07. Iteradores y Listas por Comprension

August 11, 2024

1 Iterables

Un iterador es un objeto que contiene una cantidad contable de valores.

Un iterador es un objeto que se puede iterar, lo que significa que se pueden recorrer todos los valores.

Técnicamente, en Python, un iterador es un objeto que implementa el protocolo `iterator`, que consta de los métodos `__iter__()` y `__next__()`.

Python tiene varios métodos nativos que procesan iterables:

- `sort`: ordena elementos en un iterable.
- `zip`: combina elementos de iterables.
- `enumerate`: empareja elementos en un iterable con posiciones relativas.
- `filter`: selecciona elementos para los que una función es verdadera.
- `reduce`: ejecuta pares de elementos en un iterable a través de una función.

1.1 Iterator vs Iterable

Las listas, tuplas, diccionarios y conjuntos son objetos iterables. Son contenedores iterables de los que se puede obtener un iterador.

Todos estos objetos tienen un método `iter()` que se utiliza para obtener un iterador:

Ejemplo:

[ ]:
```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

```
apple
banana
cherry
```

Incluso las cadenas String son objetos iterables, y pueden devolver un iterador.

[ ]:
```
mystr = "banana"
myit = iter(mystr)
```

```
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

```
b
a
n
a
n
a
```

1.2 Recorriendo un iterador

Se puede utilizar un ciclo for para iterar a través del objeto iterable.

Ejemplo: Iterar los valores de una tupla

[ ]:
```
mytuple = ("apple", "banana", "cherry")

for x in mytuple:
  print(x)
```

```
apple
banana
cherry
```

Ejemplo: Iterar los caracteres de una cadena

[ ]:
```
 mystr = "banana"

for x in mystr:
  print(x)
```

```
b
a
n
a
n
a
```

El ciclo `for` en realidad crea un objeto iterador y ejecuta el método `next()` para cada bucle.

1.3 Crear un iterador

Para crear un objeto/clase como iterador, debe implementar los métodos `__iter__()` y `__next__()` en su objeto.

Recordando la creación de clases, todas ellas tienen una función llamada `__init__()`, que le permite realizar algunas inicializaciones cuando se crea el objeto.

El método `__iter__()` actúa de manera similar, puede realizar operaciones (inicializar, etc.), pero siempre debe devolver el objeto iterador en sí.

El método `__next__()` también le permite realizar operaciones y debe devolver el siguiente elemento de la secuencia.

Ejemplo: Crear un iterador que devuelve números, comenzando con 1 y cada secuencia se incrementa en uno.

[ ]:
```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
1
2
3
4
5
```

1.4 StopIteration

El código anterior iteraría indefinidamente para un número indefinido de llamadas `next()` o si fuese utilizado en un ciclo `for`. Para prevenir esto se puede utilizar la sentencia `StopIteration`.

En la definición del método `__next__()` se puede definir la condición de terminación que lanzará un error si se sobrepasa la cantidad de iteraciones definidas.

Ejemplo. Detener la iteración luego de 20 repeticiones.

[ ]:
```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
```

```
  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

1.5 Iterador `range`

Ya hemos utilizado con anterioridad el método **`range`**, pero ahora ya podemos especificar que el método devuelve un iterador que genera números bajo demanda, en lugar de construir la lista resultante en la memoria. Se puede forzar a que el iterador sea una lista de números mediante el método `list`.

[ ]:
```
R = range(10)
R
```

[ ]:
```
range(0, 10)
```

[ ]:
```
I = iter(R)
print(next(I))
print(next(I))
print(I.__next__())
```

