Funciones generadoras

Por regla general, cuando queremos crear una lista de algún tipo, lo que hacemos es crear la lista vacía, y luego con un bucle varios elementos e ir añadiendolos a la lista si cumplen una condición:

```
[numero for numero in [0,1,2,3,4,5,6,7,8,9,10] if numero % 2 == 0 ]
```

```
[0, 2, 4, 6, 8, 10]
```

También vimos cómo era posible utilizar la función **range()** para generar dinámicamente la lista en la memoria, es decir, no teníamos que crearla en el propio código, sino que se interpretaba sobre la marcha:

```
[numero for numero in range(0,11) if numero % 2 == 0 ]
```

```
[0, 2, 4, 6, 8, 10]
```

La verdad es que **range()** es una especie de función generadora. Por regla general las funciones devolvuelven un valor con **return**, pero la preculiaridad de los generadores es que van *cediendo* valores sobre la marcha, en tiempo de ejecución.

La función generadora **range(0,11)**, empieza cediendo el **0**, luego se procesa el for comprobando si es par y lo añade a la lista, en la siguiente iteración se cede el **1**, se procesa el for se comprueba si es par, en la siguiente se cede el **2**, etc.

Con esto se logra ocupar el mínimo de espacio en la memoria y podemos generar listas de millones de elementos sin necesidad de almacenarlos previamente.

Veamos a ver cómo crear una función generadora de pares:

```
def pares(n):
    for numero in range(n+1):
        if numero % 2 == 0:
            yield numero

pares(10)
```

```
<generator object pares at 0x000002945F38BFC0>
```

Como vemos, en lugar de utilizar el **return**, la función generadora utiliza el **yield**, que significa ceder. Tomando un número busca todos los pares desde 0 hasta el número+1 sirviéndonos de un range().

Sin embargo, fijaros que al imprimir el resultado, éste nos devuelve un objeto de tipo generador.

De la misma forma que recorremos un **range()** podemos utilizar el bucle for para recorrer todos los elementos que devuelve el generador:

```
for numero in pares(10):
    print(numero)
```

```
0
2
4
6
8
10
```

Utilizando comprensión de listas también podemos crear una lista al vuelo:

```
[numero for numero in pares(10)]
```

```
[0, 2, 4, 6, 8, 10]
```

Sin embargo el gran potencial de los generadores no es simplemente crear listas, de hecho como ya hemos visto, el propio resultado no es una lista en sí mismo, sino una secuencia iterable con un montón de características únicas.

Iteradores

Por tanto las funciones generadoras devuelven un objeto que suporta un protocolo de iteración. ¿Qué nos permite hacer? Pues evidentemente controlar el proceso de generación. Teniendo en cuenta que cada vez que la función generadora cede un elemento, queda suspendida y se retoma el control hasta que se le pide generar el siguiente valor.

Así que vamos a tomar nuestro ejemplo de pares desde otra perspectiva, como si fuera un iterador manual, así veremos exactamente a lo que me refiero:

```
pares = pares(3)
```

Bien, ahora tenemos un iterador de pares con todos los números pares entre el 0 y el 3. Vamos a conseguir el primer número par:

```
next(pares)
```

```
0
```

Como vemos la función integrada **next()** nos permite acceder al siguiente elemento de la secuencia. Pero no sólo eso, si volvemos a ejecutarla...

```
next(pares)
```

```
2
```

Ahora devuelve el segundo! ¿No os recuerdo esto al puntero de los ficheros? Cuando leíamos una línea, el puntero pasaba a la siguiente y así sucesivamente. Pues aquí igual.

¿Y qué pasaría si intentamos acceder al siguiente, aún sabiendo que entre el 0 y el 3 sólo tenemos los pares 0 y 2?

```
next(pares)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-34-68378216ba43> in <module>()
----> 1 next(pares)

StopIteration:
```

Pues que nos da un error porque se ha acabado la secuencia, así que tomad nota y capturad la excepción si váis a utilizarlas sin saber exactamente cuantos elementos os devolverá el generador.

Así que la pregunta que nos queda es ¿sólo es posible iterar secuencias generadas al vuelo? Vamos a probar con una lista:

```
lista = [1,2,3,4,5]
next(lista)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-38-28c22b67c419> in <module>()
    1 lista = [1,2,3,4,5]
----> 2 next(lista)
    3
    4 cadena = "Hola"
    5 next(cadena)

TypeError: 'list' object is not an iterator
```

¿Quizá con una cadena?

```
cadena = "Hola"
next(cadena)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-39-44ca9ed1903b> in <module>()
    1 cadena = "Hola"
----> 2 next(cadena)

TypeError: 'str' object is not an iterator
```

Pues no, no podemos iterar ninguna colección como si fuera una secuencia. Sin embargo, hay una función muy interesante que nos permite covertir las cadenas y algunas colecciones a iteradores, la función **iter()**:

```
lista = [1,2,3,4,5]
lista_iterable = iter(lista)
print( next(lista_iterable) )
print( next(lista_iterable) )
print( next(lista_iterable) )
print( next(lista_iterable) )
print( next(lista_iterable) )
```

```
1
2
3
4
5
```

```
cadena = "Hola"
cadena_iterable = iter(cadena)
print( next(cadena_iterable) )
print( next(cadena_iterable) )
print( next(cadena_iterable) )
print( next(cadena_iterable) )
```

```
H
o
l
a
```

Con esto hemos visto las bases, os sugiero probar por vuestra cuenta más colecciones a ver si encontráis alguna más que se pueda iterar.

Última edición: 6 de Octubre de 2018