

Entendiendo Decoradores en Python

- El principio de todo
- ¿Qué es un decorador?
- Funciones decoradoras
- Decoradores con parámetros
- Clases decoradores
- Decorar clases



El principio de todo

Todo en Python es un objeto.

- Identidad
- Tipo
- Valor

Objetos

```
>>> a = 1
>>> id(a)
145217376
>>> a.__add__(2)
3
```

Otros objetos:

```
[1, 2, 3]      # listas
5.2           # flotantes
"hola"        # strings
```

Funciones

Las funciones también son objetos.

```
def saludo():  
    print "hola"
```

```
>>> id(saludo)  
3068236156L  
>>> saludo.__name__  
'saludo'  
>>> dice_hola = saludo  
>>> dice_hola()  
hola
```

Decorador (definición no estricta)

Un decorador es una *función* **d** que recibe como parámetro otra *función* **a** y retorna una nueva *función* **r**.

- d: función decoradora
- a: función a decorar
- r: función decorada

```
a = d(a)
```

Código

```
def d(a):  
    def r(*args, **kwargs):  
        # comportamiento previo a la ejecución de a  
        a(*args, **kwargs)  
        # comportamiento posterior a la ejecución de a  
    return r
```

Código

```
def d(a):  
    def r(*args, **kwargs):  
        print "Inicia ejecucion de", a.__name__  
        a(*args, **kwargs)  
        print "Fin ejecucion de", a.__name__  
    return r
```

Manipulando funciones

```
def suma(a, b):  
    print a + b
```

```
>>> suma(1,2)  
3  
>>> suma2 = d(suma)  
>>> suma2(1,2)  
Inicia ejecucion de suma  
3  
Fin ejecucion de suma  
>>> suma = d(suma)  
>>> suma(1, 2)  
Inicia ejecucion de suma  
3  
Fin ejecucion de suma
```

Azuca sintáctica

A partir de Python 2.4 se incorporó la notación con @ para decorar funciones.

```
def suma(a, b):  
    return a + b
```

```
suma = d(suma)
```

```
@d  
def suma(a, b):  
    return a + b
```

Atención

El decorador malvado.

```
def malvado(f):  
    return False
```

```
>>> @malvado  
... def algo():  
...     return 42  
...  
>>> algo  
False  
>>> algo()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'bool' object is not callable
```

Decoradores en cadenas

Similar al concepto matemático de componer funciones.

```
@registrar_uso
@medir_tiempo_ejecucion
def mi_funcion(algunos, argumentos):
    # cuerpo de la funcion
```

```
def mi_funcion(algunos, argumentos):
    # cuerpo de la funcion

mi_funcion = registrar_uso(medir_tiempo_ejecucion(mi_funcion))
```

Decoradores con parámetros

- Permiten tener decoradores más flexibles.
- Ejemplo: un decorador que fuerce el tipo de retorno de una función

```
@to_string  
def count():  
    return 42
```

```
>>> count()  
'42'
```

Decoradores con parámetros

Primera aproximación.

```
def to_string(f):  
    def inner(*args, **kwargs):  
        return str(f(*args, **kwargs))  
    return inner
```

Decoradores con parámetros

Algo más genérico?

```
@typer(str)
def c():
    return 42

@typer(int)
def edad():
    return 25.5
```

```
>>> edad()
25
```

Decoradores con parámetros

`typer` es una fábrica de decoradores.

```
def typer(t):  
    def _typer(f):  
        def inner(*args, **kwargs):  
            r = f(*args, **kwargs)  
            return t(r)  
        return inner  
    return _typer
```

Clases decoradoras

- Decoradores con estado.
- Código mejor organizado.

```
class Decorador(object):  
  
    def __init__(self, a):  
        self.variable = None  
        self.a = a  
  
    def __call__(self, *args, **kwargs):  
        # comportamiento previo a la ejecución de a  
        self.a(*args, **kwargs)  
        # comportamiento posterior a la ejecución de a
```

Clases decoradoras

```
@Decorator
def nueva_funcion(algunos, parametros):
    ...
```

- Se instancia un objeto del tipo Decorador con nueva_función como argumento.
- Cuando llamamos a nueva_funcion se ejecuta el método `__call__` del objeto instanciado.

```
def nueva_funcion(algunos, parametros):
    ...
nueva_funcion = Decorador(nueva_funcion)
```

Decoradores (definición más estricta)

Un decorador es una *callable* **d** que recibe como parámetro un *objeto* **a** y retorna un nuevo objeto **r** (por lo general del mismo tipo que el original o con su misma interfaz).

- d: clase que defina el método `__call__`
- a: cualquier objeto
- r: objeto decorado

```
a = d(a)
```

Decorar clases (Python >= 2.6)

Identidad:

```
def identidad(C):  
    return C
```

```
>>> @identidad  
... class A(object):  
...     pass  
...  
>>> A()  
<__main__.A object at 0xb7d0db2c>
```

Decorar clases (Python >= 2.6)

Cambiar totalmente una clase:

```
def abuse(C):  
    return "hola"
```

```
>>> @abuse  
... class A(object):  
...     pass  
...  
>>> A()  
Traceback (most recent call last):  
  File "", line 1, in  
TypeError: 'str' object is not callable  
>>> A  
'hola'
```

Decorar clases (Python >= 2.6)

Reemplazar con una nueva clase:

```
def reemplazar_con_X(C):  
    class X():  
        pass  
    return X
```

```
>>> @reemplazar_con_X  
... class MiClase():  
...     pass  
...  
>>> MiClase  
<class __main__.X at 0xb78d7cbc>
```

Decorar clases (Python >= 2.6)

Instancia:

```
def instanciar(C):  
    return C()
```

```
>>> @instanciar  
... class MiClase():  
...     pass  
...  
>>> MiClase  
<__main__.MiClase instance at 0xb7d0db2c>
```

Dónde encontramos decoradores?

Permisos en Django

```
@login_required
def my_view(request):
    ...
```

URL routing en Bottle

```
@route('/')
def index():
    return 'Hello World!'
```

Standard library

```
classmethod, staticmethod, property
```

Muchas gracias!

- Comentarios, dudas, sugerencias: jjconti@gmail.com
- <http://www.juanjoconti.com.ar>

- <http://www.juanjoconti.com.ar/categoria/aprendiendo-python/>
- <http://www.juanjoconti.com.ar/2008/07/11/decoradores-en-python-i/>
- <http://www.juanjoconti.com.ar/2009/07/16/decoradores-en-python-ii/>
- <http://www.juanjoconti.com.ar/2009/12/30/decoradores-en-python-iii/>