```
0
1
2
```

[ ]:
```
I = iter(R)
for x in R:
    print(next(I))
```

```
0
1
2
3
4
5
6
7
8
9
```

[ ]:
```
list(R)
```

[ ]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.6 Iteradores map, zip y filter

Al igual que `range`, los métodos `map`, `zip` y `filter` son iteradores en lugar de producir una lista con los resultados. Pero a diferencia de `range`, estos métodos métodos son sus propios iteradores. Es decir, después de recorrer sus resultados una vez, se agotan. En otras palabras, no puede tener múltiples iteradores en sus resultados que mantengan diferentes posiciones en esos resultados.

En el siguiente ejemplo, `map` devuelve un iterador y no una lista.

[ ]:
```
M = map(abs, (-1, 0, 1))
M
```

[ ]:
```
<map at 0x7fa89c5382b0>
```

[ ]:
```
next(M)
```

[ ]:
```
1
```

[ ]:
```
M = map(abs, (-1, 0, 1))

for m in M:
```

```
    print(m)
```

```
1
0
1
```

[ ]:
```
M = map(abs, (-1, 0, 1))
list(M)
```

[ ]:
```
[1, 0, 1]
```

La función `zip` devuelve un iterador que funciona de la misma forma.

[ ]:
```
Z = zip((1,2,3), (10,20,30))
Z
```

[ ]:
```
<zip at 0x7fa89c1c7500>
```

[ ]:
```
list(Z)
```

[ ]:
```
[(1, 10), (2, 20), (3, 30)]
```

[ ]:
```
Z = zip((1,2,3), (10,20,30))

for pair in Z:
    print(pair)
```

```
(1, 10)
(2, 20)
(3, 30)
```

[ ]:
```
Z = zip((1,2,3), (10,20,30))
next(Z)
```

[ ]:
```
(1, 10)
```

[ ]:
```
next(Z)
```

[ ]:
```
(2, 20)
```

El método nativo filter devuelve elementos en un iterable para los cuales la función pasada como argumento devolvió `True`. El método `filter` acepta un iterable para procesarlo y devuelve un iterable para los resultados generados.

[ ]:
```
filter(bool, ['spam', '', 'ni'])
```

[ ]:
```
<filter at 0x7fa89c7d9c90>
```

[ ]:
```
list(filter(bool, ['spam', '', 'ni']))
```

[ ]:
```
['spam', 'ni']
```

1.7 Iteradores múltiples y sencillos

El objeto `range` difiere de los demás métodos mencionados, ya que soporta len e indexación pero no es su propio iterador, (se crea uno con `iter` cuando se itera manualmente) y admite múltiples iteradores sobre su resultado que recuerdan sus posiciones de manera independiente.

[ ]:
```
R = range(3)
next(R)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[58], line 2
      1 R = range(3)
----> 2 next(R)

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

[ ]:
```
I1 = iter(R)
next(I1)
```

[ ]:
```
0
```

[ ]:
```
next(I1)
```

[ ]:
```
1
```

[ ]:
```
I2 = iter(R)
next(I2)
```

[ ]:
```
0
```

[ ]:
```
next(I1)
```

[ ]:
```
2
```

Por el contrario, `zip`, `map` y `filter` no soportan múltiples iteradores activos en el mismo resultado.

[ ]:
```
Z = zip((1, 2, 3), (10, 20, 30))
I1 = iter(Z)
I2 = iter(Z)
```

[ ]:
```
next(I1)
```

[ ]:
```
(1, 10)
```

[ ]:
```
next(I2)
```

[ ]: (2, 20)

[ ]:
```
next(I1)
```

[ ]: (3, 30)

[ ]:
```
next(I2)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[69], line 1
----> 1 next(I2)

StopIteration:
```

Para el caso de `map`:

[ ]:
```
M = map(abs, (-1, 0, 1))
I1 = iter(M)
I2 = iter(M)
print(next(I1), next(I1), next(I1))
print(next(I2))
```

```
1 0 1
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[74], line 5
      3 I2 = iter(M)
      4 print(next(I1), next(I1), next(I1))
----> 5 print(next(I2))

StopIteration:
```

Y en el caso de `range` para un código similar:

[ ]:
```
R = range(3)
I1 = iter(R)
I2 = iter(R)
print(next(I1), next(I1), next(I1))
print(next(I2))
```

```
0 1 2
0
```

1.8 Iteradores de vista de Diccionario

En Python las claves, valores y métodos de elementos del diccionario devuelven objetos iterables `view` que generan elementos de resultado de a uno por vez, en lugar de producir listas de resultados de una sola vez en la memoria. Los elementos `view` mantienen el mismo orden físico que el del diccionario y reflejan los cambios realizados en el diccionario subyacente.

