Decoradores en Python (/weblog/2009/08/31/decoradores-en-python/)

Escrito por admin (/weblog/authors/admin/) on 31 de Agosto de 2009 en Django (/weblog/categories/django/), Python (/weblog/categories/python/).

¿Qué es un decorador?

Un decorador es el nombre de un *patrón de diseño*. Los decoradores alteran de manera dinámica la funcionalidad de una función, método o clase sin tener que hacer subclases o cambiar el código fuente de la clase decorada. En el sentido de Python un decorador es algo más, incluye el patrón de diseño, pero va más allá, Bruce Eckel los asimila a las macros de Lisp. Los decoradores y su utilización en nuestros programas nos ayudan a hacer nuestro código más limpio, a autodocumentarlo y, a diferencia otros lenguajes, no requieren que nos aprendamos otro lenguaje de programación distingo (cómo pasa con las anotaciones de Java por ejemplo). En su utilización podemos simular la programación orientada a aspectos (AOP) o utilizarlos para añadir sistemas de control a nuestras funciones, de log, caché, ... Las posibilidades son infinitas. Los decoradores forman parte de Python desde la versión 2.4 y cómo dice Michele Simionato (http://http//www.phyast.pitt.edu/micheles/) nos aportan lo siguiente:

- Reducen el código común y repetitivo (el llamado código *boilerplate*).
- Favorecen la separación de responsabilidades del código
- Aumentan la legibilidad y la mantenibilidad
- Los decoradores son explícitos.

Clasificación de los decoradores

Podemos dividir los decoradores en grupos:

- Según los parámetros que admiten:

    - No admiten parámetros
    - Sí admiten parámetros
- Según si preservan la firma o signatura del método al que decoran:
    - Decoradores no que preservan la firma
    - Decoradores que sí la preservan

Los decoradores más sencillos son aquellos que no admiten parámetros y no preservan la firma

Un decorador que no hace nada

Para empezar crearemos un decorador que el que hará se convertir cualquier función en un /dev/null, es decir, no devolverá nada y no hará nada con la función. Llamaremos a nuestro decorador **forat_negre**, agujero negro.

```
1  def forat_negre(f):
2      def none():
3          pass
4      return none
5  @forat_negre
6  def di_hola():
7      return "hola"
```

Si ejecutamos *di_hola()* no tendremos resultado alguno, mejor dicho, tendremos `None` como resultado de la función que estamos decorando. La sintaxis @ del decorador de Python es lo que se denomina *syntactic sugar*, es decir, una manera de escribir las cosas que aumenta la legibilitat del código. Sin embargo, debemos tener presente que el decorador se podría haber escrito como:

```
1  di_hola = forat_negre(di_hola)
2  di_hola()
```

y tendríamos el mismo decorador. Recordemos que las funciones son objetos de primera clase en PYthon y que se pueden asignar y pasar como parámetros. Aunque el ejemplo es muy sencillo nos sirve para ver lo siguiente: **Un decorador no es más que un envoltorio de una función** y por lo tanto tiene que devolver una función, más concretamente un _callable_, para entendernos, cualquier cosa que poniendo un doble paréntesis al lado () no pete.

```
1  def retorna_objecte(f):
2      ....:     def obj():
3      ....:         return object()
```

```
    ....:     return obj
    ....:

In [17]: def di_hola():
    ....:     return "Hola"
    ....:

In [18]: di_hola = retorna_objecte(di_hola)

In [19]: di_hola()
Out[19]: <object object at 0xf7f745e8>
```

