O temido `Map()`

Eduardo Mendes
github.com/z4r4tu5tr4

z4r4tu5tr4@babbge: screenfetch

Nome: Eduardo Mendes
Instituição: Fatec Americana
Uptime: 12097080s
Email: mendexeduardo@gmail.com
git: github.com/z4r4tu5tr4

Map, pra que? Tenho listcomps!

	map	Comp
List	map(func, iter)	[func(x) for x in iter]
Gen	map(func, iter)	(func(x) for x in iter)
Set	map(func, iter)	{func(x) for x in iter}
Dict	map(func, iter)	{func(x):func(y) for x,y in iter}

Map, pra que? Tenho listcomps!

	map	**Comp**
List	**list**(map(func, iter))	[func(x) for x in iter]
Gen	map(func, iter)	(func(x) for x in iter)
Set	**set**(map(func, iter))	{func(x) for x in iter}
Dict	**dict**(map(func, iter)	{func(x):func(y) for x,y in iter}

Familia comp x Recursividade (????)

map + lambda

```
func = lambda x: x*2 \
        if \
           not(x < 10 and x > 0)\
        else \
           func(x+10)

map(func, [-4,5,6,11]) #[-8, 30, 32, 22]
```

Familia comp x Recursividade (????)

```
[x*2 for x in [-4,5,6,11]\
        if not(x< 10 and x> 0)\
        else (???)]

         ^
SyntaxError: invalid syntax
```

Familia comp x R

```
[x*2 for x in [-
        if not(x<
        else (???)]

         ^
SyntaxError: invalid syntax
```

Isso é um filtro, mas sim, poderíamos ter usado um lambda

Então... Vamos falar de Map?

Diferenças entre Python 2 e Python3

- Python 2:
  - In: map(func, [1,2,3,4])
  - Out: [1,2,3,4]

func = lambda x: x

- Python 3
  - In: map(func, [1,2,3,4])
  - Out: <map at 0x7fcae36c4828>

Diferenças

- Python 2:
  - In:
  - Out:

- Python 3
  - In: map(func(x ,2,3,4])
  - Out: <map at 0x7fcae36c4828>

Lazy evaluation
(Avaliação preguiçosa)

Só executo quando for preciso

next()

Lazy evaluation - Geradores - Call-by-need

```
 2016-03-07 17:39:26 ☆  Babbage in ~
○ → python -m memory_profiler  test.py
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
     3   26.938 MiB    0.000 MiB   @profile
     4                             def vetor():
     5   70.750 MiB   43.812 MiB       return [x for x in range(1058**2)]


Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
     7   70.750 MiB    0.000 MiB   @profile
     8                             def g_vetor():
     9   70.750 MiB    0.000 MiB       return (x for x in range(1058**2))
```

Lazy evaluation - Geradores - Call-by-need

```
2016-03-07 17:39:26 ☆  Babbage in ~
○ → python -m memory_profiler test.py
Filename: test.py

Line #    Mem usage    Increment
===============================
     3   26.938 MiB    0.000 MiB
     4
     5   70.750 MiB   43.812 MiB


Filename: test.py

Line #    Mem usage    Increment
===============================
     7   70.750 MiB    0.000 MiB
     8
     9   70.750 MiB    0.000 MiB
```

```
@profile
def vetor():
    return [x for x in range(1058**2)]

@profile
def g_vetor():
    return (x for x in range(1058**2))

#1119364
x = vetor()
y = g_vetor()
```

itertools.imap (Python 2)

```
from itertools import imap

in:   imap(lambda x:x, [1,2,3])
out:  <itertools.imap object at
0x7fc8b8928710>
```

from fn.iters import map (facilitando a migração)

Python 2.7.11

```
>>> from fn.iters import map
>>> map(lambda x: x, [1,2,3])
<itertools.imap object at 0x7f52b6393550>
```

