Introducción a la programación en Python

Tema 8. Programación Funcional

Autor: Francisco Mariño Ruiz

Contenido

1 Programación Funcional 3
1.1 Introducción 3
1.2 Funciones de orden superior 4
1.3 Iteracciones de orden superior sobre listas 6
1.4 Funciones lambda 9
1.5 Compresión de listas 10
1.6 Generadores 12
1.7 Decoradores 15
2 Bibliografía 19

1 Programación Funcional

1.1 Introducción

En el siguiente capítulo, veremos lo que es la “programación funcional”. Se trata de un paradigma en el que la programación se basa casi totalmente en el uso de funciones. No nos referiremos a funciones como subprogramas como los que hemos visto hasta ahora, si no a funciones desde el punto de vista funcional.

En lenguajes puramente funcionales, un programa consistirá exclusivamente en aplicar distintas funciones a un valor de entrada para obtener un valor de salida. Python, que no es un lenguaje puramente funcional, incluye varias características tomadas de los lenguajes funcionales como son las funciones de orden superior o las funciones lambda.

1.2 Funciones de orden superior

Se denominan así a aquellas funciones que pueden tener funciones como parámetros de entrada o devolver funciones como resultado. Podrán cumplir una o ambas características. Se trata de usar las funciones como si de una variable se tratase.

En el siguiente ejemplo, veremos cómo se genera una función de este tipo:

```
1  def saludar(lang):
2      def saludar_es():
3          print ("Hola")
4      def saludar_en():
5          print ("Hi")
6      def saludar_fr():
7          print ("Salut")
8      lang_func = {"es": saludar_es, "en": saludar_en, "fr": saludar_fr}
9      return lang_func[lang]
```

Si observamos el código, podemos ver que en la primera línea definimos una nueva función "saludar", que será nuestra función de orden superior. A continuación, dentro de esta función se definen varias funciones "saludar_es", "saludar_en" y "saludar_fr" y posteriormente un diccionario clave:valor donde se asigna a cada uno de los posibles idiomas de entrada (los que el programador ha decidido) una función que será la que devuelva en su respuesta la función de orden superior. Por tanto, lo que hará la función es que cuando se llame con un determinado parámetro, devolverá la función que se corresponda con ese parámetro. Esta función podrá ser almacenada en un parámetro, pero no se ejecutará en esta primera llamada.

Veamos cómo se utilizaría. En primer lugar, llamamos a la función como lo haríamos normalmente:

```
10 f = saludar("es")
11 print(f)
```

```
<function saludar.<locals>.saludar_es at 0x000001C3B889D438>
```

A pesar de que en un primer momento pudiésemos pensar que llamando a la función con un lenguaje nos imprimirá el saludo en ese lenguaje, como hemos dicho antes, la salida que obtendremos es la función de saludo que se corresponde con el lenguaje indicado.

Si almacenamos el resultado de llamar a saludar en una variable, podremos llamarla tantas veces como queramos sin tener que especificar más valores.

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

```
Hola
```

En caso de que queramos utilizar la función directamente, sin necesidad de almacenarla en una variable también podríamos hacerlo, pero del siguiente modo:

```
14 print(saludar(“es”)())
```

```
Hola
```

Lo que hemos hecho es pasar el primer par de paréntesis con el parámetro para llamar a la función de orden superior y el segundo par para llamar a la función que devuelve la primera y que de este modo se ejecute.

1.3 Iteraciones de orden superior sobre listas

Siguiendo con este paradigma de programación funcional podemos utilizar las funciones "*map*", "*filter*" y "*reduce*" que nos permitirán escribir código más simple y corto sin necesidad de tener que preocuparnos por todas las complejidades que a veces pueden tener los bucles y las ramificaciones.

1.3.1 Función *map*

La función "*map*" nos permite aplicar una determinada función a cada elemento de una secuencia que le proporcionemos, devolviendo un objeto de tipo "*map*", que es fácilmente transformable a una lista. Un ejemplo sería el siguiente:

```
1  def cuadrado(n):
2      return n * n
3
4  lista = [1, 2, 3, 4]
5  salida = map(cuadrado, lista)
6  salida2 = list(salida)
7  print (salida2)
```

```
[1, 4, 9, 16]
```

En primer lugar, hemos definido la función que vamos a utilizar, posteriormente hemos definido la lista de valores sobre la que queremos utilizar con la función y finalmente hemos utilizado la función "*map*" pasándole el nombre de la función que queremos utilizar y la lista de elementos que serán sus parámetros de entrada.