A nuestro decorardor `forat_negre` le hemos pasado una función sin parámetros. Si intentamos pasarle una función con parámetros nos encontraremos con una pequeña sorpresa...

```
@forat_negre
def suma(a,b):
    return a,b

suma(2,3)

TypeError Traceback (most recent call last)
TypeError: none() takes no arguments (2 given)
```

que por otro lado es del todo normal, hemos definido el `agujero_negro` de tal manera que devuelve una función sin parámetros, así que si le intentamos pasar los parámetros que tenía la función decorada sencillamente se queja y peta. Vamos a definir un poco mejor nuestro decorador para que esto no nos pase y que pueda admitir al menos tantos parámetros como la función que decora.

```
def forat_negre(f):
    "d'aquí no surt res"
    def none(*args, **kw_args):
        pass
    return none

@forat_negre
def suma(a,b):
    "suma dos parametres qualsevols si pot"
    return a+b
```

```
11
12 suma(2,2)
```

Ahora ya no da error. Así pues tenemos *otra conclusión*: además de devolver una función, tenemos que procurar que la definición de las función que devolvemos admita al menos el mismo número de parámetros que la función que queremos decorar. Si no sabemos cuántos son estos nos curamos en salud con *args* y *kw_args*. Fijémonos que no hemos mantenido la firma de la función. Si como experimento intentáis hacer un `help(suma)` no obtenemos la ayuda de la función original. Volveremos sobre esto un poco más adelante. Por ahora ya sabemos como crear decoradores simples a partir de una función.

Haciendo decoradores no intrusivos

Si habéis hecho un `help(suma)` o un `suma.__name__` quizás alguno se habrá sorprendido al ver que el nombre de la función es `_none_` en lugar de la esperada `suma`. Si repasáis lo que hemos hecho tampoco es de extrañar: hemos sustituido la función original por otra. Recordamos que el decorador f aplicado sobre la función g es equivalente a hacer g = f(g). Lo aconsejable sería que el decorador fuera capaz de mantener la documentación y el nombre de la función que decora, ya que de este modo se simplificaría el uso de la función y los autocompletadores de código no se vuelverían tan locos. Esto lo podemos hacer de dos maneras: la larga y la corta

La manera larga

```
1 def forat_negre(f):
2     def none(*args, **kw_args):
3         pass
4     none.__doc__= f.__doc__
5     none.__dict__= f.__dict__
6     none.__name__= f.__name__
7     return none
```

Con las tres instrucciones adicionales que hemos puesto volvemos a recuperar los metadatos de la función original que pasamos al decorador. Si ahora hacemos un `help` veremos que èste devuelve el nombre de la función correcta __suma__ y que la ayuda también es la suya.

```
1 Help on function suma in module __main__:
2 suma(*args, **kw_args)     Suma dos parametres qualsevols si pot
```

La manera corta

Preservar los metadatos es bastante útil y común, de hecho en el módulo `functools` encontramos la función `wraps` que es en sí misma un decorador y que hace precisamente ésto. De este modo el código anterior quedaría:

```
1  from functools import wraps
2
3  def forat_negre(f):
4      @wraps(f)
5      def none(*args, **kw_args):
6          pass
7      return none
```

Observemos que hemos utilizado un decorador para crear otro decorador. Veremos más utilización de decoradores un poco más adelante.

Un decorador con argumentos

El decorador que hemos programado en el apartado anterior era bastante simple, hacía muy poca cosa y no tenía parámetros. Si queremos crear decoradores tenemos que hacer primero de todo que sean útiles, y también nos encontraremos con la necesidad de que estos decoradores admitan parámetros. En Django, por ejemplo, podéis encontrar que el decorador de cache (/admin/zinnia/entry/2/http:/docs.djangoproject.com/en/dev/topics/cache/) admite parámetros que nos permiten decirle durante cuánto tiempo tiene que cachear los resultados, o el decorador `vary_on_headers`, que nos permite modificar el contenido de la respuesta de las vistas añadiendo las cabeceras que indicamos. Vamos a ver como lo podemos conseguir nosotros. También hay dos maneras de hacerlo, la clara y la compleja. La manera clara es la que recomendamos y utiliza una clase para hacer el decorador, la compleja requiere más esfuerza para entender qué está haciendo el decorador, es más corta, pero personalmente prefiero un código más legible. Los decoradores que hemos programado como funciones se pueden crear también como clases, pero en este caso, creo que la definición en forma de funciones es más sencilla de seguir, y nos permitirá distinguir claramente entre los dos tipos de decoradores: los que no admiten parámetros que se construyen preferentemente mediante funciones y los que admiten parámetros, que se construyen preferentemente usando clases. Para seguir con el agujero negro, ahora en nuestro ejemplo haremos que se muestre el resultado de la funcion que decoramos o no de manera aleatoria. Pora ello pasaremos al decorador una función como parámetro que al ser ejecutada determinará si se tiene que mostrar el resultado de la función decorada o no