[ ]:
```
D = dict(a=1, b=2, c=3)
D
```

[ ]:
```
{'a': 1, 'b': 2, 'c': 3}
```

[ ]:
```
K = D.keys()
K
```

[ ]:
```
dict_keys(['a', 'b', 'c'])
```

[ ]:
```
next(K)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[82], line 1
----> 1 next(K)

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

[ ]:
```
I = iter(K)
next(I)
```

[ ]:
```
'a'
```

[ ]:
```
next(I)
```

[ ]:
```
'b'
```

[ ]:
```
for k in D.keys():
    print(k, end=' ')
```

```
a b c
```

Al igual que con todos los iteradores, siempre puedes forzar una vista de diccionario para que construya una lista real. Sin embargo, esto no suele ser necesario, excepto para mostrar resultados de forma interactiva o para aplicar operaciones de lista como la indexación:

[ ]:
```
K = D.keys()
list(K)
```

[ ]:
```
['a', 'b', 'c']
```

[ ]:
```
V = D.values()
list(V)
```

[ ]:
```
[1, 2, 3]
```

[ ]:
```
D.items()
```

[ ]:
```
dict_items([('a', 1), ('b', 2), ('c', 3)])
```

[ ]:
```
list(D.items())
```

[ ]:
```
[('a', 1), ('b', 2), ('c', 3)]
```

[ ]:
```
for (k, v) in D.items():
    print(f'{k} -> {v}')
```

```
a -> 1
b -> 2
c -> 3
```

Además, los diccionarios también tienen iteradores, que devuelven claves sucesivas, por lo que no suele ser necesario llamar a las claves directamente en este contexto.

[ ]:
```
D
```

[ ]:
```
{'a': 1, 'b': 2, 'c': 3}
```

[ ]:
```
I = iter(D)
next(I)
```

[ ]:
```
'a'
```

[ ]:
```
next(I)
```

[ ]:
```
'b'
```

No se necesita llamar a la función `keys()` para iterar el diccionario, pero `keys` también es un iterador.

[ ]:
```
for key in D:
    print(key)
```

```
a
b
c
```

Por último, recuerde nuevamente que, dado que `keys` ya no devuelve una lista, el patrón de codificación tradicional para escanear un diccionario por claves ordenadas no funcionará. En su lugar, convierta primero las vistas de `keys` con una llamada de lista, o utilice la llamada `sorted` en una vista de `keys` o en el diccionario mismo, de la siguiente manera:

[ ]:
```
D = dict(a=1, c=3, b=2)
D
```

[ ]:
```
{'a': 1, 'c': 3, 'b': 2}
```

[ ]:
```
for k in sorted(D.keys()):
    print(f'{k} -> {D[k]}')
```

```
a -> 1
b -> 2
c -> 3
```

O mejor aún, una buena práctica para el ordenamiento de las llaves:

[ ]:
```
D
```

[ ]:
```
{'a': 1, 'c': 3, 'b': 2}
```

[ ]:
```
for k in sorted(D):
    print(f'{k} -> {D[k]}')
```

