

Python: Programación funcional

Curso Programación C



www.seas.es

Aprende a programar fácilmente en
este lenguaje. ¡Infórmate aquí!

+1

3

Twittear

6

Recomendar

7

La programación funcional es un paradigma en el que la programación se basa casi en su totalidad en funciones, entendiendo el concepto de función según su definición matemática, y no como los simples subprogramas de los lenguajes imperativos que hemos visto hasta ahora.

En los lenguajes funcionales puros un programa consiste exclusivamente en la aplicación de distintas funciones a un valor de entrada para obtener un valor de salida.

Python, sin ser un lenguaje puramente funcional incluye varias características tomadas de los lenguajes funcionales como son las funciones de orden superior o las funciones lambda (funciones anónimas).

Funciones de orden superior

El concepto de funciones de orden superior se refiere al uso de funciones como si de un valor cualquiera se tratara, posibilitando el pasar funciones como parámetros de otras funciones o devolver funciones como valor de retorno.

Esto es posible porque, como hemos insistido en diversas ocasiones, en Python todo son objetos. Y las funciones no son una excepción.

Veamos un pequeño ejemplo

```
def saludar(lang):
    def saludar_es():
        print "Hola"

    def saludar_en():
        print "Hi"

    def saludar_fr():
        print "Salut"

    lang_func = {"es": saludar_es,
                 "en": saludar_en,
                 "fr": saludar_fr}
    return lang_func[lang]

f = saludar("es")
f()
```

Como podemos observar lo primero que hacemos en nuestro pequeño programa es llamar a la función `saludar` con un parámetro `"es"`. En la función `saludar` se definen varias funciones: `saludar_es`, `saludar_en` y `saludar_fr` y a continuación se crea un diccionario que tiene como claves cadenas de texto que identifican a cada lenguaje, y como valores las funciones. El valor de retorno de la función es una de estas funciones. La función a devolver viene determinada por el valor del parámetro `lang` que se pasó como argumento de `saludar`.

Como el valor de retorno de `saludar` es una función, como hemos visto, esto quiere decir que `f` es una variable que contiene una función. Podemos entonces llamar a la función a la que se refiere `f` de la forma en que llamaríamos a cualquier otra función, añadiendo unos paréntesis y, de forma opcional, una serie de parámetros entre los paréntesis.

Esto se podría acortar, ya que no es necesario almacenar la función que nos pasan como valor de retorno en una variable para poder llamarla:

```
>>> saludar("en")()
```

```
Hi
>>> saludar("fr")()
Salut
```

En este caso el primer par de paréntesis indica los parámetros de la función `saludar`, y el segundo par, los de la función devuelta por `saludar`.

Iteraciones de orden superior sobre listas

Una de las cosas más interesantes que podemos hacer con nuestras funciones de orden superior es pasarlas como argumentos de las funciones `map`, `filter` y `reduce`. Estas funciones nos permiten sustituir los bucles típicos de los lenguajes imperativos mediante construcciones equivalentes.

`map(function, sequence[, sequence, ...])`

La función `map` aplica una función a cada elemento de una secuencia y devuelve una lista con el resultado de aplicar la función a cada elemento. Si se pasan como parámetros `n` secuencias, la función tendrá que aceptar `n` argumentos. Si alguna de las secuencias es más pequeña que las demás, el valor que le llega a la función `function` para posiciones mayores que el tamaño de dicha secuencia será `None`.

A continuación podemos ver un ejemplo en el que se utiliza `map` para elevar al cuadrado todos los elementos de una lista:

```
def cuadrado(n):
    return n ** 2

l = [1, 2, 3]
l2 = map(cuadrado, l)
```

`filter(function, sequence)`

La función `filter` verifica que los elementos de una secuencia cumplan una determinada condición, devolviendo una secuencia con los elementos que cumplen esa condición. Es decir, para cada elemento de `sequence` se aplica la función `function`, si el resultado es `True` se añade a la lista y en caso contrario se descarta.

A continuación podemos ver un ejemplo en el que se utiliza `filter` para conservar solo los números que son pares.

```
def es_par(n):
    return (n % 2.0 == 0)

l = [1, 2, 3]
l2 = filter(es_par, l)
```

reduce(function, sequence[, initial])

La función reduce aplica una función a pares de elementos de una secuencia hasta dejarla en un solo valor.