El método claro de hacer decoradores con argumentos

```
1  #!/usr/bin/env python
2  # -*- coding: UTF-8 -*-
```

```
import random

class forat_negre_sonat(object):
    "Un decorador amb fam"
    def __init__(self, mostrar):
        self.mostrar = mostrar

    def __call__(self, f):
        def none(*args, **kw_args):
            if self.mostrar():
                return f(*args, **kw_args)
            else:
                return "Nop"
        return none

@forat_negre_sonat(mostrar = lambda :random.choice((True, False)))
def suma(a, b):
    "Suma dos elements que li passam com a paràmetre"
    return a+b

if __name__=="__main__":
    print suma(2,3)
    print suma(5,6)
    print suma(9,5)
```

Observemos el código: 1. Hemos creado una clase Python que a su constructor (el `__init__`) puede tomar el parámetro o parámetros que queramos. Es un constructor normal, así que admite parámetros por defecto. 2. Recordemos que hemos dicho que el decorador tiene que ser un objeto llamable (callable), en una clase, la _llamabilidad_ la da el método `__call__`. Esta clase la definiremos de forma que obtenga la función a decorar com un parámetro. De este modo tenemos acceso tanto a los parámetros del decorador, que hemos pasado al constructor, como a la función decorada, que hemos pasado como parámetro al call. Después de esto ya sólo queda encapsular la llamada como lo hacíamos al caso anterior, devolviendo el decorador en lugar de la función a decorar. En el ejemplo además, he intentado mostrar que el parámetro puede ser el que nosotros queramos, en concreto he pasado una función anónima, creada con `lambda` que es la que se encarga de establecer la aleatoriedad del resultado. Si queréis podemos hacer este decorador un poco más completo, haciendo que admita además de funciones valores y que preserve el nombre y documentación de la función decorada.

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-
import random

```

```
class forat_negre_sonat(object):
    "Un decorador amb fam"
    def __init__(self, mostrar=None):
        self.mostrar = mostrar

    def __call__(self, f):
        def none(*args, **kw_args):
            if callable(self.mostrar):
                opcion = self.mostrar()
            else:
                opcion = self.mostrar
            if opcion:
                return f(*args, **kw_args)
            else:
                return "Nop"
        none.__name__ = f.__name__
        none.__doc__ = f.__doc__
        return none

@forat_negre_sonat(mostrar = lambda :random.choice((True, False)))
def suma(a, b):
    "Suma dos elements que li passam com a paràmetre"
    return a+b

@forat_negre_sonat(mostrar=True)
def resta(a,b):
    return a-b

if __name__=="__main__":
    print "Exemple amb %s " % suma.__name__
    print suma(2,3)
    print suma(5,6)
    print suma(9,5)
    print "Exemple amb %s " % resta.__name__
    print resta(2,3)
    print resta(5,6)
```

El método complejo de crear decoradores con parámetros

