T11_IteradoresGeneradores

January 29, 2021

1 Python de cero a experto

Autor: Luis Miguel de la Cruz Salas

1.1 Pythonico es más bonito: Pensando como pythonista (intermedio)

1.1.1 Iteradores y Generadores

iterables (*iterable*) e **iteradores** (*iterator*)

- La mayoría de los objetos contenedores se pueden recorrer usando un ciclo **for ... in ...** .
- Estos contenedores se conocen como iterables (objetos iterables, secuencias iterables, contenedores iterables, conjunto iterable, …).

Por ejemplo:

[1]:
```
mi_cadena = "12345"

# Iterating over a String
print("\nString Iteration: ", end='')
for char in mi_cadena:
    print(char, end=' ')
```

```
String Iteration: 1 2 3 4 5
```

[2]:
```
mi_lista = [1,2,3,4,5]

# Iterating over a list
print('\nList Iteration: ', end='')
for element in mi_lista:
    print(element, end=' ')
```

```
List Iteration: 1 2 3 4 5
```

[3]:
```
mi_tupla = (1,2,3,4,5)

# Iterating over a tuple (immutable)
print("\nTuple Iteration: ", end='')
for element in mi_tupla:
    print(element, end=' ')
```

```
Tuple Iteration: 1 2 3 4 5
```

[4]:
```
mi_dict = {'uno':1, 'dos':2, 'tres':3, 'cuatro':4, 'cinco':5}

# Iterating over dictionary
print("\nDictionary Iteration: ", end='')
for key in mi_dict:
    print(key, end=' ')
```

```
Dictionary Iteration: uno dos tres cuatro cinco
```

[5]:
```
mi_archivo = open("mi_archivo.txt")

# Iterating over file
print("\nFile Iteration: ")
for line in mi_archivo:
    print(line, end = '')
```

```
File Iteration:
1
2
3
```

Datos importantes: - Este es un estilo claro y conveniente que impregna el universo de Python. - La instrucción **for** llama a la función **iter()** que está definida dentro del objeto **contenedor**. - La función **iter()** regresa como resultado un objeto **iterador** que define el método **___next___()** el cual puede acceder a los elementos del objeto contenedor, uno a la vez. - Cuando no hay más elementos, **___next___()** lanza una excepción de tipo **StopIteration** que le dice al ciclo **for** que debe terminar. - Se puede ejecutar al método **___next___()** usando la función de biblioteca **next()**.

Por ejemplo:

[6]:
```
contenedor = 'xyz'
iterador = iter(contenedor)
print(type(iterador))
next(iterador)
next(iterador)
next(iterador)
```

```
<class 'str_iterator'>
```

[6]:
```
'z'
```

[7]:
```
next(iterador)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-7-2389250a88e0> in <module>
----> 1 next(iterador)

StopIteration:
```

Iterable con *list comprehension*

[8]:
```
Icuadrados = [x*x for x in range(3)]

for i in Icuadrados:
    print(i, end=' ')

type(Icuadrados)
```

```
0 1 4
```

[8]:
```
list
```

[9]:
```
for i in Icuadrados:
    print(i, end=' ')
```

```
0 1 4
```

Estos iterables son manejables y prácticos debido a que se pueden leer tanto como se desee, pero se almacenan todos los valores en memoria y esto no siempre es coveniente, sobre todo cuando se tienen muchos valores.

Generadores

- Los objetos **generadores** son iteradores.
- Pero solo se puede iterar sobre ellos una vez. Esto es porque los generadores no almacenan todos los valores en memoria, ellos generan los valores al vuelo.

Por ejemplo:

[10]:
```
cuadradosG = (x*x for x in range(3))

for i in cuadradosG:
    print(i, end=' ')
```

```
type(cuadradosG)
```

```
0 1 4
```

[10]:
```
generator
```

[11]:
```
for i in cuadradosG:    # Este ciclo no imprimirá nada por que
    print(i, end=' ')   # el generador ya se usó antes
```

Un generador solo se puede usar una vez, pues va calculando sus valores uno por uno e inmediatamente los va olvidando. En el ejemplo anterior tenemos:

- genera el 0, es usado y lo olvida
- genera el 1, es usado y lo olvida
- genera el 4, es usado y lo olvida

Yield Descripción.

- Es una palabra clave que suspende la ejecución de una función y envía un valor de regreso a quien la ejecuta, pero retiene la información suficiente para reactivar la ejecución de la función donde se quedó.
- Esto permite al código producir una serie de valores uno por uno, en vez de calcularlos y regresarlos todos.

Por ejemplo:

[12]:
```
def generadorSimple():
    print('yield_1 : ', end=' ')
    yield 1
    print('yield_2 : ', end=' ')
    yield 2
    print('yield_3 : ', end=' ')
    yield 3

for valor in generadorSimple():
    print(valor)
```

```
yield_1 :  1
yield_2 :  2
yield_3 :  3
```

- **yield** es usada como un **return**, excepto que la función regresa un objeto **generador**.
- Las funciones generadoras regresan un objeto generator.
- Los objetos generadores pueden ser usados en:
  - un **for ... in ...**
  - ejecutando la función **___next___( )** del generador.

[13]:
```
x = generadorSimple()
print(type(x))
```

```
print(x.__next__())
print(x.__next__())
print(x.__next__())
```

```
<class 'generator'>
yield_1 :  1
yield_2 :  2
yield_3 :  3
```

[14]:
```
print(x.__next__())
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-14-c15fe6e1370e> in <module>
----> 1 print(x.__next__())

StopIteration:
```

- Entonces, una función generadora regresa un objeto **generador** que es iterable, es decir, se puede usar como un **iterador**.

[15]:
```
def construyeUnGenerador(v):
    for i in range(v):       # Equivalente a: yield 0*0
        yield i*i            #                yield 1*1
                             #                yield 3*3
cuadradosY = construyeUnGenerador(10)
print(cuadradosY)
print(type(cuadradosY))

for i in cuadradosY:
    print(i)
```

```
<generator object construyeUnGenerador at 0x7fa624641f20>
<class 'generator'>
0
1
4
9
16
25
36
49
64
81
```

- Se recomienda usar **yield** cuando se desea iterar sobre una secuencia, pero no se quiere almacenar toda la secuencia en memoria.

- Si el cuerpo de la función contiene una instrucción **yield**, la función automáticamente se convierte en una función generadora.

Ejemplo 1. Crear un programa que genere los cuadrados del 1 al ∞ usando **yield**.

[16]:
```
# Función generadora infinita que genera el cuadrado de un número
def cuadradoSiguiente():
    i = 1;
    while True:
        yield i*i
        i += 1  # La siguiente ejecución se
                # reactiva en este punto

for numero in cuadradoSiguiente():
    if numero > 100:
         break
    print(numero)
```

```
1
4
9
16
25
36
49
64
81
100
```

Ejemplo 2. Crear un generador de los números de Fibonacci.

[17]:
```
def fib(limite):
    a, b = 0, 1

    while a < limite:
        yield a
        a, b = b, a + b

N = 100
x = fib(N)

print("\nUsando la función __next()__")

while True:
    try:
        print(x.__next__(), end=' ');
    except StopIteration:
        break
```

```
print("\nUsando un ciclo for ... in ...")
for i in fib(N):
    print(i, end=' ')
```

```
Usando la función __next()__
0 1 1 2 3 5 8 13 21 34 55 89
Usando un ciclo for … in …
0 1 1 2 3 5 8 13 21 34 55 89
```

[ ]: