

Generadores e iteradores en Python

Alejandro Fernández Camello

Python Coruña

18 de noviembre de 2023



¿En qué se diferencian el código 1 y 2?

Código 1

```
for i in list(range(1_000_000_000)):  
    print(i)
```

Código 2

```
for i in range(1_000_000_000):  
    print(i)
```



1 ¿Por qué son útiles los iteradores y generadores?

2 Iteradores

3 Generadores

4 Conclusiones



1 ¿Por qué son útiles los iteradores y generadores?

2 Iteradores

3 Generadores

4 Conclusiones



Ventajas de usar iteradores y generadores

- Permiten la evaluación perezosa de datos (*lazy evaluation*), generándolos solo cuando se requieren.



Ventajas de usar iteradores y generadores

- Permiten la evaluación perezosa de datos (*lazy evaluation*), generándolos solo cuando se requieren.
- Reducen la cantidad de memoria necesaria al almacenar menos datos.



Ventajas de usar iteradores y generadores

- Permiten la evaluación perezosa de datos (*lazy evaluation*), generándolos solo cuando se requieren.
- Reducen la cantidad de memoria necesaria al almacenar menos datos.
- Facilitan el procesamiento de datos sin necesidad de generarlos todos de antemano.



Ventajas de usar iteradores y generadores

- Permiten la evaluación perezosa de datos (*lazy evaluation*), generándolos solo cuando se requieren.
- Reducen la cantidad de memoria necesaria al almacenar menos datos.
- Facilitan el procesamiento de datos sin necesidad de generarlos todos de antemano.
- Simplifican el código y minimizan el riesgo de cometer errores.



1 ¿Por qué son útiles los iteradores y generadores?

2 **Iteradores**

3 Generadores

4 Conclusiones



Implementación

```
from collections.abc import Iterator

class MyIterator(Iterator):
    def __init__(self, *args, **kwargs):
        pass

    def __iter__(self):
        return self

    def __next__(self):
        pass
```



Iterador sobre los cuadrados de los números naturales

```
class SquareIterator(Iterator):  
    def __init__(self):  
        self.current = 1  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        self.current += 1  
        return self.current ** 2  
  
infinite_squares = SquareIterator()  
  
for i, square in enumerate(infinite_squares, 1):  
    print(f"{i} -> {square}")
```



¿Qué iteradores trae ya definidos Python?

- `range`
- `enumerate`
- `map`
- `filter`



Funcionamiento interno de range

```
class RangeIterator:
    def __init__(self, start, stop, step=1):
        self.start, self.stop, self.step = start, stop, step
        self.current = start - step

    def __iter__(self):
        return self

    def __next__(self):
        self.current += self.step
        if self.current >= self.stop:
            raise StopIteration
        return self.current

for i in range(0, 5, 2):
    print(i)
```



Funcionamiento de enumerate

```
class EnumerateIterator:
    def __init__(self, iterable):
        self.iterable = iter(iterable)
        self.index = -1

    def __iter__(self):
        return self

    def __next__(self):
        self.index += 1
        value = next(self.iterable)
        return self.index, value

for index, value in EnumerateIterator("abc"):
    print(index, value)
```



Funcionamiento de map

```
class MapIterator:
    def __init__(self, function, iterable):
        self.function = function
        self.iterable = iter(iterable)

    def __iter__(self):
        return self

    def __next__(self):
        value = next(self.iterable)
        return self.function(value)

for value in MapIterator(lambda x: x * 2, [1, 2, 3]):
    print(value)
```



Funcionamiento de filter

```
class FilterIterator:
    def __init__(self, function, iterable):
        self.function = function
        self.iterable = iter(iterable)

    def __iter__(self):
        return self

    def __next__(self):
        while True:
            value = next(self.iterable)
            if self.function(value):
                return value

for value in FilterIterator(lambda x: x % 2 == 0, [1, 2, 3, 4]):
    print(value)
```



¿No estamos teniendo que escribir demasiado código?

¿Por qué?



¿No hay alguna manera de delegar la gestión del estado al propio programa?

Sí la hay, ¡los generadores!



1 ¿Por qué son útiles los iteradores y generadores?

2 Iteradores

3 Generadores

4 Conclusiones



Generadores al rescate

Iterador

```
class FibonacciIterator:
    def __init__(self, max_value):
        self.max_value = max_value
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        self.a, self.b = self.b, \
            self.a + self.b
        if self.a > self.max_value:
            raise StopIteration
        return self.a
```

Generador

```
def fibonacci_generator(max_value):
    a, b = 0, 1
    while a <= max_value:
        yield a
        a, b = b, a + b
```



Iterador

```
for fib in FibonacciIterator(21):  
    print(i)
```

Generador

```
for fib in fibonacci_generator(21):  
    print(i)
```

¡Se usan exactamente igual que los iteradores!



La palabra mágica `yield`

- Utilizar `yield` en lugar de `return` transforma automáticamente el código en un generador en lugar de una función.
- Al alcanzar `return` o `yield`, tanto funciones como generadores devuelven el control al punto del programa que los invocó.
- Si se invoca una función de nuevo, esta se reinicia desde el principio.
- En contraste, un generador reanuda la ejecución justo después del último `yield`.



La palabra mágica yield

Generador

```
def fibonacci_generator(max_value):  
    a, b = 0, 1  
    while a <= max_value:  
        yield a  
        a, b = b, a + b
```



Convertir listas a generadores

Cuando usamos comprensión de listas hay una forma muy sencilla de convertirla en un generador. ¡Simplemente cambia los corchetes por paréntesis!

Listas a generadores

```
squares_list = [x*x for x in range(1, 6)]  
print(squares_list)  
  
squares_gen = (x*x for x in range(1, 6))  
print(squares_gen)  
print(list(squares_gen))  
[1, 4, 9, 16, 25]  
<generator object <genexpr> at 0x7fbcf1b8cc50>  
[1, 4, 9, 16, 25]
```



¡Los generadores pueden hacer aún más cosas!

- 1 Se les pueden mandar valores al generador por medio del método `send`.
- 2 Si le añadimos un `return` podemos devolver valores al terminar la ejecución del generador.



Implementación

```
def hello_n_times(n):  
    for i in range(n):  
        received = yield "Hello"  
        if received:  
            print(f"Received: {received}")  
    return f"I have printed {n} times hello"
```

```
gen = hello_n_times(3)  
print(next(gen))  
print(gen.send("World"))  
print(next(gen))
```

```
try:  
    next(gen)  
except StopIteration as e:  
    print(e.value)
```

```
Hello  
Received: World  
Hello  
Hello  
I have printed 3 times hello
```



1 ¿Por qué son útiles los iteradores y generadores?

2 Iteradores

3 Generadores

4 Conclusiones



- Los iteradores y generadores contribuyen a una mayor eficiencia al reducir el uso de memoria.
- Permiten el cálculo de valores de manera secuencial en lugar de generarlos todos a la vez.
- Los generadores logran una funcionalidad similar a la de los iteradores, pero con código más conciso y menor complejidad.
- Los iteradores *built-in* como `range`, `enumerate`, `map`, y `filter` son extremadamente útiles para simplificar el código.



¡Muchas gracias por haberme escuchado!

Tenéis disponible la presentación y el código en el siguiente enlace:
https://github.com/alexfdez1010/talk_generators_iterators