Como podemos observar, hemos tenido que hacer una conversión explicita para pasar la variable salida a una variable de tipo lista ("*salida2*"), ya que la función nos devuelve un objeto de tipo "*map*".

Si la función tuviese más parámetros, en la función "*map*" tendríamos que pasar tantas listas como parámetros tuviese la función que fuésemos a utilizar. Ejemplo:

```
1  def potencia(n, m):
2      return n ** m
3
4  lista = [1, 2, 3, 4]
5  lista2 = [2, 3, 4, 5]
6  salida = map(potencia, lista, lista2)
7  salida2 = list(salida)
8  print (salida2)
```

```
[1, 8, 81, 1024]
```

1.3.2 Función *filter*

La función "*filter*" nos va a permitir verificar si los elementos de una lista cumplen una determinada condición que estableceremos en una función. Básicamente, la función "*filter*" toma los valores de la lista, examina si cumplen la condición que establece la función y en caso de que se cumpla los añade a la lista de salida, descartando todos aquellos que no lo cumplan.

Veamos un ejemplo:

```
1  def par(n):
2      return (n % 2.0 == 0)
3
4  lista = [1, 2, 3, 4]
5  salida = filter(par, lista)
6  salida2 = list(salida)
7  print (salida2)
```

```
[2, 4]
```

En primer lugar, definimos una función que evalúa si un número es par (el resto de la división entre dos es igual a cero). A continuación, utilizamos la función "*filter*", pasando como argumentos la función que hemos creado y la lista de números. Tal y como veíamos en la función "*map*" se devuelve un objeto de tipo "*filter*" por lo que tenemos que hacer una transformación explícita a lista para poder ver su contenido. Como esperábamos, el resultado que hemos obtenido son los valores pares de la lista.

1.3.3 Función *reduce*

La función "*reduce*" toma una lista de elementos y aplica una función sobre los elementos de dos en dos hasta que sólo quede un elemento. Supongamos la lista de números que teníamos en los ejemplos anteriores [1, 2, 3, 4] y una función sencilla:

```
1  def suma(n, m):
2      return n + m
```

La función "*reduce*" haría lo siguiente:

- Toma los dos primeros valores de la lista (1 y 2) y los suma obteniendo el resultado 3, por lo que la lista quedaría [3, 3, 4].
- Toma los dos primeros valores de la lista actual (3 y 3) y los suma obteniendo el resultado 6 y actualiza la lista [6, 4].
- Repite la operación hasta acabar, en este caso sería sólo una operación más con la que obtendríamos el resultado 10.

Al contrario que las funciones "*map*" y "*filter*", esta función devuelve directamente el valor de salida como resultado. Además, la función necesita ser importada para poder ser usada.

```
1  from functools import reduce
2
3  def suma(n, m):
4      return n + m
```

```
5
6  lista = [1, 2, 3, 4]
7  salida = reduce(suma, lista)
8  print (salida)
```

```
10
```

La función "*reduce*" nos permitiría utilizar cualquier función que necesite como entrada dos elementos del mismo tipo y devuelva un solo elemento, que ha de ser del mismo tipo que los elementos de entrada. En el ejemplo hemos utilizado números, pero podríamos haber usado cadenas o cualquier otro tipo.

1.4 Funciones lambda

Las funciones lambda son aquellas funciones que se crean sin nombre en la misma línea en la que se van a usar y que no son reutilizables. Son útiles para utilizar en los iteradores de orden superior que hemos visto en el punto anterior sin tener que definir una función específica previamente. Por limitaciones de sintaxis, estas funciones sólo se podrán utilizar cuando las funciones solamente lleven una única instrucción.

Las funciones lambda se construyen mediante la palabra clave lambda seguida de un espacio y los parámetros de la función separados por comas. A continuación, vienen los dos puntos (sin paréntesis) y el código de la función.

Veamos cómo se aplicaría en los ejemplos anteriores:

```
1  lista = [1, 2, 3, 4]
2  salida = map(lambda n: n * n, lista)
3  salida2 = list(salida)
4  print (salida2)
```

```
[1, 4, 9, 16]
```

```
1  lista = [1, 2, 3, 4]
2  salida = filter(lambda n: n % 2.0 == 0, lista)
3  salida2 = list(salida)
4  print (salida2)
```

```
[2, 4]
```

```
1  from functools import reduce
2
3  lista = [1, 2, 3, 4]
4  salida = reduce(lambda n, m: n + m, lista)
5  print (salida)
```

```
10
```

Podemos comprobar, que al tratarse de funciones sencillas que se crean solamente para utilizar con las funciones de iteración, es muy útil para reducir código.

