

Programación Funcional

Antonio Espín Herranz

Programación funcional

- 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.

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.
- Una función para python es un objeto.

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() # ejecuta la función seleccionada.
```

Con el parámetro lang seleccionamos la función que vamos a ejecutar. **lang** representa la clave de un diccionario donde los valores son los nombres de las funciones.

Ejemplo

- 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 con 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.
- *Tener en cuenta que estas funciones **están obsoletas** en **python 3** y se sustituyen por **list comprehension**.*

map

- **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.

Ejemplo

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

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

```
l2 = list(map(cuadrado, l)) # map devuelve un iterador y se utiliza para crear una nueva lista.
```

- A todos los elementos de la lista l se le aplica la función cuadrado.
- l2 será [1, 4, 9]
- **Mejor utilizar:**
L2 = [cuadrado(i) for i in L]

filter

- **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.

Ejemplo

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

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

```
l2 = list(filter(es_par, l)) # igual que map devuelve un iterador.
```

- Aplica la función `es_par` a todos los elementos de la lista `l`, para los elementos que la función devuelve `true` se añaden a la lista `l2`.
- En este caso `l2` será `[2]`.
- Mejor utilizar:

```
L2 =[i for i in L if i % 2 == 0]
```

reduce

- **reduce(function, sequence[, initial])**
 - La función reduce aplica una función a pares de elementos de una secuencia hasta dejarla en un solo valor.

```
import functools
```

```
def sumar(x, y):  
    return x + y
```

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

```
l2 = functools.reduce(sumar, l)
```

El resultado sería 6.

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 (**SIN paréntesis**), dos puntos (:) y el **código de la función**.

Ejemplo

- `L = [1, 2, 3]`
- `L2 = list(filter(lambda n: n % 2.0 == 0, L))`

Cabecera de
La función



Cuerpo de
La función

Compresión de listas

- **En Python 3.x map, filter y reduce pierden importancia y se consideran obsoletas.**
- Estas funciones se mantendrán, reduce pasará a formar parte del módulo functools, con lo que quedará fuera de las funciones disponibles por defecto, y map y filter **se desaconsejarán en favor de las *list comprehensions* o *comprensión de listas*.**

Comprensión de listas

- Calcular el cuadrado de cada valor de la lista:
 $L2 = [n ** 2 \text{ for } n \text{ in } L]$
 - Esta expresión se leería como “**para cada n en L haz n ** 2**”.
 - **n** va tomando los distintos valores de la lista.
- Conservar **sólo los elementos pares**:
 $L2 = [n \text{ for } n \text{ in } L \text{ if } n \% 2.0 == 0]$

Ejemplo

```
L = [0, 1, 2, 3]
```

```
m = ["a", "b"]
```

```
n = [s * v for s in m
```

```
      for v in L
```

```
      if v > 0]
```

```
print (n)
```

```
['a', 'aa', 'aaa', 'b', 'bb', 'bbb']
```


Compresión de Listas

- También se puede utilizar para generar diccionarios y conjuntos.

```
>>> import random
>>> d = {random.randint(1,50) for i in range(10)}
>>> d
{36, 7, 8, 44, 46, 48, 17, 19, 21, 31}
>>> type(d)
<class 'set'>
>>>
```

```
>>> d = {i:random.randint(10,20) for i in range(10)}
>>> d
{0: 11, 1: 15, 2: 13, 3: 11, 4: 11, 5: 14, 6: 18, 7: 18, 8: 17, 9: 14}
>>> type(d)
<class 'dict'>
>>> _
```

Generadores

- Las expresiones generadoras funcionan de forma muy similar a la comprensión de listas.
- La sintaxis es exactamente igual, a excepción de que se utilizan paréntesis en lugar de corchetes:

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

Generadores

- **No se devuelve una lista, sino un generador.**

Con la lista:

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

```
>>> l2
```

```
[0, 1, 4, 9]
```

Con el generador:

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

```
>>> l2
```

```
<generator object at 0x00E33210>
```

Generadores

- 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** en lugar de **return**

```
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.

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

Decoradores

- Un decorador no 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
```

```
>>> imp("hola")  
hola
```

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

Decoradores

- Para implementar decoradores mejor utilizar anotaciones:
- Por una lado tenemos la función y el decorador:
- Funciona como un interceptor. Crea un envoltorio alrededor de la función.

```
def mi_decorador(metodo):  
    def funcion_interna(*args):  
        # código de la función envolvente ...  
  
        # Al final llamamos a la función para que se ejecute  
        metodo(args)  
    return funcion_interna  
  
# En las funciones que quiero aplicar el decorador las anoto por encima  
# con: @nombre_funcion  
  
@mi_decorador  
def funcionExterna(param1, param2, ...):  
    #código de la función
```