En borrador permanente
el blog de Juanjo Conti – abstracto, lúdico y digital

Decoradores en Python (III) – Clases decoradoras

Publicado el 30 de diciembre, 2009 por Juanjo

Siguiendo con la serie de posts sobre decoradores en Python, y fiel al espíritu que los originó (ir mostrando lo que voy aprendido a medida que necesito resolver problemas específicos o descubro aplicaciones concretas) hoy les traigo un nuevo uso para los decoradores en Python: **funciones caché**.

Anteriormente: Decoradores I, Decoradores II.

Funciones caché

Una función caché[0], es aquella que siempre que se le pide que compute un resultado para un grupo de parámetros dado, primero se fija en una memoria interna si no realizó ya el cálculo. Si ya lo hizo, retorna el valor computado anteriormente. Si aún no lo hizo, computa el valor, lo guarda en una memoria interna y luego lo retorna.
Esta técnica es muy útil en funciones que requieren un cómputo intensivo y obtener un resultado lleva mucho tiempo. Permita acelerar sustancialmente los tiempos de ejecución a cambio de utilizar más memoria.

La siguiente es una forma de implementarlo en Python para un computo en particular:

```
cache = {}
```

```
def fmem(arg):
    if arg in cache:
        print "Recuperando valor de la memoria"
        return cache[arg]
    else:
        r = (arg ** 10) * (arg ** -5)
        cache[arg] = r
        return r
```

Como memoria se utiliza un diccionario y el argumento de la función fmem es la clave del diccionario[1].

Este es el resultado de utilizarla en el intérprete interactivo:

```
>>> fmem(1)
1.0
>>> fmem(2)
32.0
>>> fmem(2)
Recuperando valor de la memoria
32.0
```

Decoradores con estado

En esta implementación, la técnica de memorización se mezcla con el cálculo que era el objetivo original de la función. Si queremos aplicar la técnica sobre distintas funciones vamos a tener que entrometer la implementación de la caché en todas las funciones. Peor aún, si en el futuro se quiere realizar un cambio en la forma de almacenar y recuperar los valores almacenados, ¡tendríamos que modificar todas las funciones! La forma de resolver estos problemas es implementando un decorador que agregue esta funcionalidad a las funciones decoradas: resolvemos ambos problemas, el de intrución y el de mantenibilidad. Todo el código que provee esta funcionalidad extra es encapsulado en el decorador.

Las funciones decoradoras, como las que vimos en los anteriores artículos, no nos sirven para esta tarea.

Necesitamos un decorador que pueda almacenar un estado. Ya que cualquier *callable* puede ser un decorador, **implementaremos el decorador mediante una clase**.

Funciones caché con clases decoradoras

La definición de la clase decoradora consiste en dos métodos:

- un método de inicialización, dónde se inicializa el atributo cache con un diccionario vacío y se guarda una referencia a la función decorada.
- un método __call__ que será ejecutado cuando se llame a la función decorada.

```
class mem(object):

    def __init__(self, g):
        self.cache = {}
        self.g = g

    def __call__(self, arg):
        if arg in self.cache:
            print "Recuperando valor de la memoria"
            return self.cache[arg]
        else:
            r = self.g(arg)
            self.cache[arg] = r
            return r
```

Luego, lo único que resta es decorar todas las funciones que querramos “dotar de memoria” para obtener mejoras de performance en su ejecución:

```
@mem
def f(arg):
    return (arg ** 10) * (arg ** -5)
```

La salida obtenida al ejecutar la función decorada en el intérprete interactivo es la misma qué en el ejemplo anterior:

```
>>> fmem(1)
1.0
>>> fmem(2)
32.0
>>> fmem(2)
Recuperando valor de la memoria
32.0
```

Más

La implementación del decorador mem solo sirve para decorar funciones que reciben un único argumento. Podemos mejorar su definición para que pueda decorar funciones con cualquier número de argumentos:

```
class mem2(object):

    def __init__(self, g):
        self.cache = {}
        self.g = g

    def __call__(self, *args):
        if args in self.cache:
            print "Recuperando valor de la memoria"
            return self.cache[args]
        else:
            r = self.g(*args)
            self.cache[args] = r
            return r
```

```
@mem2
def f2(arg1, arg2):
    return (arg1 ** 10) * (arg2 ** -5)
```

Notas

[0] Se puede leer más sobre este concepto en Caching Function Results:Faster Arithmetic by Avoiding Unnecessary Computation de Stephen E. Richardson [SMLI TR-92-1]
[1] Esta implementación tiene la limitación de que si el argumento de la función es un objeto mutable, no podrá ser usado como clave de un diccionario y se lanzará una excepción.

Acerca de Juanjo
Mi nombre es Juanjo Conti, vivo en Santa Fe y soy Ingeniero en Sistemas de Información. Mi lenguaje de programación de cabecera es Python; lo uso para trabajar, estudiar y jugar. Como hobby escribí algunos libros.
Ver todas las entradas por Juanjo →

Esta entrada fue publicada en Aprendiendo Python y etiquetada decoradores, Python. Guarda el enlace permanente.

J. Pablo Fernández • hace 4 años
Algo que podes hacer para que funcione con mas de un argumento es convertir *args a una tupla y usar eso como key en el diccionario.

Ahora, mejor simplemente usa el decorador memoize, que viene con Python y ya :)

jjconti Moderador • hace 4 años
Juan Pablo:

1) eso es justamente lo que hago! args *ya* es una tupla y se usa como clave en el dict.
2) Parece que memoize no viene con Python. Lo busqué por ese nombre y encontré este código que es muy parecido al mío, aunque también soporta decorar funciones con kwargs.

• Responder • Compartir ›

Nacho • hace 4 años

Bárbaro Juanjo!
Esto que contás de caché lo conocía como Memoization [1].
Respecto a las notas, no se si es una limitación lo que decís acerca del objeto mutable. Si el objeto fuera mutable entonces no se si estaría garantizada la salida como para poder almacenarla. Por lo tanto me parece correcto que esté limitado en ese sentido.

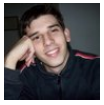
Saludos!
[1] http://en.wikipedia.org/wiki/M...

• Responder • Compartir ›

jjconti Moderador • hace 4 años

En un hilo en la lista de correos de PyAr se habló de los inconvenientes de utilizar un dict como caché y de las ventajas de usar otros enfoques como LRU.

• Responder • Compartir ›



jjconti Moderador • hace 4 años

Me acabo de dar cuenta que se puede implementar el decorador caché sin clases, con funciones directamente...

• Responder • Compartir ›

jjconti Moderador ➦ jjconti • hace 4 años

Valió como ejemplo.

• Responder • Compartir ›

Ruben • hace 3 años

Excelente artículo!!
Una duda ... Cuando implementas funciones caché con clases decoradoras , haces:

@mem
def f(arg):

return (arg ** 10) * (arg ** -5)

Luego para mostrar resultados haces:

>>> fmem(1)
1.0
>>> fmem(2)
32.0
>>> fmem(2)
Recuperando valor de la memoria
32.0

fmem?
parece un error de tipeo...
¿No habria que hacer algo como así?

a = mem(f)
print a(1)
print a(2)
print a(2)

Perdón si meto la pata, no soy prograndor
Rubén

• Responder • Compartir ›

Franko ➚ Ruben • hace 5 meses

Si, es un error de typeo, lo correcto sería solamente poner:
>>> f(1)
1.0
>>> f(2)
32.0
>>>f(2)
Recuperando valor de la memoria
32.0

• Responder • Compartir ›