1.5 Compresión de listas

Las funciones "*map*", "*filter*" y "*reduce*" son heredadas de las versiones 2.X de Python. Anteriormente estaban las 3 disponibles sin necesidad de hacer importaciones y tanto "*map*" como "*filter*" ofrecían listas como salida sin necesidad de hacer conversiones de tipo. En las versiones 3.X se mantiene la existencia de las funciones "map" y "filter", pero se añaden las funciones de compresión de listas, las cuales se aconseja que se utilicen para sustituir a estas.

La compresión de lista es una característica que se ha tomado de otros lenguajes de programación funcional y que consiste en una construcción que nos permite crear listas a partir de otras listas.

Cada una de estas construcciones consta de una expresión que determina como se modificará la lista de entrada, seguida de una o varias cláusulas "*for*" y opcionalmente una o varias cláusulas "*if*". Por ejemplo, a la hora de sustituir la función "*map*" que veíamos anteriormente, utilizaríamos el siguiente código:

```
1  lista = [1, 2, 3, 4]
2  salida = [n *n for n in lista]
3  print (salida)
```

```
[1, 4, 9, 16]
```

Como vemos, en una sola línea hemos generado una nueva lista con una expresión que literalmente se leería: Multiplica n por sí mismo, siendo n cada elemento de la lista de entrada.

Podríamos hacer lo mismo con el ejemplo de la función "*filter*":

```
1  lista = [1, 2, 3, 4]
2  salida = [n for n in lista if n % 2 == 0]
3  print (salida)
```

```
[2, 4]
```

Literalmente le estamos diciendo que nos cree una nueva lista que contenga cada elemento n de lista si cumple la condición de ser par.

Como hemos comentado anteriormente podremos utilizar la compresión de listas con más de un "*for*", esto nos permitirá replicar el comportamiento de dos bucles anidados. Veamos otro ejemplo:

```
1  lista = [1, 2, 3, 4]
2  lista2 = ['a', 'b']
3  salida = []
4
5  for n in lista:
6      for m in lista2:
7          salida.append(n * m)
8
9  print (salida)
```

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

Si vemos el código, vemos una lista de números, una lista de caracteres y una lista vacía, a continuación, un doble bucle que recorre las dos listas y añade a la lista de valores la multiplicación de los valores de la primera lista por la segunda. Un código que podríamos simplificar utilizando la compresión de listas:

```
1  lista = [1, 2, 3, 4]
2  lista2 = [‘a’, ‘b’]
3  salida = [n * m for n in lista for m in lista2]
4
5  print (salida)
```

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

Hemos hecho el doble bucle “*for*” en una sola línea de código que directamente genera la lista de salida.

1.6 Generadores

Las expresiones generadoras son expresiones que funcionan de un modo muy similar a la compresión de listas, utilizando una sintaxis idéntica salvo por el uso de paréntesis en lugar de corchetes. La diferencia es que, en lugar de devolver una lista, como en el caso de la compresión de listas, devuelve un objeto de tipo generador.

```
lista = [1, 2, 3, 4]

salida = [n *n for n in lista]
print (salida)
```

```
[1, 4, 9, 16]
```

```
salida = (n *n for n in lista)
print (salida)
```

```
<generator object <genexpr> at 0x0000017A74FD08E0>
```

Un generador es una clase especial de función que genera valores sobre los que iterar en tiempo real, es decir, los valores no se almacenan en una lista, si no que se van creando según se necesitan para iterar sobre ellos. Para ello, la función va a utilizar la palabra clave “yield” en lugar de “return” para devolver el siguiente valor de iteración. Veamos un ejemplo sencillo:

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

x = primer_generador(0, 5, 1)
print (x)
```

```
<generator object primer_generador at 0x0000017A74FD0E60>
```

Hemos creado un generador que contiene números entre 0 y 5 ambos incluido. Este generador podría utilizarse en cualquier momento cuando necesitemos un objeto iterable, por ejemplo, en un bucle “for-in”:

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