Python 3.5.1

```
>>> from fn.iters import map
>>> map(lambda x: x,[1,2,3])
<map object at 0x7f4b664a5898>
```

from fn import _ (lambdas legíveis)

```
from fn import _

func = (_*2) / _

func(2,0.5)     #8
```

```
from fn import _

list(map(_, [1,2,3]))

#[1,2,3]
```

from fn import _ (lambdas legíveis)

```
from fn import _

func = (_*2) / _

func(2,0.5)     #
```

```
from fn import _

list(map(_, [1,2,3]))

#[1,2,3]
```

```
type(_)
fn.underscore._Callable
```

Eliminando Loops

Funcional, porque funcional é legal*

***Use com moderação**

O poder oculto dos lambdas em maps [0]

```
do = lambda f, *args: f(*args)
```

- **Uma função que resolve funções**

```
map(do, [func1, func2], [l_arg1, l_arg2])
```

- **Um iterador de funções**

O poder oculto dos lambdas em maps [1]

```
do = lambda f, *args: f(*args)
```

```
hello = lambda first, last: print(“Olá”, first, last)
```

```
bye = lambda first, last: print(“Adeus”, first, last)
```

```
map(do, [hello, bye],[‘Amom’,’Eric’], [‘Mendes’,’Hideki’])
```

```
Out: Olá Amom Mendes
     Adeus Eric Hideki
```

O poder oculto dos lambdas em maps [2]

do = lambda **f, *args**: f(*args)

hello = **lambda** first, last: **print**(“Olá”, first, last)

bye = **lambda** first, last: **print**(“Adeus”, first, last)

map(do, [hello, bye],[‘Amom’,’Eric’], [‘Mendes’,’Hideki’])

Out: Olá Amom Mendes
Adeus Eric Hideki

O poder oculto dos lambdas em maps [3]

```
do = lambda f, *args: f(*args)

hello = lambda first, last: print("Olá", first, last)

bye = lambda first, last: print("Adeus", first, last)

map(do, [hello, bye],['Amom','Eric'], ['Mendes','Hideki'])
```

Out: Olá Amom Mendes
Adeus Eric Hideki

Um for sem for [0]

```
do_all = lambda fns, *args: [
                                list(map(fn, *args)) for fn in fns]
```

```
_ = do_all([hello, bye], [‘Amom’,’Eric’],
                                [‘Mendes’,’Hideki’])
```

Um for sem for [1]

Func	Arg 1	Arg 2
Hello	Amom	Mendes
Hello	Eric	Hideki
Bye	Amom	Mendes
Bye	Eric	Hideki

Olá Amom Mendes
Olá Eric Hideki
Adeus Amom Mendes
Adeus Eric Hideki

Um while sem while

```
def n_while(x):
        print(x)        # Exibe na tela
        return x        # Garante a saída
```

```
diga = lambda: n_while(
                        input("zz, disse "))=='q' or diga()
```

