemplo real que me gusta mucho son los decoradores para el control de acceso de Django, como por ejemplo `login_required` o `permission_required`. Para hacer que una vista en Django sólo sea accesible por usuarios registrados es tan simple como decorar esa vista con `login_required`, así:

```
@login_required
def una_vista(request):
    blablablabla
```

Internamente `login_required` comprueba que el usuario haya iniciado sesión en el sistema (mirando en el parámetro `request`), si lo ha hecho continua ejecutando el código de `una_vista()`, y si no redirige al usuario a un formulario de inicio de sesión para que introduzca su nombre de usuario y contraseña. Es un caso de uso muy habitual en el diseño de páginas web, resuelto con una sencilla línea de código.

Otro decorador muy útil, también en Django, aunque el concepto es aplicable a cualquier aplicación que trabaje con base de datos, es `commit_on_success` (y familia). Este decorador hace que todos los accesos a base de datos que se realizan en la función decorada se ejecuten en una misma transacción, de forma que si ocurre algún error en la función (excepción) se realiza un *rollback* de la transacción, y si se termina con éxito se *commitea* automáticamente.

Un último ejemplo, también de Django (qué puedo decir, es lo que mejor conozco) es el decorador `cache_page`, que *cachea* automáticamente el resultado de las vistas, y los reaprovecha automáticamente si se vuelven a solicitar.

enlace permanente editó **12 May '11, 00:44**

respondido **12 May '11, 00:36**

haplo
5.4k ●2 ●50 ●74
Aceptadas: 68%

Muchas gracias. La teoría más o menos la pillo pero lo que no tengo muy claro es donde aplicarlo y que me suponga una gran ventaja (zopenco que es uno). Los ejemplos que ponen ayudan mucho. Gracias.

kikocorreoso (12 May '11, 09:51)

1

Alguien que lo explica muy bien, y por supuesto mejor que yo :) es Juanjo Conti:

- Decoradores 1
- Decoradores 2
- Decoradores 3

Y si quedan dudas, están las charlas y otras cosillas para leer

enlace permanente

respondido **03 Jun '13, 16:23**

matigro
56 ●4

Aceptadas: 0%

Tu respuesta

[ocultar vista previa]

wiki comunitario

Enviar la respuesta (después debes identificarte o registrarte para publicarla)