Generadores

Los generados son funciones especiales que nos van devolviendo una secuencia de valores de a un elemento por cada vez que la llamamos. Además, los generadores son una forma sencilla y potente de iterador.

Los generadores sirven para crear iteradores pero de manera *lazy* (o perezosa). Ya que el iterador tiene que tener todos los elementos disponibles. En cambio, los generadores van generando el nuevo valor cada vez que se llama al método __next__(). Una característica de los generadores que se desprende de esta propiedad es que puedo tener generadores que generan infinitos elementos. Por ejemplo, un generador de los números naturales.

Los generadores se crean como una función, la diferencia es que en lugar de devolver un resultado con la sentencia *return* lo hace con la sentencia *yield.* La sentencia *yield* devuelve el valor, pero a diferencia de la sentencia return, deja en "pausa" la función hasta la próxima vez que es invocada.

Veamos un ejemplo de su creación:

```
# Un generador de los números desde el 0 al 9
def gen_10():
    for n in range(10):
        yield n
```

Para utilizarlo, primero hay que crear una instancia del generador y luego se obtienen los valores generados con la función __next__()[1]. Cuando se termina de generar todos los valores, el generador lanza una excepción del tipo StopIteration.

```
>>> a_gen_10 = gen_10()
>>> a_gen_10.__next__()
0
>>>  a_gen_10.__next__()
1
…
>>> a_gen_10.__next__()
9
>>> a_gen_10.__next__()
StopIteration
```

Notar que al igual que los iteradores, los generadores se pueden consumir con un ciclo **for**.

Veamos un ejemplo de un generador de los números naturales (que son infinitos):

```
def naturales():
   n = 1
   while True:
      yield n
      n = n + 1

naturales = naturales()
```

[1] Al igual que los iteradores, devuelve el siguiente valor con la función __next__(). Como vimos la diferencia radica en que el iterador ya tiene todos los valores disponibles y el generador los va generando uno a uno cuando se le pide.

Si dentro del generador hay una sentencia return, al ejecutar esta sentencia se lanza una excepción del tipo StopIteration, notificando de esta manera que ya se generaron todos los valores. Por ejemplo, si quiere crear un generador que me devuelva los primeros n números naturales:

```
def n_naturales(n):
    i = 1
    while True:
        if i > n:
            return
        yield i
        i = i + 1

# Genera los primeros 100 números naturales
naturales = naturales(100)

# Genera los primeros 150 números naturales
naturales = naturales(150)
```

A continuación se muestra una forma de escribir los generadores muy simple y compacta. Es parecida a las listas por comprensión solo que se escribe entre paréntesis:

```
# Lista por comprensión
>>> [p for p in range(10) if p % 2 == 0]
[0, 2, 4, 6, 8]

# Generador
(p for p in range(10) if p % 2 == 0)
<generator object <genexpr> at 0x10f1a3190>
```

Por último, veamos en qué casos es útil utilizar los generadores:

- Cuando trabajamos con estructuras infinitas.
- Cuando trabajamos con estructuras que ocupan mucha memoria, se puede reducir el espacio de memoria a reservar.
- Cuando la generación de los elementos es costosa, se puede retrasar su cálculo hasta el último momento.
- Cuando trabajamos con otros generadores, para no forzarlos a tener que calcularse todos los valores de los otros generadores.