```
a -> 1
b -> 2
c -> 3
```

2 Listas por Comprensión

En el capítulo anterior, estudiamos herramientas de programación funcional como `map` y `filter`, que mapean operaciones sobre secuencias y recopilan resultados. Debido a que esta es una tarea tan común en la codificación, Python finalmente generó una nueva funcionalidad: **listas por comprensión** (o la comprensión de listas), que es incluso más flexible que las herramientas anteriormente mencionadas.

En resumen, las listas por comprensión aplican una expresión arbitraria a los elementos de un iterable, en lugar de aplicar una función. Como tal, pueden ser herramientas más generales.

2.1 Listas por Comprensión Vs. map

Trabajemos con un ejemplo que demuestra los conceptos básicos. La función nativa de Python `ord` devuelve el código ASCII de un caracter (la función nativa `chr` hace lo opuesto, devuelve el caracter de un código ASCII):

[ ]:
```
ord('H')
```

[ ]:
```
72
```

Ahora supongamos que se desea tener una lista con el código ASCII de cada uno de los caracteres de una cadena. Una forma de realizarlo sería a través de un ciclo `for` e ir añadiendo cada valor en una lista:

[ ]:
```
res = []
for x in 'Hola':
    res.append(ord(x))
res
```

[ ]:
```
[72, 111, 108, 97]
```

Ahora empleando el método `map` para realizar la misma tarea:

[ ]:
```
res = list(map(ord, "Hola"))
res
```

[ ]:
```
[72, 111, 108, 97]
```

Esta operación requirió menos código utilizando `map`.

Sin embargo, es posible obtener el mismo resultado empleando ahora listas por comprensión.

Mientras que `map` asocia (mapea) una *función* sobre una *secuencia*, las listas por comprensión asocian una *expresión* sobre una *secuencia*.

[ ]:
```
res = [ord(x) for x in "Hola"]
res
```

[ ]:
```
[72, 111, 108, 97]
```

Las listas por comprensión recopilan los resultados de aplicar una expresión arbitraria a una secuencia de valores y los devuelven en una nueva lista. Sintácticamente, las listas por comprensión se encierran entre corchetes para enfatizar que construye una lista.

En su forma simple, dentro de los corchetes se codifica una expresión que nombra una variable seguida de un de bucle `for` que nombra la misma variable. Luego, Python recopila los resultados de la expresión para cada iteración del bucle implícito.

El resultado obtenido por el último código es el mismo con respecto a sus equivalentes con `for` y `map`. Sin embargo, las listas por comprensión se vuelven más convenientes cuando deseamos aplicar una expresión arbitraria a una secuencia:

[ ]:
```
[x**2 for x in range(10)]
```

[ ]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Esta lista recopila el cuadrado de los números enteros de 0 hasta 9. Si deseamos una versión del código empleando `map` se necesita de una función pequeña que haga el cuadrado de un número. Esta es una oportunidad de utilizar a su vez una función `lambda`, dado que no se utilizará para otro propósito adicional.

[ ]:
```
list(map(lambda x:x**2, range(10)))
```

[ ]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A pesar de que esta línea realiza el mismo trabajo, requiere un poco más de código y resulta también un poco más difícil de leer debido a la función `lambda`. Para expresiones más avanzadas, las listas por comprensión típicamente requieren menos código.

2.2 Añadiendo pruebas y ciclos anidados

Las listas por comprensión son incluso más generales que lo que se ha mostrado hasta ahora. Por ejemplo, se puede codificar una cláusula `if` después del `for` para agregar lógica de selección.

Las listas por comprensión con cláusulas `if` pueden considerarse análogas al filtro incorporado que se analizó previamente: omiten elementos de secuencia para los que la cláusula `if` no es verdadera.

Para demostrarlo, aquí se muestran dos códigos que recogen números pares del 0 al 4; al igual que con la alternativa `map` a la lista por comprensión, la versión con `filter` requiere una pequeña función `lambda` para la expresión de prueba. A modo de comparación, aquí también se muestra el bucle `for` equivalente:

[ ]:
```
# Lista por comprensión
[x for x in range(5) if x %2 ==0]
```

[ ]:
```
[0, 2, 4]
```

[ ]:
```
# filter y lambda
list(filter(lambda x:x%2==0, range(5)))
```

[ ]:
```
[0, 2, 4]
```

[ ]:
```
# ciclo for
res = []
for x in range(5):
    if x%2==0:
        res.append(x)
res
```

[ ]:
```
[0, 2, 4]
```

En todos los casos se utiliza el operador módulo (residuo de la división) para identificar números pares. El ćodigo que emplea `filter` no es considerablemente más largo que la versió de listas por comprensión. Sin embargo, podemos combinar una condicional `if` y una expresión arbitraria en nuestra lista por comprensión para darle el efecto de `filter` y `map`, en una sola expresión:

[ ]:
```
[x**2 for x in range(10) if x%2==0]
```

[ ]:
```
[0, 4, 16, 36, 64]
```

Esta vez, recopilamos los cuadrados de los números pares del 0 al 9: el ciclo `for` omite los números para los cuales el condicional `if` a la derecha es falso, y la expresión a la izquierda calcula los cuadrados.

La versión del código equivalente con `map` requiere más trabajo: tenemos que combinar selecciones `filter` con iteración `map`, lo que daría como resultado una expresión notablemente más compleja:

[ ]:
```
list(map(lambda x:x**2, filter(lambda x:x%2==0, range(10))))
```

[ ]:
```
[0, 4, 16, 36, 64]
```

Las listas por comprensión son aún más generales. Se puede codificar cualquier cantidad de ciclos `for` anidados en una lista por comprensión, y cada una puede tener una condicional `if` asociada opcional.

La estructura general de las listas por comprensión es la siguiente:

