




14 Iteradores y Generadores.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

15 Iteradores

- Como vimos en la sección XXXX, en Python existen objetos que contienen secuencias de otros objetos (listas, tuplas, diccionarios, etc).
- La mayoría de los objetos contenedores se pueden recorrer usando un ciclo **for ... in ...**. Este es un estilo claro y conveniente que impregna el universo de Python.

Por ejemplo:

```
mi_cadena = "abcd"

print("\nIteración sobre una cadena: ", end='')
for char in mi_cadena:
    print(char, end=' ')
```

Iteración sobre una cadena: a b c d

Notas importantes: - 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__()**, con el que se 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__()**, al iterador, usando la función de biblioteca **next()**.

Por ejemplo:

```
iterador = iter(mi_cadena) # Obtenemos un iterador para la cadena
print(type(iterador)) # Obtenemos el tipo del iterador
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: a
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: b
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: c
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: d
```

<class 'str_ascii_iterator'>

a

b

c
d

Cuando ya llegamos al final de la secuencia e intentamos aplicar `__next__()` obtenemos una excepción:

```
next(iterador) # Sobrepasó los elementos, se obtiene la excepción StopIteration
```

`StopIteration`:

Observa que cuando se hace el recorrido de la cadena usando el ciclo `for` no se produce ninguna excepción debido a que maneja la excepción para terminar el proceso adecuadamente.

Se puede crear un iterador y aplicarle la función `next()` a cualquier secuencia, por ejemplo a una lista

```
# Creación de una lista
cuadradosI = [x*x for x in range(10)]
print(cuadradosI)
```

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

```
# Recorriendo la lista usando un iterador en una lista concisa:
iterador = iter(cuadradosI)
[next(iterador) for x in range(10)]
```

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

Estos objetos iterables son manejables y prácticos debido a que se pueden usar tantas veces como se desee, pero se almacenan todos los valores en memoria y esto no siempre es conveniente, sobre todo cuando se tienen muchos valores.

16 Generadores

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

```
(expresion for x in secuencia)
```

donde `expresion` es una expresión válida de Python que genera los elementos del generador; `x` es un elemento al que se le aplica la `expresion` y `secuencia` es cualquier secuencia válida en Python.

Por ejemplo:

```
# Un generador simple
gen = (x for x in range(3))
```

```
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen)) # Produce una excepción de tipo StopIteration
```

```
0
1
2
```

StopIteration:

```
# Creamos el generador
cuadradosG = (x*x for x in range(10))
print(type(cuadradosG))

# Recorremos el generador en un ciclo for
for i in cuadradosG:
    print(i, end=' ')
```

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

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 - etcétera.

Un generador solo se puede usar una vez, pues va calculando sus valores uno por uno e inmediatamente los va olvidando. Si intentamos utilizar una vez más el generador, ya no obtendremos nada:

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

Observa que no se produce un error porque estamos usando el generador, que ya ha sido usado con anterioridad, dentro del ciclo `for`.

17 Yield

- 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. Si la función se vuelve a ejecutar, se reanuda desde donde se detuvo la última vez.
- Esto permite al código producir una serie de valores uno por uno, en vez de calcularlos y regresarlos todos.
- Una función que contiene la declaración `yield` se le conoce como función generadora.

Por ejemplo:

```
# Función generadora
def generadorSimple():
    print('yield 1 : ', end=' ')
    yield 1
    print('yield 2 : ', end=' ')
    yield 2
    print('yield 3 : ', end=' ')
    yield 3

# Se construye un generador
gen = generadorSimple()

# Se usa el generador
print('Primera ejecución de la función generadora: {}'.format(next(gen)))
print('Segunda ejecución de la función generadora: {}'.format(next(gen)))
print('Tercera ejecución de la función generadora: {}'.format(next(gen)))
```

```
yield 1 : Primera ejecución de la función generadora: 1
yield 2 : Segunda ejecución de la función generadora: 2
yield 3 : Tercera ejecución de la función generadora: 3
```

Si se intenta usar una vez más el generador obtendremos una excepción de tipo **StopIteration**:

```
print('Cuarta ejecución de la función generadora: {}'.format(next(gen)))
```

StopIteration:

Notas importantes. - **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 ciclos **for ... in ...** o **while**.

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

```
def construyeUnGenerador(v):
    for i in range(v):
        yield i*i

# Se construye una función generadora
cuadradosY = construyeUnGenerador(10)
print(type(cuadradosY))

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

```
<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.

17.1 Ejemplo 1.

Crear una función generadora que genere los cuadrados del 1 al ∞ .

```
# Función generadora 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

17.2 Ejemplo 1.

Crear un generador de los números de Fibonacci.

```
# Función generadora
def fib(limite):
    a, b = 0, 1

    while a < limite:
        yield a
        a, b = b, a + b # La siguiente iteración se reactiva en este punto
```

```
N = 100

# Generador
x = fib(N)

while True:
    try:
        print(next(x), end=' ') # Usamos la función next() para iterar
    except StopIteration:      # Manejamos la excepción
        break
```

0 1 1 2 3 5 8 13 21 34 55 89

```
# Usando la función generadora directamente en un ciclo for
for i in fib(N):
    print(i, end=' ')
```

0 1 1 2 3 5 8 13 21 34 55 89