Live de Python

Iteradores e geradores

Obrigado!

apoia.se/livedepython

```
z4r4tu5tr4 at babbage in ~/git/apoiadores
$ python apoiadores.py
Amaziles Carvalho    Andre Machado        Bruno Guizi          David Reis
Dayham Soares        Edimar Fardim        Eliabe Silva         Elias Soares
Emerson Lara         Fabiano Silos        Fabiano Teichmann    Fabiano Gomes
Fernando Furtado     Fábio Serrão         Gleison Oliveira     Humberto Rocha
Jean Vetorello       Johnny Tardin        Jonatas Oliveira     Jonatas Simões
João Lugão           Jucélio Silva        Júlia Kastrup        Leon Teixeira
Magno Malkut         Maria Boladona       Nilo Pereira         Pablo Henrique
Paulo Tadei          Pedro Alves          Rafael Galleani      Regis Santos
Renan Moura          Renato Santos        Rennan Almeida       Rodrigo Vaccari
Sérgio Passos        Thiago Araujo        Tiago Cordeiro       Vergil Valverde
Vicente Marcal       Wander Silva         Wellington Camargo   Welton Souza
William Oliveira     Willian Gl           Yros Aguiar          Falta você
```

Python Pro

Python para Profissionais

Roteiro

- Um basicão da massa
  - for
  - slice
  - sequências
- Entendendo o padrão iterator
- Aplicando iterator em python
- Geradores
- Lazy evaluation x eager evaluation

Um basicão da massa

Qualquer coisa que é possível fazer um **for**, pode ser considerado um iterador.

```
In [1]: string = 'Live de Python'

In [2]: type(string)
Out[2]: str

In [3]: for letra in string:
   ...:     print(letra)
   ...:
```

Live de Python

Um basicão da massa

Ou seja, qualquer coisa que pode ser “desmontada” ou “ter coisas dentro” e me permite acessar essas coisas sequencialmente (a grosso modo), pode ser considerado um iterador. Entre os tipos “nativos” temos:

- Listas
- Tuplas
- Strings
- Dicionários
- Conjuntos

Um basicão da massa

A ideia principal de um iterador é resolver o problema de trabalhar com sequências que não cabem na memória.

Por exemplo, um arquivo muito grande, uma resposta complexa de banco de dados etc...

Slice

Quando falamos de sequências em Python, contamos com o slice. Que é uma maravilha, diga-se de passagem.

Isso é implementado pelo protocolo de sequência (Ver live de python #32)

Onde implementamos o método **__getitem__**

Slice

Onde implementamos o método **__getitem__**

Live de Python

Slice

Onde implementamos o método **__getitem__**


```
In [1]: l = [2, 4, 6]

In [2]: l[0], l[1], l[2]
Out[2]: (2, 4, 6)

In [3]: l[0]
Out[3]: 2
```

Live de Python

Slice

Onde implementamos o método **__getitem__**

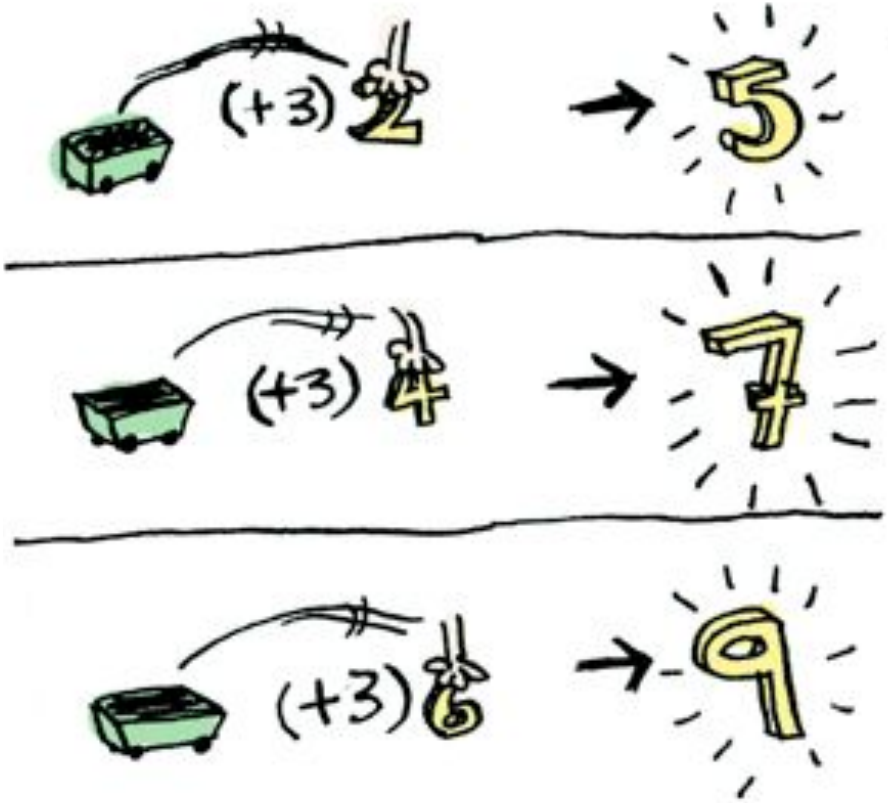

```
In [1]: l = [2, 4, 6]

In [2]: l[0], l[1], l[2]
Out[2]: (2, 4, 6)

In [3]: l[0]
Out[3]: 2
```