A continuación podemos ver un ejemplo en el que se utiliza `reduce` para sumar todos los elementos de una lista.

```
def sumar(x, y):  
    return x + y  
  
l = [1, 2, 3]  
l2 = reduce(sumar, l)
```

Funciones lambda

El operador lambda sirve para crear funciones anónimas en línea. Al ser funciones anónimas, es decir, sin nombre, estas no podrán ser referenciadas más tarde.

Las funciones lambda se construyen mediante el operador lambda, los parámetros de la función separados por comas (atención, SIN paréntesis), dos puntos (:) y el código de la función.

Esta construcción podrían haber sido de utilidad en los ejemplos anteriores para reducir código. El programa que utilizamos para explicar `filter`, por ejemplo, podría expresarse así:

```
l = [1, 2, 3]  
l2 = filter(lambda n: n % 2.0 == 0, l)
```

Comparemoslo con la versión anterior:

```
def es_par(n):  
    return (n % 2.0 == 0)  
  
l = [1, 2, 3]  
l2 = filter(es_par, l)
```

Las funciones lambda están restringidas por la sintaxis a una sola expresión.

Comprensión de listas

En Python 3 `map` y `filter` se verán sustituidas por las list comprehensions o comprensión de listas, característica tomada del lenguaje de programación funcional Haskell y que está presente en Python desde la versión 2.0.

La comprensión de listas es una construcción que permite crear listas a partir de otras listas. Cada una de estas construcciones consta de una expresión que determina cómo modificar el elemento de la lista original, seguida de una o varias cláusulas `for` y opcionalmente una o varias cláusulas `if`.

Veamos un ejemplo de cómo se podría utilizar la comprensión de listas para elevar al cuadrado todos los elementos de una lista, como hicimos en nuestro ejemplo de `map`.

```
l2 = [n ** 2 for n in l]
```

Esta expresión se leería como “para cada `n` en `l` haz `n ** 2`”. Como vemos tenemos primero la expresión que modifica los valores de la lista original (`n ** 2`), después el `for`, el nombre que vamos a utilizar para referirnos al elemento actual de la lista original, el `in`, y la lista sobre la que se itera.

El ejemplo que utilizamos para la función `filter` (conservar solo los números que son pares) se podría expresar con comprensión de listas así:

```
l2 = [n for n in l if n % 2.0 == 0]
```

Veamos por último un ejemplo de comprensión de listas con varias cláusulas `for`:

```
l = [0, 1, 2, 3]
m = ["a", "b"]
n = [s * v for s in m
      for v in l
      if v > 0]
```

Esta construcción sería equivalente a una serie de `for-in` anidados:

```
l = [0, 1, 2, 3]
m = ["a", "b"]
n = []

for s in m:
    for v in l:
        if v > 0:
            n.append(s* v)
```

Generadores

Las expresiones generadoras funcionan de forma muy similar a la comprensión de listas. De hecho su sintaxis es exactamente igual, a excepción de que se utiliza paréntesis en lugar de corchetes:

```
l2 = (n ** 2 for n in l)
```

Sin embargo las expresiones generadoras se diferencian de la comprensión de listas en que no se devuelve una lista, sino un generador.

```
>>> l2 = [n ** 2 for n in l]
>>> l2
[0, 1, 4, 9]
>>> l2 = (n ** 2 for n in l)
>>> l2
<generator object at 0x00E33210>
```

Un generador es una clase especial de función que *genera* valores sobre los que iterar. Para devolver el siguiente valor sobre el que iterar se utiliza la palabra clave `yield`. Veamos por ejemplo un generador que devuelva números de `n` a `m` con un salto `s`.

```
def mi_generador(n, m, s):
    while(n <= m):
        yield n
        n += s
```

```
>>> x = mi_generador(0, 5, 1)
>>> x
<generator object at 0x00E25710>
```

El generador se puede utilizar en cualquier lugar donde se necesite un objeto iterable. Por ejemplo en un `for-in`:

```
for n in mi_generador(0, 5, 1):
    print n
```

Como no estamos creando una lista completa en memoria, sino generando un solo valor cada vez que se necesita, en situaciones en las que no sea necesario tener la lista completa el uso de generadores puede suponer una gran diferencia de memoria. En todo caso siempre es posible crear una lista a partir de un generador mediante la función `list`:

```
lista = list(mi_generador)
```