```
[
    expression for target1 in iterable1 [if condition1]
                for target2 in iterable2 [if condition2]
                ...
                for targetN in iterableN [if conditionN]
]
```

Cuando los ciclos `for` están anidados dentro de una lista por comprensión, funcionan como ciclos `for` anidados. Por ejemplo:

[ ]:
```
res = [x+y for x in [0, 1, 2] for y in [100, 200, 300]]
res
```

[ ]:
```
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

La versión equivalente del código:

[ ]:
```
res = []
for x in [0, 1, 2]:
    for y in [100, 200, 300]:
        res.append(x+y)

res
```

[ ]:
```
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Aunque las listas por comprensión construyen listas, recuerde que pueden iterar sobre cualquier secuencia u otro tipo iterable. El siguiente código recorre una cadena en lugar de una lista de números y, por lo tanto recopila resultados de una concatenación.

[ ]:
```
res = [x+y for x in 'hola' for y in 'HOLA']
res
```

[ ]:
```
['hH',
 'hO',
 'hL',
 'hA',
 'oH',
 'oO',
 'oL',
 'oA',
```

```
 'lH',
 'lO',
 'lL',
 'lA',
 'aH',
 'aO',
 'aL',
 'aA']
```

Por último, se muestra a contuación un código que muestra el efecto de añadir condicionales `if` a ciclos `for` anidados. El código debe formar pares ordenados donde su primer elemento sea par y su segundo elemento impar, empleando valores desde 0 hasta 4.

[ ]:
```
[(x,y) for x in range(5) if x%2==0 for y in range(5) if y%2==1]
```

[ ]:
```
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

La versión equivalente empleando ciclos `for`:

[ ]:
```
res = []
for x in range(5):
    if x%2==0:
        for y in range(5):
            if y%2==1:
                res.append((x,y))

res
```