Live de Python

Slice




```
In [5]: for n in [1,2,3]:
   ...:     print(n + 3)
   ...:
4
5
6
```

Slice - Listas

```
>>> n = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> n[0]      # 0
>>> n[6:]     # [6, 7, 8, 9]
>>> n[:-6]    # [0, 1, 2, 3]
>>> n[::2]    # [0, 2, 4, 6, 8]
```

[De onde : até onde: de quanto em quanto]

[2 : 10 : 3]

```
>>> n[:3]      # [0, 1, 2, 3]
>>> n[::2]     # [0, 2, 4, 6, 8]
```

Implementando minha sequência

Se eu quiser imitar o comportamento dos objetos do python, podemos implementar o método **__getitem__**. Que explica pro python como pegar 1 a 1 os dados do nosso tipo

```
In [6]: class MinhaSequencia:
   ...:     def __init__(self, dados):
   ...:         self.dados = dados
   ...:     def __getitem__(self, posição):
   ...:         return self.dados[posição]
   ...:

In [7]: MinhaSequencia([1,2,3,4])
Out[7]: <__main__.MinhaSequencia at 0x7fa9088aeda0>
```

Live de Python

Implementando minha sequência

```
In [8]: list(MinhaSequencia([1,2,3,4]))
Out[8]: [1, 2, 3, 4]

In [9]: for elemento in MinhaSequencia([1,2,3,4]):
   ...:     print(elemento)
   ...:
1
2
3
4

In [10]: MinhaSequencia([1,2,3,4])[-1]
Out[10]: 4
```

Ok! Estamos nivelados

Mas como o python faz pra iterar?

Entendendo o iterador

Toda vez que o python vai “iterar” sobre algo, ele chama uma função chama **iter()**.

```
In [11]: iter([1,2,3,4])
Out[11]: <list_iterator at 0x7fa90834ae10>

In [12]: iter([1,2,3,4])[0]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-b7ddce99bcac> in <module>
----> 1 iter([1,2,3,4])[0]

TypeError: 'list_iterator' object is not subscriptable
```

Live de Python

Exemplos

A função **enumerate()** é um bom exemplo de iterador. Você não consegue acessar os itens da sequência individualmente, mas o **for** sabe como fazer isso.

```
In [18]: enumerate([1,2,3])[-3]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-a01a0f935532> in <module>
----> 1 enumerate([1,2,3])[-3]

TypeError: 'enumerate' object is not subscriptable

In [19]: for e in enumerate([1,2,3]):
    ...:     print(e)
    ...:
(0, 1)
(1, 2)
(2, 3)
```

Exemplos

Temos uma gama de funções que mostra resultados dessa maneira:

- map
- filter
- zip
- reversed
- Módulo itertools
  - count, takewhile, groupby, ...

Beleza, mas o que acontece?

Todos os objetos **iteráveis** implementam um método chamado **__iter__**, que é responsável por retornar um **iterador**.

?????????

Iterável vs Iterador

iterável é qualquer coisa que chamamo pela função **iter()** retorna um iterador.

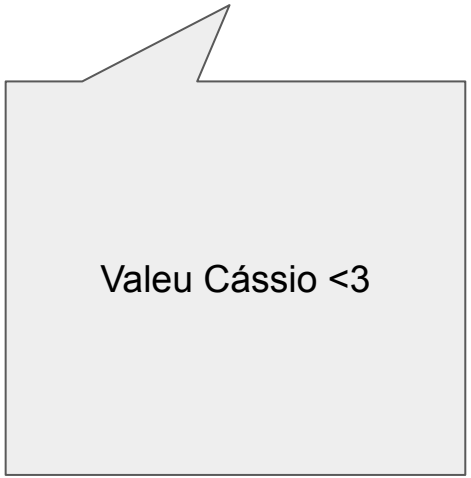
Iterador é a coisa que sabe como chamar o próximo da sequência usando a função **next().**

??????????????

Iterável vs Iterador

iterável é qualquer coisa que chamamo pela função **iter()** retorna um iterador.

Iterador é a coisa que sabe como chamar o próximo da sequência usando a função **next().**



??????????????