Pool maps

Resolvendo pontos críticos de execução

Facilitando pontos críticos

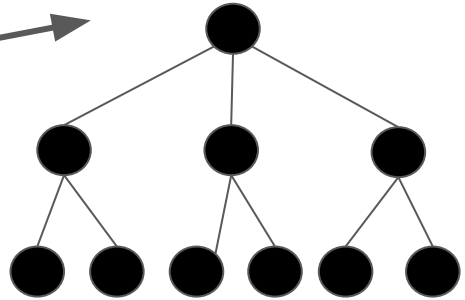
```
>>> os.cpu_count()
    #n
```

- Consigo N threads por vez

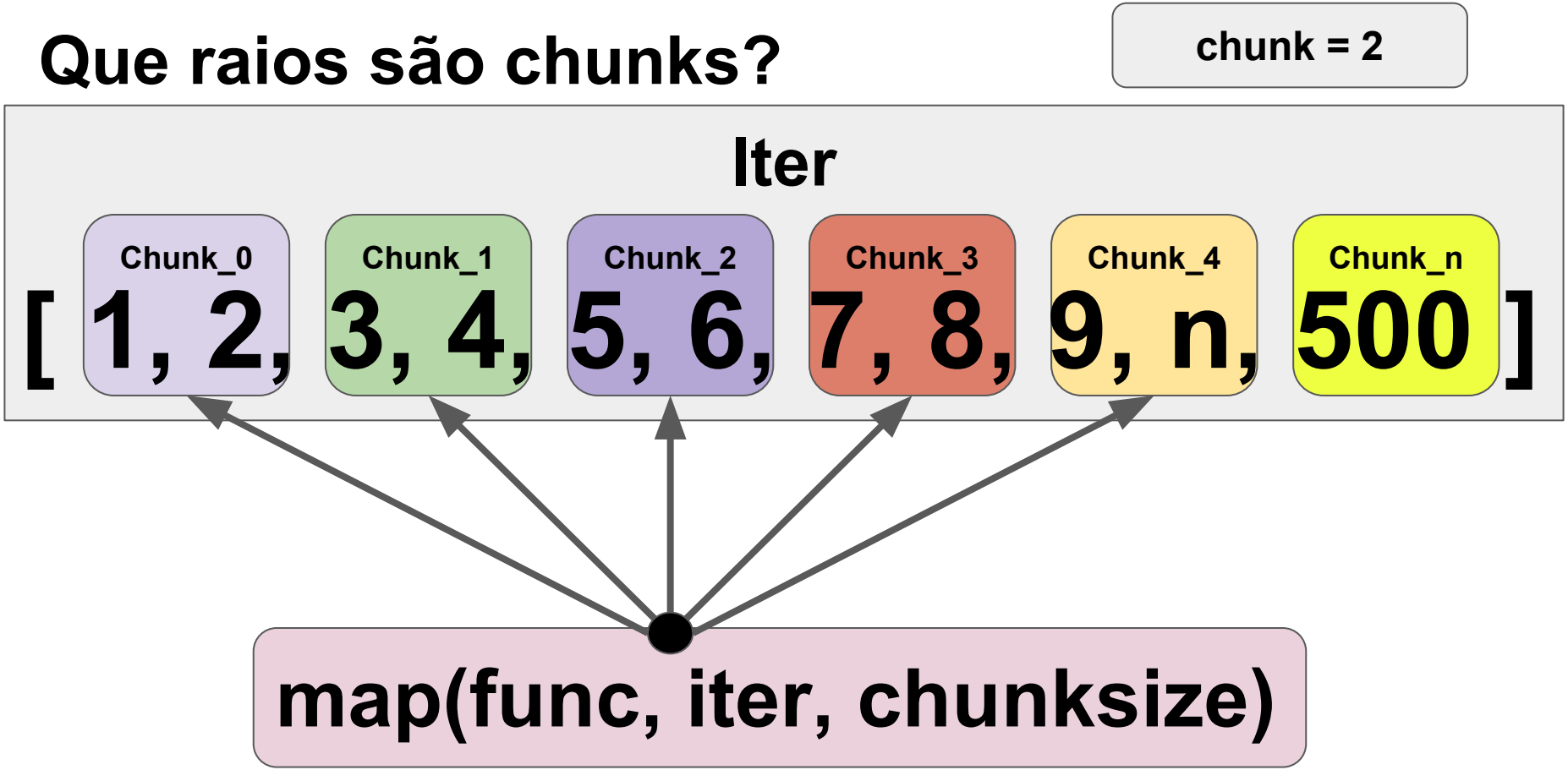
```
>>> Pool._maxtasksperchild
    #n
```

- Termino a thread depois de N

- **1 Programa**
- **3 Theads**
- **6 por thread**



Que raios são chunks?



map()

```
>>> from multiprocessing import Pool
>>> from os import cpu_count

>>> node = Pool(cpu_count)

>>> node.map(lambda x:x, range(100), 10])

# [0,1,2,3,4,5,6,7,8,9 ... 99]
```