[ ]:
```
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

El código equivalente empleando `map` y `filter` es extremadamente complejo y profundamente anidado, así que ni siquiera intentaré mostrarlo aquí.

2.3 Listas por comprensión y matrices

Una forma básica de codificar matrices en Python es con listas anidadas. A continuación, se definen dos matrices de 3×3 como listas de listas anidadas:

[ ]:
```
M = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
N = [
    [2, 2, 2],
    [3, 3, 3],
    [4, 4, 4]
]
```

Dada esta estructura, se puede ubicar por índice los renglones y columnas, utilizando operaciones de índice ordinarias:

[ ]:
```
M[1]
```

[ ]:
```
[4, 5, 6]
```

[ ]:
```
M[1][2]
```

[ ]:
```
6
```

Sin embargo, las listas por comprensión son herramientas poderosas para procesar dichas estructuras, ya que escanean automáticamente las filas y columnas por nosotros. Por ejemplo, aunque esta estructura almacena la matriz por filas, para recolectar la segunda columna podemos simplemente iterar a través de las filas y extraer la columna deseada, o iterar a través de las posiciones en las filas e indexar a medida que avanzamos

[ ]:
```
[row[1] for row in M]
```

[ ]:
```
[2, 5, 8]
```

[ ]:
```
[M[row][1] for row in (0, 1, 2)]
```

[ ]:
```
[2, 5, 8]
```

Dadas las posiciones, también podemos realizar tareas fácilmente, como extraer una diagonal.

La siguiente expresión usa `range` para generar la lista de elementos que se encuentran en el mismo índice para fila y columna, esto es, `M[0][0]`, `M[1][1]`, y así sucesivamente (suponiendo que la matriz tiene la misma cantidad de filas y columnas):

[ ]:
```
[M[i][i] for i in range(len(M))]
```

[ ]:
```
[1, 5, 9]
```

También podemos usar listas por comprensión para combinar varias matrices. A continuación, primero se crea una lista plana que contiene el resultado de multiplicar las matrices por pares y luego se crea una estructura de lista anidada que tiene los mismos valores anidando listas por comprensión:

[ ]:
```
[M[row][col] * N[row][col] for row in range(len(M)) for col in range(len(N))]
```

[ ]:
```
[2, 4, 6, 12, 15, 18, 28, 32, 36]
```

Y si anidamos la lista, para crear una lista bidimiensional:

[ ]:
```
[[M[row][col] * N[row][col] for col in range(len(M))] for row in range(len(N))]
```

[ ]:
```
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Esta última expresión funciona dado que la iteración del renglón ocurre en el ciclo exterior del anidamiento de ciclos `for`. Es decir, para cada renglón se iteran todas las columnas y se almacena en la lista resultante.

Un código equivalente empleando ciclos `for` es el siguiente:

[ ]:
```
res = []
for row in range(len(M)):
    temp = []
    for col in range(len(N)):
        temp.append(M[row][col] * N[row][col])
    res.append(temp)

res
```

[ ]:
```
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Comparando ambas versiones, el código que emplea listas por comprensión sólo requiere una línea. Además de acuerdo a la documentación, se ejecuta más rápido para matrices más grandes.

2.4 Regresando a los iteradores: Generadores

Python proporciona herramientas que producen resultados solo cuando son necesarios, en lugar de hacerlo todos a la vez. En particular, dos construcciones del lenguaje retrasan la creación de resultados siempre que sea posible:

- *Funciones generador*. Las funciones de generador se codifican como declaraciones `def` normales, pero utilizan declaraciones `yield` para devolver los resultados de a uno por vez, suspendiendo y reanudando su estado entre cada uno.
- *Expresiones generador*. Las expresiones de generador son similares a las comprensiones de listas, pero devuelven un objeto que produce resultados a pedido en lugar de construir una lista de resultados.

2.4.1 Funciones generador: yield vs. return

Suspensión de estado A diferencia de las funciones normales que devuelven un valor y salen, las funciones generadoras suspenden y reanudan automáticamente su ejecución y estado alrededor del punto de generación del valor. Por eso, suelen ser una alternativa útil tanto para calcular una serie completa de valores por adelantado como para guardar y restaurar manualmente el estado en las clases. Debido a que el estado que las funciones generadoras conservan cuando se suspenden incluye todo su ámbito local, sus variables locales conservan información y la ponen a disposición cuando se reanudan las funciones.

La principal diferencia entre generador y funciones normales, es que el generador *genera* un valor, en lugar de devolverlo: la sentencia `yield` suspende la función y envía un valor de vuelta al invocador, pero conserva su estado para permitir que la función reanude desde allí. Por ende, cuando la función se reanuda continúa la ejecución en ese estado. Esto permite que el código genere una serie de valores a lo largo del tiempo, conforme se requieran, en lugar de calcularlos todos a la vez.

El protocolo de iteración Para verdaderamente comprender las funciones generadoras, es necesario saber que están estrechamente relacionadas con la noción del protocolo de iteración en Python. Como hemos visto, los objetos iterables definen un método `__next__`, que devuelve el siguiente elemento en la iteración o genera la excepción especial `StopIteration` para finalizar la iteración. El iterador de un objeto se obtiene con la función nativa `iter`.

El ciclo `for` de Python y todos los demás contextos de iteración, utilizan el protocolo de iteración para recorrer una secuencia o un generador de valores, si es que el protocolo está soportado. Si no, la iteración recurre a la indexación repetida de secuencias.

Para implementar este protocolo, las funciones que contienen `yield` se compilan como generadores. Cuando se les llama, devuelven un objeto generador que soporta la interfaz de iteración con un método creado automáticamente llamado `__next__` para reanudar la ejecución.

Las funciones generadoras también pueden tener un `return` que, además de definir el final del bloque def, terminan la generación de valores, técnicamente generando una excepción `StopIteration` después de cualquier salida de la función.

Desde la perspectiva de quien llamó a la función, el método `__next__` reanuda la función y se ejecuta hasta que se devuelva el siguiente resultado de `yield` o bien se genere un `StopIteration`.

Ejemplo: El siguiente código define una función generadora que se puede utilizar para generar el cuadrado de una serie de números.

[ ]:
```
def gensquares(N):
    for i in range(N):
        yield i**2
```

[ ]:
```
x = gensquares(5)
x
```

[ ]:
```
<generator object gensquares at 0x7fa38dbb7d30>
```

[ ]:
```
next(x)
```

[ ]:
```
4
```

[ ]:
```
next(x)
```

[ ]:
```
9
```

Ahora, incrustando la función generadora en un ciclo:

[ ]:
```
for i in gensquares(10):
    print(i)
```