Live de Python

Tá, vamos exemplificar (Por Luciano Ramalho)

De maneira transparente

```
In [24]: frase = 'Olár bbs'

In [25]: for letra in frase:
    ...:     print(letra)
    ...:
O
l
á
r

b
b
s
```

Tá, vamos exemplificar (Por Luciano Ramalho)

De maneira transparente

```
In [24]: frase = 'Olár bbs'

In [25]: for letra in frase:
    ...:     print(letra)
    ...:
O
l
á
r

b
b
s
```

Vamos pensar que não temos for

```
In [26]: frase = 'Olár bbs'

In [27]: iterador = iter(frase)

In [28]: iterador
Out[28]: <str_iterator at 0x7fa9088b8550>

In [29]: while True:
    ...:     try:
    ...:         print(next(iterador))
    ...:     except StopIteration:
    ...:         break
    ...:
O
l
á
```

Iteradores

Então, no caso anterior, a função **iter()**, retornou um objeto iterador <iterator>. Que dentro dele tem um método chamado **__next__**, que sabe “como chamar o próximo”.

Iteradores

```
In [52]: class MeuIterador:
    ...:     def __init__(self):
    ...:         self.l = [1,2,3]
    ...:         self.index = 0
    ...:     def __iter__(self):
    ...:         return self # <- retorna o ob. que tem o __next__
    ...:     def __next__(self):
    ...:         """Explica como pegar o próximo elemento."""
    ...:         try:
    ...:             proximo = self.l[self.index]
    ...:             self.index += 1
    ...:         except IndexError:
    ...:             raise StopIteration()
    ...:         return proximo
    ...:
```

Tá complicado, eu sei.

Mas agora tudo vai melhorar

Geradores

Geradores são maneiras de “fazer” um iterador. Sem toda aquela confusão de classes, mas a introdução foi importante. Pq você sabe agora como funciona tudo nos mínimos detalhes.

yield

Ah... o Yield....

A palavra reservada **yiled** pode ser usado no corpo de qualquer função*. Isso vai transformar a sua função em um iterável. Tá, vamos lá...

```
In [58]: def geradora():
    ...:     yield 'A'
    ...:     yield 'B'
    ...:     yield 'C'
    ...:

In [59]: geradora()
Out[59]: <generator object geradora at 0x7f3d40183c00>
```

Ah... o Yield....

A palavra reservada **yiled** pode ser usado no corpo de qualquer função*. Isso vai transformar a sua função em um iterável. Tá, vamos lá...

```
In [58]: def geradora():
    ...:     yield 'A'
    ...:     yield 'B'
    ...:     yield 'C'
    ...:

In [59]: geradora()
Out[59]: <generator object gerac
```

```
In [64]: g = geradora()

In [65]: next(g)
Out[65]: 'A'

In [66]: next(g)
Out[66]: 'B'

In [67]: next(g)
Out[67]: 'C'
```

Yield

yield pode nos ajudar a escrever geradores de maneira simples e gerar iteráveis de maneira graciosa. Sem **__iter__**, sem **__next__**. A Linguagem faz isso pra você. Você só se preocupa com o é necessário.

```
In [68]: def contador():
    ...:     index = 0
    ...:     while True:
    ...:         yield index
    ...:         index += 1
    ...:
```

Yield

yield pode nos ajudar a escrever geradores de maneira simples e gerar iteráveis de maneira graciosa. Sem **__iter__**, sem **__next__**. A Linguagem faz isso pra você. Você só se preocupa com o é necessário.

```
In [69]: c = contador()

In [70]: next(c), next(c), next(c), next(c)
Out[70]: (0, 1, 2, 3)

In [71]: next(c), next(c), next(c), next(c)
Out[71]: (4, 5, 6, 7)

In [72]: next(c), next(c), next(c), next(c)
Out[72]: (8, 9, 10, 11)
```

Nomenclatura

Quando usamos **return**, dizemos que a função ‘retorna algo’. Quando usando **yield** dizemos que a função faz/gera algum tipo de valor.

Função geradora

Você já entendeu que as funções geradoras **geram** valores. A parte mais legal é que você "descreve" como gerar o próximo valor e ganha um iterador. Isso é **lindo**

```
In [73]: def multiplo(valor, inicio=1):
    ...:     while True:
    ...:         yield valor * inicio
    ...:         inicio += 1
    ...:
```

yield from

A combinação do sucesso

A ideia de fazer um for dentro de outro acabou, obrigado python 3.3

```
In [1]: def raio_simplificador(lista_de_listas):
   ...:     for lista in lista_de_listas:
   ...:         yield from lista
   ...:

In [2]: list(raio_simplificador([ [1], [2]]))
Out[2]: [1, 2]
```