```
def forat_negre_dos(mostrar):
```

```
    def wrap(f):
        @wraps(f)
        def wrapped_function(*args, **kw_args):
            if callable(mostrar):
                opcion = mostrar()
            else:
                opcion = mostrar
            if opcion:
                return f(*args, **kw_args)
            else:
                return "Nop"
        return wrapped_function
    return wrap
```

Bien, enrevesado, lo que se dice enrevesado no lo es, ya que una cosa tan simple no tiene demasiada historia, pero fijaos que el codigo se sigue bastante peor. El primero que hemos hecho es definir nuestra función, donde hemos puesto los parámetros que admite. Esta función devuelve otra función que admite un argumento, que es la función a decorar, que a su vez admite un número indeterminado de argumentos (recordemos que esto lo estamos forzando nosotros). Cómo la segunda función, `wrapped_function` está definida dentro de `wrap`, tiene acceso al parámetro del decorador y puede utilizarlo.

Encadenando decoradores

Los decoradores se pueden encadenar, es decir, una función puede tener tantos decoradores como haga falta y necesitemos, sólo limitados por nuestro sentido común y la legibilidad del programa. Dos decoradores son habituales, tres no se ven mucho, cuatro o más son para pensárselo. Para el ejemplo tomaré prestado Python Decorator Library uno de los decoradors más útiles, el memoize, que nos permite cachear una función a partir de sus parámetros. La implementación que hace Python Decorator Library del patrón memoize bastante es sencilla de seguir con lo que ahora sabemos y además nos servirá para completar la construcción de decoradores sin parámetros usando una clase.

```
class memoized(object):
   """Decorator that caches a function's return value each time it is called.
   If called later with the same arguments, the cached value is returned, and
   not re-evaluated.
   """
   def __init__(self, func):
      self.func = func
      self.cache = {}
   def __call__(self, *args):
```

```
        try:
            return self.cache[args]
        except KeyError:
            self.cache[args] = value = self.func(*args)
            return value
        except TypeError:
            # uncachable -- for instance, passing a list as an argument.
            # Better to not cache than to blow up entirely.
            return self.func(*args)
    def __repr__(self):
        """Return the function's docstring."""
        return self.func.__doc__
```

A diferencia de la construcción con parámetros, en el constructor de la clase memoized se pone como parámetro la función a decorar, y al método __call__ los parámetros de la función, en lugar de la función a decorar como se hacía al otro método. ¿Por qué se ha usado esta manera si la otra es más sencilla? Pues porqué necesitamos mantener en memoria la caché y lo que se hace es mantenerla en un diccionario dentro de la misma clase. Si la caché fuera externa (con memcached por ejemplo), se habría podido hacer perfectamente en forma de función. Además definiremos un decorador que nos servirar para indicar cuando entramos a la función y comprobar el decorador memoized.

```
def log(f):
    "Registra l'execució de la funció"
    def wrap(*args):
        print "Excutant %s, args: %s" % \\
            (f.__name__, ",".join(str(x) for x in args))
        return f(*args)
    return wrap

@memoized
@log
def fibonacci(n):
    "Return the nth fibonacci number."
    if n in (0, 1):
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print fibonacci(12)
```

Probad de ejecutar este código con y sin la función memoized. Con los dos decoradors activos veréis que cada decorador toma como entrada la función ya decorada que sale del decorador que tiene más abajo. Así el memoized coge como entrada la función fibonacci ya decorada con el log. Podéis hacer la prueba con un ejemplo más simple:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

def uppercase(f):
    "Dada una función f que devuelve un string lo pasa todo a mayúsculas"
    def wrap():
        return f().upper()
    return wrap

def make_bold(f):
    "Dada una función f que devuelve un string le añade los tags de bold"
    def wrap():
        return "<strong>%s</strong>" % f()
    return wrap

@make_bold
@uppercase
def say_hello():
    return "Hello world"

print say_hello()
```

Probad cambiando la orden de los decoradores y veréis perfectamente como se van aplicando éstos desde la función hacia arriba. En el ejemplo primero se convierte el "Hello word" a mayúsculas y después se le aplican los tags de negrita.

