Exercícios sobre Iteradores e Geradores em Python

Carlos Camarão

1 de Outubro de 2018

1 Iteradores e Geradores

1.1 Iteradores

Iteradores são usados em comandos **for** para obtenção de cada elemento de uma estrutura de dados iterável, em cada iteração do comando: o primeiro elemento é obtido pela chamada ao construtor do iterador e o próximo elemento, em cada iteração do comando **for**, é obtido por chamada ao método **next**.

Por exemplo, pode-se usar comandos **for** para obter cada elemento de:

1. uma lista:

```
>>> for i in [1, 2, 3, 4]:
...     print i,
...
1
2
3
4
```

2. caractere de uma cadeia de caracteres:

```
>>> for c in "abcd":
...     print c
...
a
b
c
d
```

3. chaves de um dicionário:

```
>>> for chave in {"a": 1, "b": 2}:
...     print chave
...
a
b
```

4. linhas de um arquivo-texto:

```
>>> for lin in open("arq.txt"):
...     print lin
...
primeira linha
segunda linha
```

Iteradores são objetos de classes que definem métodos `__iter__`, para criar um objeto iterador, e **next**, para retornar um (próximo) elemento a cada chamada. Por exemplo, considere a definição a seguir, de iterator que se comporta como a função pré-definida **xrange** (**xrange** não gera toda a lista de valores, como a função **range**, mas um iterador, i.e. gera os elementos da lista um por um em vez de gerar toda a lista):

```
class xrange_:
    def __init__(self, n):
        self.i = 0
        self.n = n

    def __iter__(self):
        return self

    def next(self):
        if self.i < self.n:
            i = self.i
            self.i += 1
            return i
        else:
            raise StopIteration()
```

Exemplo de uso de `xrange_` e **next**:

```
>>> y = xrange_(3)
>>> y.next()
0
>>> y.next()
1
>>> y.next()
2
>>> y.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in next
StopIteration
```

Exemplos de funções que têm iteradores como argumentos são **list** e `join`:

```
>>> ",".join(["a", "b", "c"])
'a,b,c'
>>> ",".join({"x": 1, "y": 2})
'y,x'
>>> list("abc")
['a', 'b', 'c']
>>> list({"x": 1, "y": 2})
['y', 'x']
```

Note: **iter** sobre um dicionário é o mesmo que `iterkeys`: realiza iteração sobre as chaves do dicionário.

Chaves de um dicionário têm uma ordem arbitrária (não são ordenadas necessariamente na ordem em que aparecem no texto).

Geradores facilitam a geração de iteradores.

1.2 Geradores

Um gerador é um objeto iterador criado por meio do comando `yield` (veja explicação a seguir) ou por meio de uma *expressão geradora*. Por exemplo, um objeto gerador pode ser definido simplesmente como resultado de chamada a uma *função-geradora* `xrange_` definida como a seguir:

```
def xrange_(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

O comando `yield` retorna um objeto gerador (iterador criado via `yield`). Cada chamada a **next** retorna um valor. Por exemplo:

```
>>> x_ = xrange_(3)
>>> x_
<generator object xrange_ at ...>
>>> x_.next()
0
>>> x_.next()
1
>>> x_.next()
2
>>> x_.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

O uso de `yield` é um modo de implementar explicitamente a estratégia de avaliação preguiçosa (um valor é obtido a cada chamada a **next**, em vez de ser obtido automaticamente quando é usado, como ocorre no caso da estratégia de avaliação preguiçosa). Em ambos os casos o objetivo é a otimização de espaço (i.e. evitar o armazenamento de todos os valores ao mesmo tempo).

1.3 Expressões Geradoras

Expressões generadoras geram valores como em uma lista definida por geração e filtragem, mas usam parênteses em vez de colchetes, criando um gerador (iterador gerado por expressão geradora):

```
>>> xx = (x*x for x in range(10))
>>> xx
<generator object <genexpr> at ...>
>>> list(xx)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

1.4 Exercício Resolvido

Escreva um programa que usa geradores para escrever função que recebe uma cadeia de caracteres (*string*) `s` e um iterador de nomes de arquivos-texto, e imprima o conteúdo de cada linha de cada arquivo-texto que contém a cadeia `s`.

Solução:

```
def ylins (arqs):
    for f in arqs:
        for lin in open(f):
            yield lin

def grep (s,linhas):
    return (lin for lin in linhas if s in lin)

def printLins (lins):
    for lin in lins:
        print lin

def main(s, arqs):
    lins = yLins(arqs)
    lins = grep(s, lins)
    printLins(lins)
```

1.5 Exercícios

Use geradores (gerados pelo comando `yield`) para implementar as funções a seguir.

1. Escreva uma função que receba um inteiro n e uma lista de nomes de arquivos como argumentos e imprima todas as linhas contidas nos arquivos com tamanho maior que n caracteres.

2. Escreva um programa que receba uma lista de nomes de arquivos como argumentos e imprima i) para cada arquivo, seu nome e o número de linhas contidas no arquivo, e ii) o número total de linhas contidas nos arquivos.

3. Escreva uma função *enumerate_* que se comporte como *enumerate*.

   A função *enumerate* recebe um iterador *it* e retorna um iterador sobre pares `(`i`,`v`)`, onde i é um índice e v um valor, do iterador *it*.

   Por exemplo:

```
>>> list(enumerate(["a", "b", "c"])
[(0, "a"), (1, "b"), (2, "c")]
>>> for i, c in enumerate(["a", "b", "c"]):
...     print i, c
...
0 a
1 b
2 c
```