```
0
1
2
3
4
5
```

Como vemos, el generador actúa de la misma manera en la que se haría si estuviésemos recorriendo una lista, pero la diferencia radica en que una lista debemos tenerla guardada en memoria, mientras que un generador no necesita estar guardado, si no que genera los valores según se necesitan para cada iteración y los desecha al terminar la iteración. Esta diferencia nos

permitirá optimizar el uso de memoria en aquellos casos en los que no sea necesario guardar la lista en memoria.

La función generadora podrá ser todo lo compleja que queramos, de modo que se generen los valores al vuelo y se utilicen. Veamos otro ejemplo:

```
1  def generador_multiplos(n, m):
2      mult = 1
3      while (mult <= m):
4          yield n * mult
5          mult += 1
6
7  for n in generador_multiplos(7, 5):
8      print (n)
```

```
7
14
21
28
35
```

En esta ocasión hemos creado una función que nos devuelva los m primeros múltiplos de un número n.

También podríamos crear generadores infinitos, si no establecemos un condición de finalización, por ejemplo:

```
1  def genera_infinito(n):
3      while True:
4          yield n
5          n += 1
6
7  for n in genera_infinito(7):
8      print (n)
```

```
7
8
9
10
11
...
```

El código se ejecutaría hasta que lo parásemos manualmente (CONTROL + C si lo estás ejecutando directamente o consola o STOP si estás en PyCharm). Nos podría servir para ejecutarlo hasta que se cumpla otra condición externa que establezcamos en el for, que con un break podría parar la ejecución.

```
1  def genera_infinito(n):
2      while True:
3          yield n
4          n += 1
5
6  for n in genera_infinito(7):
```

```
7       print (n)
8       if n > 10:
9           break
```

```
7
8
9
10
11
```

1.7 Decoradores

Un decorador es una función que recibe una función como parámetro de entrada y devuelve otra función como resultado. Su uso principal es para modificar el comportamiento de otras funciones. Veamos un ejemplo sencillo:

```
1  def decorador(funcion):
2      def funcion_decorada(a, b):
3          print (“Se va a llamar a la función”)
4          c = funcion(a, b)
5          print (“Ha terminado la función”)
6          return c
7      return funcion_decorada
8
9  @decorador
10 def suma(a, b):
11     print (“Sumando…”)
12     return a + b
13
14 suma(5, 8)
```

```
Se va a llamar a la función
Sumando…
Se ha llamado la función
```

Analicemos el código. En primer lugar, hemos definido la función decoradora, que tiene como parámetro de entrada una función. A continuación, definimos una nueva función llamada “*funcion_decorada*”. En este caso la nueva función es sencilla, hace un “*print*” para decir que se va a llamar a la función, ejecuta la función que se pasó como entrada al decorador, almacenando el resultado en una variable, hace un nuevo “*print*” para decir que se ha ejecutado la función y devuelve el resultado de la función que se ha ejecutado. Finalmente, el decorador devuelve la “*función_decorada*”.

En la línea 9 del código vemos como se llama a un decorador, para ello utilizamos el carácter “@” seguido del nombre del decorador y tras el definimos la función, en este caso una sencilla función de suma. En la última línea, ejecutamos la función, a la que ya se ha aplicado el decorador. Podríamos utilizar el decorador sobre otra función:

```
15 @decorador
16 def resta(a, b):
17     print (“Restando…”)
18     return a - b
19
20 resta(5, 8)
```

```
Se va a llamar a la función
Restando…
Se ha llamado la función
```

En el ejemplo que hemos visto, el decorador no hace mucho, simplemente añade un par de líneas genéricas al código, pero podríamos hacer que se personalizase un poco más.

```
def decorador(funcion):
    def funcion_decorada(a, b):
        print (“Se va a llamar a la función”, funcion.__name__)
        c = funcion(a, b)
        print (“Ha terminado la función”, funcion.__name__, “con resultado”, c)
        return c
    return funcion_decorada

@decorador
def suma(a, b):
    print (“Sumando…”)
    return a + b

suma(5, 8)

@decorador
def resta(a, b):
    print (“Restando…”)
    return a - b

resta(5, 8)
```

```
Se va a llamar a la función suma
Sumando…
Se ha llamado la función suma con resultado 13