```
0
1
4
9
16
25
```

```
36
49
64
81
```

Ejemplo: Generador de Números Pares

[ ]:
```python
def generador_pares():
    n = 0
    while True:
        yield n
        n += 2

# Ejemplo de uso
pares = generador_pares()
for _ in range(20):
    print(next(pares))  # Output: 0, 2, 4, 6, 8
```

```
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
```

Ejemplo: Generador de Fibonacci

[ ]:
```python
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Ejemplo de uso
fib = fibonacci()
```

```
for _ in range(10):
    print(next(fib))  # Output: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

```
0
1
1
2
3
5
8
13
21
34
```

Ejemplo: Generador de una Secuencia con un Rango Dado

[ ]:
```
def generador_rango(inicio, fin):
    while inicio < fin:
        yield inicio
        inicio += 1

# Ejemplo de uso
for numero in generador_rango(1, 5):
    print(numero)  # Output: 1, 2, 3, 4
```

```
1
2
3
4
```

Ejemplo: Generador que Filtra Números Pares

[ ]:
```
def filtrar_pares(lista):
    for numero in lista:
        if numero % 2 == 0:
            yield numero

# Ejemplo de uso
lista = [1, 2, 3, 4, 5, 6, 7, 8]
pares = filtrar_pares(lista)
for par in pares:
    print(par)  # Output: 2, 4, 6, 8
```

```
2
4
6
8
```

Ejemplo: Generador de aproximaciones al número π

$$\pi = 4 \times \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

[ ]:
```
iter = int(input("Introduzca el número de iteraciones: "))

pi = 0
for n in range(iter+1):
    pi += 4*(-1)**n / (2*n+1)

print(f'pi({iter}) = {pi}')
```

```
pi(100000) = 3.1416026534897203
```

[ ]:
```
def aproximacionPi():
    pi, n = 0, 0
    while True:
        yield n, pi
        pi += 4*(-1)**n / (2*n+1)
        n += 1

x = aproximacionPi()
```

[ ]:
```
next(x)
```

[ ]:
```
(1, 4.0)
```

[ ]:
```
N = int(input("Introduzca el número de iteraciones: "))
p = aproximacionPi()

for i in range(N):
    print(next(p))
```

```
(0, 0)
(1, 4.0)
(2, 2.666666666666667)
(3, 3.466666666666667)
(4, 2.8952380952380956)
```