La signatura pendiente

>Antes de acabar nos queda un tema pendiente: la firma. Los decoradors que hemos creado pueden preservar el nombre y la documentación de la función que decoran, pero no preservan la firma, es decir, el número de parámetros que le pasamos. Michele Simionato ha escrito un módulo excelente llamado *decorator* que extiende la utilizació de los decoradores, mateniendo la firma de la función, el nombre y la documentación, y además nos da la posibilidad de crear factorías de decoradors. Una herramienta para tener siempre a mano. Con este módulo podríamos escribir el código del ejemplo anterior cómo:

```
from decorator import decorator
```

```
 2
 3  @decorator
 4  def uppercase(f, *args):
 5      "Donada una funció f que retorna un string ho passa a majúscules"
 6      return f(*args).upper()
 7
 8  @decorator
 9  def make_bold(f, *args):
10      "Afegeix el tag strong a la sortida de la funció"
11      return "<strong>%s</strong>" % f(*args)
12
13  @uppercase
14  @make_bold
15  def say_hello(nom):
16      "Di hola, home!"
17      return "Hello world %s" % nom
18
19  if __name__=="__main__":
20      from inspect import getargspec
21      print say_hello('World')
22      print say_hello.func_name
23      print say_hello.__doc__
24      print getargspec(say_hello)
```

Si ejecutáis el código podréis ver que no nos ha hecho falta recurrir a wraps o a reasignar el nombre, la propia librería de Simionato lo ha hecho. Además, si nos fijamos en la salida del ejemplo:

> **HELLO WORLD WORLD** say_hello Di hola, home! ArgSpec(args=['nom'], varargs=None, keywords=None, defaults=None)

La primera línea corresponde a la salida de la función que hemos decorado. La segunda es el nombre de dicha función. Vemos el nombre de la función original y no la del decorador. La documentación también se ha mantenido y para acabar, podemos ver que la firma de la función es correcta, nos dice que tiene un argumento obligatorio llamado `nom`.

Conclusión

Espero haber dejado un poco más claro el tema de los decoradores. Crearlos no es difícil, utilizarlos es todavía más simple, sólo tenemos que tener claro qué son y cuando usarlos. Son una herramienta potente que nos permite hacer nuestro código más legible y cohesionado. Sin miedo y a disfrutar con los decoradores!. Cómo todo en esta vida, usadlos con sentido común y moderación.

Referencias

Para escribir este artículo me he basado en distintas fuentes, las más interesantes las cito a continuación: PEP 318 (http://www.python.org/dev/peps/pep-0318/) Decorators I : Introduction to Python Decorators (http://www.artima.com/weblogs/viewpost.jsp?thread=240808) Decorators II: Decorator Arguments (http://www.artima.com/weblogs/viewpost.jsp?thread=240845) Python Decorators (http://wiki.python.org/moin/PythonDecorators) Understanding decorators (http://uswaretech.com/blog/2009/06/understanding-decorators/)Charming Python: Decorators make magic easy (http://www.ibm.com/developerworks/linux/library/l-cpdecor.html) Decorator 3.1.2 (http://pypi.python.org/pypi/decorator/3.1.2) Decorator Pattern (http://en.wikipedia.org/wiki/Decorator_pattern) Python decorator Library (http://wiki.python.org/moin/PythonDecoratorLibrary)

Tags : decorator (/weblog/tags/decorator/) python (/weblog/tags/python/)

Url corta : http://blog.apsl.net/weblog/2/ (http://blog.apsl.net/weblog/2/)

Debates : 3 commentarios (/weblog/2009/08/31/decoradores-en-python/#comments) , 1929 trackbacks (/weblog/2009/08/31/decoradores-en-python/#trackbacks)

Entradas similares

Ley de cookies con Django (/weblog/2014/02/13/ley-de-cookies-con-django/)

Class Based Views - Formularios (/weblog/2012/02/29/class-based-views-formularios/)

Trucos del administrador de Django (/weblog/2009/09/01/trucos-del-administrador-de-django/)

