Entendiendo Decoradores en Python

La siguiente es la *Agenda* o los tópicos que traté en la charla:

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

Por supuesto, ya que vamos a hablar de decoradores, con una foto alusiva.

El principio de todo

	Todo en Python es un objeto • Identidad • Tipo • Valor

Todo en Python es un objeto, todo. Los números, strigs, listas, tuplas y otras cosas más raras: los módulos son objetos, el código fuente es un objeto, todo es un objeto. TODO.

Cada objeto tiene 3 características o atributos: identidad, tipo y valor.

Objetos

Veamos algunos ejemplos. El número 1 es un objeto. Usando la función built-in `id` podemos averigurar su identidad. Su tipo es `int` y su valor es obviamente 1.

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

Al ser un objeto, podemos aplicarle algunos de sus métodos. `__add__` es el método que se llama cuando utilizamos el símbolo +.

Otros objetos:

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

Funciones

Si todo son objetos, las funciones también son objetos.

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

Podemos obtener el id de una función mediante `id`, acceder a sus atributos o incluso hace que otro nombre apunte al mismo objeto función:

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

Decorador (definición no estricta)

Vamos a tomarnos por un momento una libertad y diremos que 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

Podemos aplicar el decorador utilizando una notación funcional:

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

Veamos ahora cómo implementamos un decorador genérico:

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
```

Definimos una función **d**, nuestro decorador, y en su cuerpo se define una nueva función **r**, aquella que vamos a retornar. En el cuerpo de **r** ejecuta **a**, la función decorada.

Cambiemos ahora los comentarios por código que haga algo:

Código

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

Cuando ejecutemos una función decorada con el decorador anterior, se mostrará un poco de texto, luego se ejecuta la función decorada y se finaliza con un poco más de texto. Veamos un ejemplo.

En `suma2` nos guardamos la versión decorada de `suma`. Veamos ahora lo que pasa cuando la ejecutamos:

Manipulando funciones

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

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

Así mismo podemos guardarnos directamente en `suma` la versión decorada de `suman` y ahora nunca más a lo largo del programa se tendrá acceso a la versión original.

La anterior forma de aplicar un decorador es la forma *funcional*. Tenemos una más linda:

Azúcar sintáctica

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

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

suma = d(suma)
```

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

En la porción de código anterior se pueden ver dos ejemplos en donde comparamos las formas de aplicar un decorador.

Lo siguiente es ver ejemplos de decoradores *reales*:

Atención

Antiejemplo: 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
```

Este decorador es tramposo, por que en lugar de devolvernos una nueva función, devuelve algo nada que ver, una objeto booleano. Obviamente, cuando intentamos ejecutar, obtenemos un error.

Siguiente ejemplo:

Decoradores en cadenados

Similar al concepto matemático de componer funciones.

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

Es equivalente a:

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

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

Vamos adentrándonos un poco más en los laberintos oscuros Decoradores los decoradores:

Decoradores con parámetros

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

Supongamos que queremos un decorador que convierta a sitring todas las respuestas de una función. Se usaría de esta forma:

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

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

¿Cómo lo implementarías? Veamos una primera aproximación:

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

Esta forma anda, pero pensemos si podemos hacerlo de una forma más genérica. La siguiente es la forma de utilizar el decorador `typer`:

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

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

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

En realidad, `typer` no es un decorador, 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
```

Notemos que `_typer` es el verdadero decorador, la función externa recibe un parámetro `t` que es utilizado para definir la naturaleza del decorador a crear.

Ahora nos vamos un poco más lejos y veremos:

Clases decoradoras

- Decoradores con estado.
- Código mejor organizado.

El primer es un ejemplo similar a nuestra primera función decoradora, un ejemplo genérico:

```
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
```

La siguiente es la forma de usarlo:

```
@Decorador
def nueva_funcion(algunos, parametros):
    # cuerpo de la funcion
```

- 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.

También podemos aplicarlo, utilizando la vieja notación de llamada de funciones:

```
def nueva_funcion(algunos, parametros):
    # cuerpo de la funcion
nueva_funcion = Decorador(nueva_funcion)
```

Con estos ejemplos vistos, podemos hacer una definición más estricta de decoradores:

Decorador (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 orginal o con su misma interfaz).

- d: objeto de un tipo que defina el método `__call__`
- a: cualquier objeto
- r: objeto decorado

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

Decorar clases (Python >= 2.6)

A partir de Python 2.6, se permite el uso de la notación con @ antes de la definición de una clase. Esto da lugar al concepto de decoradores de clases. Si bien antes de 2.6 se podía decorar una clase (utilizando la notación funcional), recién con la introducción de este azúcar sintácticase empezó a hablar más de decoradores de claseds. Un primer ejemplo:

Identidad:

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

Retorna la misma clase que estamos decorando.

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

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'
```

Similar a uno de los ejemplos del princpio, el ejemplo nos muestra que lo que retorne un decorador tiene que tener una interfaz Similar a la del objeto que estamos decorando, así tiene sentido cambiar el uso de la versión original del objeto, por una cambiada.

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>
```

En el caso anterior vemos que la clase bue cambiada complemtamente por una clase totalmente diferente.

Instancia:

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

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

Como último ejemplo de decoradores de clase vemos un decorador que una vez aplicado, instancia la clase y asocia este objeto a su nombre. Puede verse como una forma de implementar el patrón Singleton, estudiado en programación.

Para terminar:

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!

Datos y contacto

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

- 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/
- http://www.juanjoconti.com.ar/2010/08/07/functools-update_wrapper/