Objetos Geradores

Juliana Pirolla - Revisão

Geradores no Python

Objetos geradores são estruturas nas quais retornam um *lazy iterator* , ou seja, que não armazenam seu conteúdo na memória e fazem a avaliação sob demanda, salvando o estado da chamada anterior da função. São objetos que possibilitam interações únicas (podemos iterar sob os itens usando next()) por não guardarem os retornos na memória, ao contrário da lista.

Referente à coomo implementar um gerador, temos três opções listadas do menos generalista ao mais geral: - Expressão Geradora; - sintaxe: `(i**2 for num in range(10))` - Função geradora; - sintaxe semelhante à implementação de funções, mas ao invés de return temos **yield**. - Objeto gerador; - transformar um objeto de uma classe em um gerador ao implementar os métodos **iter** e **next**.

Objetos geradores

Consiste em uma forma de transformar o objeto de determinada classe em que esteva definido certos métodos mágicos (**iter** e **next**).

Em resumo, o funcionamento das chamadas são: - `__iter__`: chamado quando o objeto é usado num contexto que o Python espera um iterador (por exemplo `for`). Em muitos casos basta retornar o próprio objeto - `__next__`: chamado quando o programa precisa de um próximo valor. Também sinaliza quando todos os valores já foram fornecidos através de um `raise` de `StopIteration()`.

Fazendo isso, o objeto acessa um valor por vez de um conjunto de valores.

Importante lembrar

O método `__iter__()` deve **retornar o próprio objeto** iterador, que normalmente é representado pela própria instância da classe (self). Ao retornar self na função **iter**(), estamos indicando que a própria instância da classe é o iterador e pode ser usado para iterar sobre os elementos desejados.

O método `next` **opera sobre o objeto retornado pelo** `iter`.

Exemplos

Vamos simular uma classe cujo objeto gera valores tal como o range(init, stop).

```
class MyRange:
    def __init__(self, ini, fin) -> None:
        self._ini = ini
        self._fin = fin
        self._corrente = ini

    def __iter__(self):
        return self  # indica que a própria instancia da classe é capaz de iterar

    def __next__(self):
        if self._corrente == self._fin:
            raise StopIteration()
        val = self._corrente
        self._corrente += 1  # já deixo o próximo pronto
        return val
```

```
m = MyRange(0,10)
for x in m:
    print(x)
```

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

Chamando diretamente os métodos

Note que podemos acessar diretamente os métodos `__iter__` e `__next__` ao usar iter e next.

```
m = MyRange(0,10)
mi = iter(m)
next(mi)
next(mi)
```

```
1
```

```
for i in range(5):
    next(mi)
```

Opção para lidar com geradores exauridos

Como vimos, os geradores poder ser iterados uma única vez. Para podermos utilizar os valores mais de uma única vez, podemos converter os objetos gerados para uma lista (muitas vezes não é o que queremos, pois teremos certo desperdício de memóra para listas grandes).

Uma opção é **alterar o método `__iter__`** e "modularizar" o processo retornando um objeto de outra chasse. Essa outra classe será a responsável por implementar o next.

```
# Esta classe controla quais valores já foram varridos.
class MyRangeController:
    def __init__(self, rg):
        self._range = rg
        self._corrente = rg._ini

    def __next__(self):
        if self._corrente == self._fin:
            raise StopIteration()
        val = self._corrente
        self._corrente +=1
        return val

# Esta classe controla os valores que pertencem à faixa desejada
class MyRangeIterator:
    def __init__(self, ini, fin):
        self._ini = ini
        self._fin = fin

    def __iter__(self):
        return MyRangeController(self)
```

```
m = MyRange(1, 5)
mi = iter(m)
type(m), type(mi)
```

```
(__main__.MyRange, __main__.MyRange)
```

```
next(mi), next(mi)
```

```
(1, 2)
```

```
for x in m:
    print(x, end=' ')
```

```
3 4
```

Método sem uma classe auxiliar

Podemos retornar uma função geradora no método **iter**.

```
class MyRange:
    def __init__(self, ini, fin):
        self._ini = ini
        self._fin = fin

    def __iter__(self):
        for i in range(self._ini, self._fin):
            yield i
```

Iter e next no operador in

Tambḿe implementamos os métodos **iter** e **next** para avaliar o objeto à direita do operador in. Isso se faz necessário pois o operador in é utilizado em situações nas quais compara-se um objeto com outras estruturas tal como dicionarios, listas, sets etc, avaliando a operação de pertencimento ou não àquele subconjunto.

- Chama o método **iter**, com o objeto retornado, chama **next** sucessivamente e comparando com o valor à esquerda do in.

- Se encontrar igualdade entre o objeto à esquerda e o objeto que está sendo avaliado, retorna True.
- Se percorrer todo o subconjunto sem existir um valor igual, retorna False.

As vezes implementar tal método não é muito efetivo dado a necessidade de realizae muits comparaçõs.

Método contains

Forma mais eficaz de implementar o operador in.

```
class MyRange:
    def __init__(self, ini, fin):
        self._ini = ini
        self._fin = fin

    def __iter__(self):
        for i in range (self._ini, self._fin):
            yield i

    def __contains__(self, val):
        return self._ini <= val < self._fin
```