 # Lista

8 threads, 10 por thread.

map_async()

```
>>> node.map_async(lambda x:x, range(100), 10])
```

self.get()

self.ready()

self.sucessful()

imap()

```
>>> node.imap(lambda x:x, range(100), 10])
```

self.next()

self._unsorted
 # {chunk_n: val}

imap_unordered()

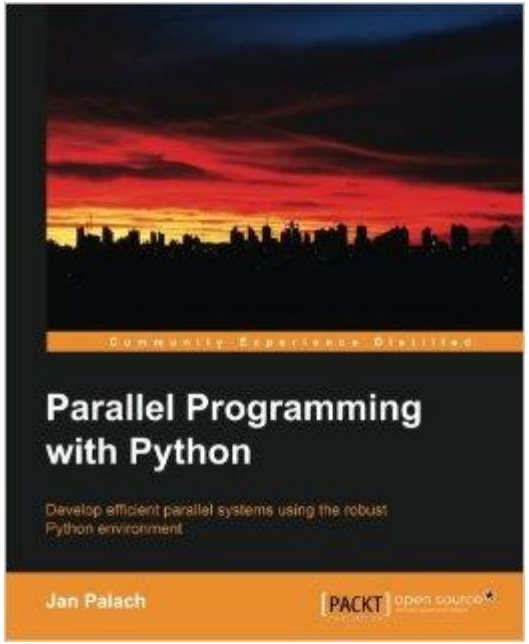
```
>>> node.imap_unordered(lambda x:x, range(100), 10])
```

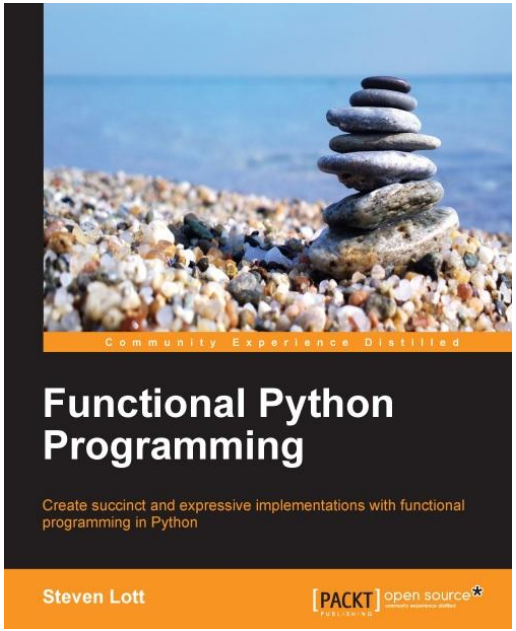
É mais rápido, mas não ordena

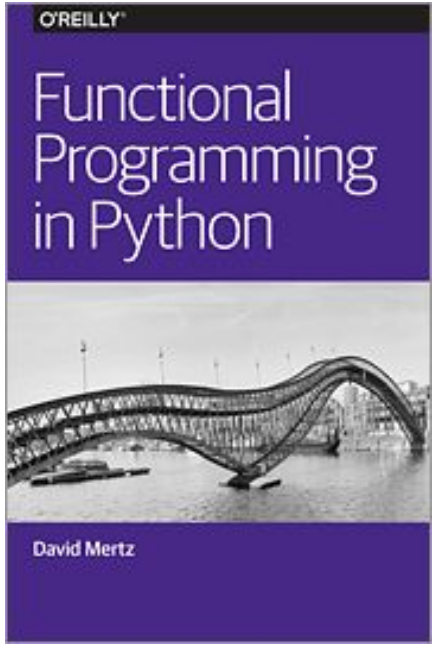
self.next()

self._unsorted
 # {chunk_n: val}

Quero mais, e agora?







Jan Palach **Steven Loft** **David Metz**

docs.python.org/3/library/multiprocessing.html

XOXO

Dúvidas?

mendesxeduardo@gmail.com