Django Class Based Views - Introducción (/weblog/2012/02/26/django-class-based-views-introduccion/)

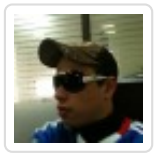
Generando correos (/weblog/2013/12/03/generando-correos/)

Trucos del administrador de Django → (/weblog/2009/09/01/trucos-del-administrador-de-django/)

Comentarios

juan on 01/09/2009 06:11 (2009-09-01T06:11:22) #

Muchas gracias interesante articulo muy bueno, he aprendido algo sobre decoradores



Miguel on 26/09/2009 01:48 (2009-09-26T01:48:20) #

Gracias, Excelente articulo

Decoradores de Fiestas (http://www.fiesta911.com.mx/) on 11/12/2011 01:58 (2011-12-11T01:58:13.395937) #

Excelente post, muy completo y bien explicado, solo hay que practicarlo un poco y como bien dices, no excederse. Saludos!

Los pingbacks están cerrados.

Trackbacks

Categorías

APSL **15 entradas** (/weblog/categories/apsl/)

Django **22 entradas** (/weblog/categories/django/)

FAQ **2 entradas** (/weblog/categories/faq/)

IT Turismo **3 entradas** (/weblog/categories/it-turismo/)

Python **17 entradas** (/weblog/categories/python/)

vim **1 entrada** (/weblog/categories/vim/)

Autores

Pau Rul·lan **1 entrada** (/weblog/authors/pau/)

Antoni Aloy **16 entradas** (/weblog/authors/aaloy/)

admin **22 entradas** (/weblog/authors/admin/)

Daniel Sastre **1 entrada** (/weblog/authors/dsastre/)

Entradas recientes

Nos hemos mudado (2) (/weblog/2014/02/22/nos-hemos-mudado-2/)

Ley de cookies con Django (/weblog/2014/02/13/ley-de-cookies-con-django/)

L'ecosistema de Python per fer coses eclèctiques (/weblog/2014/01/31/lecosistema-de-python-fer-coses-eclectiques/)

Charla en Palma Activa (/weblog/2013/12/11/charla-en-palma-activa/)

Generando correos (/weblog/2013/12/03/generando-correos/)

Comentarios recientes

Pau Rullan Ferragut en Nos mudamos! (/weblog/2012/11/25/nos-mudamos/#comment-1474-by-pau-rullan-ferragut)

David Martín en Nos mudamos! (/weblog/2012/11/25/nos-mudamos/#comment-1473-by-david-martin)

Paco en Nos mudamos! (/weblog/2012/11/25/nos-mudamos/#comment-1472-by-paco)

Tona en Nos mudamos! (/weblog/2012/11/25/nos-mudamos/#comment-1471-by-tona)

aaloy en Vim como editor para Python y Django (/weblog/2012/11/07/vim-como-editor-para-python-y-django/#comment-1469-by-aaloy)

Linkbacks recientes

No hay linkbacks.

Entradas al azar

Generando correos (/weblog/2013/12/03/generando-correos/)

Nuevos proyectos en producción (/weblog/2013/03/01/nuevos-proyectos-en-produccion/)

Introducción a Celery (/weblog/2011/01/14/introduccion-a-celery/)

El 14 inauguramos oficina (/weblog/2012/12/12/el-14-inauguramos-oficina/)

Django Class Based Views - Mostrar un objeto (/weblog/2012/03/01/django-class-based-views-mostrar-un-objeto/)

Entradas populares

PHP o Python **34** (/weblog/2010/09/23/php-o-python/)

¿Va a desaparecer Python? **16** (/weblog/2010/07/29/va-a-desaparecer-python/)

Nos mudamos! **4** (/weblog/2012/11/25/nos-mudamos/)

Introducción a Celery **4** (/weblog/2011/01/14/introduccion-a-celery/)

Django class based views - Epílogo **3** (/weblog/2012/03/03/django-class-based-views-epilogo/)