Decoradores

Un decorador no es es mas que una función que recibe una función como parámetro y devuelve otra función como resultado. Por ejemplo podríamos querer añadir la funcionalidad de que se imprimiera el nombre de la función llamada por motivos de depuración:

```
def mi_decorador(funcion):  
    def nueva(*args):  
        print "Llamada a la funcion", funcion.__name__  
        retorno = funcion(*args)  
        return retorno  
    return nueva
```

Como vemos el código de la función `mi_decorador` no hace más que crear una nueva función y devolverla. Esta nueva función imprime el nombre de la función a la que “decoramos”, ejecuta el código de dicha función, y devuelve su valor de retorno. Es decir, que si llamáramos a la función `nueva` que nos devuelve `mi_decorador`, el resultado sería el mismo que el de llamar directamente a la función que le pasamos como parámetro, exceptuando el que se imprimirá además el nombre de la función.

Supongamos como ejemplo una función `imp` que no hace otra cosa que mostrar en pantalla la cadena pasada como parámetro.

```
>>> imp("hola")  
hola  
>>> mi_decorador(imp)("hola")  
Llamada a la función imp  
hola
```

Ahora bien, la sintaxis para llamar a la función que nos devuelve `mi_decorador` no es muy clara, aunque si lo estudiamos detenidamente veremos que no tiene mayor complicación. Primero se llama a la función que decora con la función a decorar: `mi_decorador(imp)`; y una vez obtenida la función ya decorada se la puede llamar pasando el mismo parámetro que se pasó anteriormente: `mi_decorador(imp)("hola")`

Esto se podría expresar más claramente precediendo la definición de la función que queremos decorar con el signo `@` seguido del nombre de la función decoradora:

```
@mi_decorador  
def imp(s):  
    print s
```

De esta forma cada vez que se llame a `imp` se estará llamando realmente a la versión decorada. Python incorpora esta sintaxis desde la versión 2.4 en

adelante.

Si quisiéramos aplicar más de un decorador bastaría añadir una nueva línea con el nuevo decorador.

```
@otro_decorador
@mi_decorador
def imp(s):
    print s
```

Es importante advertir que los decoradores se ejecutarán de abajo a arriba. Es decir, en este ejemplo primero se ejecutaría `mi_decorador` y después `otro_decorador`.



[decoradores](#), [funcional](#), [funciones](#), [generadores](#), [lambda](#), [programacion](#), [python](#), [tutorial](#)

Comentarios

1. [Blaxter](#)

Este es el punto más flojo de python (junto con metaprogramación), en mi opinión, frente a ruby.

[Responder](#)

2. [Zootropo](#)

A parte de que en Ruby las funciones lambda pueden ser más complejas, ¿qué más echas en falta Blaxter?

Yo de todas formas no suelo usar mucho estas características.

Responder

3. Fox

Uhm, esto está muy bien, luego lo leo.

Es util usar map y filter, pero sobre todo es muuuuuuuuuuuuuuuuy util usar las list comprehension, es mucho más rapido que usar un for, pero mucho mas 😊

Responder

4. Kartoffel

A mí también me encantan las listas por comprensión, son rápidas e intuitivas con un poco de práctica 😊

Responder

5. javielinux

A leer de nuevo, mañana me pongo pq me parece muy interesante

Estoy también interesado en internacionalizacion ¿usáis gettext en python? ¿algun tuto interesante sobre este tema?

un saludo y gracias de nuevo

Responder6. kEpEx

Me queda duda si esta correcto donde comparas:

מקור

$$1 = [1, 2, 3]$$

```
l2 = map(lambda n: n ** 2, l)
```

Comparemoslo con la versión anterior:

```
def es_par(n):
```

Este sitio usa cookies para mejorar su experiencia. Si continúa en el mismo, consideramos que acepta su uso

[Aceptar](#)

```
l = [1, 2, 3]
```

```
l2 = filter(es_par, l)
```

```
"""
```

Estas comparando la version que eleva al cuadrado con la que checa si es Par o no.

Saludos

[Responder](#)

7. [Zootropo](#)

Fallo mio kEpEx, cierto. Edito.

[Responder](#)

8. [Zootropo](#)

Yo he usado gettext alguna vez javielinux. Es bastante sencillo. A ver si escribo un pequeño tutorial cuando termine con los temas más básicos de Python.