Se va a llamar a la función resta
Restando…
Se ha llamado la función resta con resultado -3
```

En este caso hemos añadido datos sobre la función que hemos ejecutado en el decorador, de forma que tenga más sentido el añadir un decorador a las funciones. Aunque en este sencillo ejemplo tan solo hemos añadido unas líneas impresas a las funciones, los decoradores nos permiten modificar las funciones de cualquier manera, de forma que podamos utilizar el mismo código en varias funciones sin necesidad de escribirlo varias veces.

Veamos otro ejemplo:

```
def comprueba_tipo(funcion):
    def funcion_decorada(a, b):
        print (“Se va a llamar a la función”, funcion.__name__)
        if isinstance(a, str):
            print (“El parámetro a es de tipo texto, se convierte a entero”)
            a = int(a)
        if isinstance(b, str):
            print (“El parámetro b es de tipo texto, se convierte a entero”)
```

```
            b = int(b)
        c = funcion(a, b)
        print (“Ha terminado la función”, funcion.__name__, “con resultado”, c)
        return c
    return funcion_decorada

@comprueba_tipo
def suma(a, b):
    print (“Sumando…”)
    return a + b

suma(‘5’, 8)

@comprueba_tipo
def resta(a, b):
    print (“Restando…”)
    return a - b

resta(5, ‘8’)
```

```
Se va a llamar a la función suma
El parámetro a es de tipo texto, se convierte a entero
Sumando…
Ha terminado la función suma con resultado 13

Se va a llamar a la función resta
El parámetro b es de tipo texto, se convierte a entero
Restando…
Ha terminado la función resta con resultado -3
```

En este ejemplo hemos añadido una comprobación sobre el tipo de dato de los parámetros de entrada de la función, de modo que si es un tipo texto se convierta a entero. Definimos una sola vez la comprobación de tipos y lo aplicamos a las dos funciones gracias a la función decoradora.

En caso de que así lo quisiésemos podríamos aplicar a una función más de un decorador. En el siguiente ejemplo, vamos a ver como creamos dos decoradores, uno que añada los textos antes y después de la función y otro que compruebe los tipos para luego aplicar los dos decoradores a la función. Es importante señalar que cuando apliquemos más de un decorador a una función, se hará de abajo a arriba, es decir se aplicará primero el decorador que más cerca esté de la función.

```
def decora_texto(funcion):
    def funcion_decorada(a, b):
        print (“Se va a llamar a la función”, funcion.__name__)
        c = funcion(a, b)
        print (“Ha terminado la función”, funcion.__name__, “con resultado”, c)
        return c
    return funcion_decorada

def comprueba_tipo(funcion):
```

```
    def funcion_decorada_tipo(a, b):
        if isinstance(a, str):
            print ("El parámetro a es de tipo texto, se convierte a entero")
            a = int(a)
        if isinstance(b, str):
            print ("El parámetro b es de tipo texto, se convierte a entero")
            b = int(b)
        c = funcion(a, b)
        return c
    return funcion_decorada_tipo

@comprueba_tipo
@decora_texto
def suma(a, b):
    print ("Sumando…")
    return a + b

suma('5', 8)
```

```
El parámetro a es de tipo texto, se convierte a entero
Se va a llamar a la función suma
Sumando…
Ha terminado la función suma con resultado 13
```

Como vemos, aplica los textos de información de llamada a la función suma y sobre esa función resultado aplica la conversión.

Aplicando los decoradores al revés obtendríamos:

```
@decora_texto
@comprueba_tipo
def suma(a, b):
    print ("Sumando…")
    return a + b

suma('5', 8)
```

```
Se va a llamar a la función funcion_decorada_tipo
El parámetro a es de tipo texto, se convierte a entero
Sumando…
Ha terminado la función funcion_decorada_tipo con resultado 13
```

Como vemos, al utilizarlo en este orden primero cambiamos el tipo y luego hacemos los textos de información, por lo que los textos nos informan de un nombre de función que no es adecuado. Por tanto, hemos de tener cuidado a la hora de utilizar los decoradores en el orden correcto para evitar este tipo de situaciones.

2 Bibliografía

Bahit, E. (s.f.). *Python para principiantes*. Obtenido de https://librosweb.es/libro/python/

Foundation, P. S. (2013). *Python.org*. Obtenido de https://www.python.org/

González Duque, R. (2011). *Python para todos.*

Suárez Lamadrid, A., & Suárez Jiménez, A. (Marzo de 2017). *Python para impacientes*. Obtenido de http://python-para-impacientes.blogspot.com.es/2017/03/el-modulo-time.html