2.5 Expresiones generadoras: Los iteradores se aproximan a las comprensiones

En todas las versiones recientes de Python, la noción de *iteradores* y *listas por comprensión* se combinan en una nueva característica del lenguaje: *expresiones generadoras*. En térmios de la sintáxis, las expresiones generadoras son como las listas por comprensión normales, pero se incluyen entre paréntesis en lugar de corchetes:

[ ]:
```
[x**2 for x in range(5)]
```

[ ]:
```
[0, 1, 4, 9, 16]
```

[ ]:
```
(x**2 for x in range(5))
```

[ ]:
```
<generator object <genexpr> at 0x7fa0ca13e4d0>
```

De hecho, al menos en términos de su funcionalidad, codificar una lista por comprensión es esencialmente lo mismo que envolver una expresión de generador en una llamada incorporada de lista para obligarla a producir todos sus resultados en una lista a la vez, esto es:

[ ]:
```
list(x**2 for x in range(5))
```

[ ]:
```
[0, 1, 4, 9, 16]
```

A pesar de este hecho, sin embargo, las expresiones generadoras son muy diferentes: devuelven un *objeto generador* que soporta el protocolo de iteración, que permite emplear *yield* para obtener el siguiente resultado conforme se pida.

[ ]:
```
G = (x**2 for x in range(5))
```

[ ]:
```
next(G)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[7], line 1
----> 1 next(G)

StopIteration:
```

Normalmente no vemos la sentencia *next* en una expresión generadora como esta, debido a que los ciclos *for* la lanzan automáticamente y por detrás:

[ ]:
```
for num in (x**2 for x in range(5)):
    print(num)
```

```
0
1
4
9
16
```

Tal como lo hemos visto, este contexto de iteración lo hace de esta forma, ésto incluye a las funciones nativas `sum`, `map` y `sorted`, así como las listas por comprensión, entre muchos otros.

[ ]:
```
sum(x**2 for x in range(5))
```

[ ]:
```
30
```

[ ]:
```
sorted(x**2 for x in range(5))
```

[ ]:
```
[0, 1, 4, 9, 16]
```

[ ]:
```
sum(map(lambda x:x**2, range(5)))
```

[ ]:
```
30
```

[ ]:
```
from functools import reduce

reduce(lambda acum, x : acum + x**2, range(5))
```

[ ]:
```
30
```

2.6 Generación de valores con tipos y clases nativos

Python permite varias formas de generar sus tipos avanzados de datos, por ejemplo, para generar un diccionario:

[ ]:
```
D = {'a':1, 'b':2, 'c':3}
x = iter(D)
```

[ ]:
```
next(x)
```

[ ]:
```
'c'
```

El diccionario puede ser iterado manualmente o con las herramientas de iteración nativas: for, map y listas por comprensión.

[ ]:
```
for key in D:
    print(key, D[key])
```

```
a 1
b 2
c 3
```

2.7 Recapitulando

[ ]:
```
# Listas por comprensión
[x*x for x in range(10)]
```

[ ]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

[ ]:
```
# Expresión generadora
(x*x for x in range(10))
```

[ ]:
```
<generator object <genexpr> at 0x7fce1b5df6b0>
```

[ ]:
```
# Conjunto por comprensión
{x*x for x in range(10)}
```

[ ]:
```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

[ ]:
```
# Diccionario por comprensión
{x:x*x for x in range(10)}
```

[ ]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

2.7.1 Diccionarios y conjuntos por comprensión

Conjunto por comprensión:

[ ]:
```
{x*x for x in range(10)}
```

[ ]:
```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Conjunto por generación y tipo:

[ ]:
```
set(x*x for x in range(10))
```

[ ]:
```
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Diccionario por comprensión:

[ ]:
```
{x:x*x for x in range(10)}
```

[ ]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Diccionario por generación y tipo:

[ ]:
```
dict((x, x*x) for x in range(10))
```

[ ]:
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

3 Referencias

- Lutz M., Learning Python, O'Reilly. 2009