Se haría algo así:

```
import gettext
gettext.install("ejemplo")
print _("Cadena il8n")
```

Luego para generar la plantilla:

```
pygettext -o ej.pot ej.py
```

[Responder](#)

9. [Marshal](#)

Saludos de nuevo, pasando por aca y probando un poco los nuevos conocimientos 😊 encuentro que no se como crear lo que denominan una “función validadora” para un cuestionario que se da en la aplicacion que escribo...tienen alguna idea de como se hace en python?

Gracias de antemano

[Responder](#)

10. [Todo un despropósito :: Retos computacionales. :: February :: 2009](#)

[...] Si el artículo sobre map y reduce os resulta interesante, os recomiendo este otro más extenso:

<http://mundogeek.net/archivos/2008/03/10/python-programacion-funcional/>. [...]

[Responder](#)

11. Ober

El map, Filter y el reduce es increíblemente útil ya que con ello se puede reducir código enormemente, por ejemplo tengo el siguiente código para hallar los n primeros números primos:

```
# -*- coding: iso-8859-1 -*-  
def primo(numero):  
    if len(filter(lambda m: numero % m == 0, [x for x in range(2, (numero/2)+1)])) == 0: return numero  
  
print filter(lambda valor: valor != None, map(primo, range(2, int(raw_input("Ingrese Limite: "))))
```

[Responder](#)

12. Juan pedro

mm muy interesante estoy aprendiendo a programar en python

[Responder](#)

13. Anónimo

me gustaria ver mas ejemplos sobre for in anidados me seria de gran ayuda

[Responder](#)

14. Mike Schumager

Hola, soy nuevo en programación Python y necesito urgente una ayuda. Estoy tratando de obtener las combinaciones de 10 elementos, tomados

de 5 en cinco (combinaciones, en las cuales sus elementos no se repiten y que es equivalente a la fórmula $n!/r!(n-r)!$; donde n =numero de elementos y r el rango. Según la fórmula son 252 combinaciones en total). Hasta aquí no hay problemas. El resultado debe cumplir, que los elementos de las combinaciones deben ser ≥ 37 . Osea, solo quiero un espectro de las combinaciones que totalicen 37. El método debe ser utilizando `itertools.combinations`.

Agradezco su atención y ayuda.

[Responder](#)

15. Anónimo

esta chido

[Responder](#)

16. [retro jordan sneakers for sale](#)

This is a great shoe b/c it gives great ankle support and at the same time is very light. Also looks great retro jordan sneakers for sale <http://www.jordanshoesverycheap.com/>

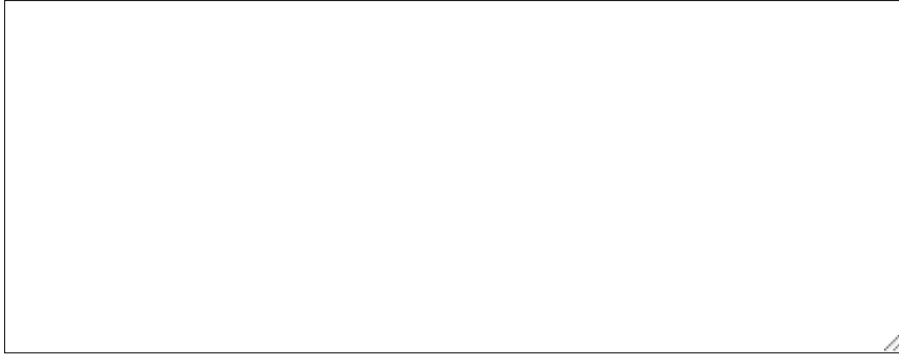
[Responder](#)

Deja un comentario

Nombre

email (no se mostrará)

Tu web (opcional)



Enviar comentario



[Mundo geek](http://mundogeek.net) es una web escrita por Raúl González Duque, dedicada principalmente a las nuevas tecnologías y la informática.

21785	lectores	5388	Followers
BY FEEDBURNER		twittercounter.com	



Mundo geek

Me gusta

A 7417 personas les gusta [Mundo geek](#).



Plug-in social de Facebook

- [Archivos](#)
- [Acerca](#)
- [Contacto](#)
- [Traducciones](#)
- [Wiki](#)

HOSTING COMPARTIDO

¡Hasta un año gratis!
En altas y renovaciones

desde
1,89€
/mes

VER PLANES 

[Aviso legal](#)