python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

Project maintained by dunossauro
Hosted on GitHub Pages — Theme by mattgraham

Retornar «< 4. Funções de redução/mapeamento - Continue lendo »» 6. Funções de ordem superior p.2

5. Funções de ordem superior

Você deve achar que esquecemos muitas funções embutidas no tópico passado, não? Funções como:

- map()
- max()
- min()
- iter()
- sorted()
- filter()

Porém, essas funções têm características especiais. Como assim? Elas podem receber além do iterável, uma outra função como argumento. Vamos lá. Você já foi introduzido ao map() no tópico passado.

5.1 map()

A função map(), fazendo um gancho com o tópico anterior, é uma função de mapeamento, contudo, ela recebe o iterável em conjunto a uma função, a que fará o mapeamento. Vamos lá:

```
def func(x):
    """
    Exemplo do tópico passado
    """
    return x +2
```

```
list(map(func, [1,2,3])) # [3, 4, 5]
```

A função que chamamos de func() é uma função extremamente simples, retorna a entrada somada com 2, simples assim. Um ponto que vale a pena ser tocado é que as funções usadas por map() só podem receber um argumento. Por quê? A função map() vai pegar um elemento da sequência e aplicar a função. Só isso, sério.

Agora vamos complicar as coisas um pouco mais....

```
# Uma lista de listas, isso em python também é uma matriz
lista = [[1,2], [2,3], [3,4], [4,5], [5,6]]

def func(x):
    """
    Retorna a mesma coisa que entrou
    """
    return x

list(map(func, lista))
```

Você concorda comigo que a entrada não é exatamente um elemento único, mas uma sequência?

Então, como temos uma lista agora, podemos fazer coisas de lista? Aplicar outras funções de iteráveis? Vamos lá:

```
# Uma lista de listas, isso em python também é uma matriz
lista = [[1,2], [2,3], [3,4], [4,5], [5,6]]

def func_rev(x):
    """
    Retorna a lista que entrou, porém invertida
    """
    return list(reversed(x))

list(map(func_rev, lista)) # [[2, 1], [3, 2], [4, 3], [5, 4], [6, 5]]
```

Isso, isso, isso. Olha como a coisa está ficando linda? O que fizemos agora é uma composição de funções, mas dentro de um map(). A notação matemática disso, caso você

```
uma função comum = f(x)

composição de funções = f(g(x))
```

Bom, agora você aprendeu o poder do map(), podemos viajar entre a outra gama de funções de ordem superior que o python oferece. Porém, fica o adendo teórico:

```
Funções de ordem superior são funções que recebem funções como argumento, ou
```

Viu, foi simples.

5.2 max()

A função max() é uma função de redução, e sem a função como parâmetro, ela vai ter o comportamento das funções que vimos no outro tópico.

```
max([1, 2, 3, 4, 5]) # 5
```

Só que... (êee... lá vem)

Se essa lista for uma lista de listas, como prosseguir?

```
lista = [[1,2], [2,3], [3,4], [4,5], [5,6]]

max(lista) # [5, 6]
```

É, ainda está funcionando.

```
lista = [[7,2], [5,3], [5,4], [5,5], [5,6]]

max(lista) # [7, 2]
```

`[7, 2]` é um bom resultado, mas vamos pensar que o que eu queria eram as somas dois dois elementos, nesse caso o resultado veio errado. `7 + 2 = 9` quando `5 + 6 = 11` . Vamos tentar outra vez:

```
def func(x):
    return x[0] + x[1]

max(lista, key=func) # [5,6]
```

Viu, esse argumento `key` pode ser uma mão na roda em muitos dias tristes em que você está sem vontade de cantar uma bela canção.

```
lista = [[7,2], [5,3], [5,4], [5,5], [5,6]]

max(lista, key=sum) # [5, 6]
```

Como você já sabe compor funções, vamos imaginar que nossa sequência de entrada poderia ser maior que dois elementos, uma maneira bonita de fazer isso seria usar o sum(). Fica muito mais elegante.

5.3 min()

Agora que já entendemos o conceito das HOFs, tudo fica mais simples. A função min() é a função equivalente a max(). Quando a max() pega o maior item da sequência, min() pega o menor.

```
lista = [[7,2], [5,3], [5,4], [5,5], [5,6]]

min(lista, key=sum) # [5, 3]
```

Não temos muito mais o que falar sobre min, é só um complemento.

5.4 iter()

A função embutida iter() tem duas formas, a primeira devolve o iterável de uma sequência.

```
lista = [1, 2, 3, 4, 5]

iter(lista) # <list_iterator at xpto>
```

Ele faz a chamada do método `__iter__()` do objeto. Até então nenhuma novidade.

imutáveis e preguiçosas.

Porém, a segunda forma é bem interessante.

```
"""
Exemplo roubado do Steven Lott
"""
lista = [1, 2, 3, 4, 5]

list(iter(lista.pop, 3)) #[5, 4]
```

Vamos por partes que agora vem muita informação pra pouca linha de código.

vamos dar um help() em iter():

```
help(iter)

iter(...)
    iter(iterable) -> iterator
    iter(callable, sentinel) -> iterator

    Get an iterator from an object.  In the first form, the argument must
    supply its own iterator, or be a sequence.
    In the second form, the callable is called until it returns the sentinel.
```

Então, quer dizer que o callable é chamado até que o retorno seja o sentinela. Como passamos como callable pop() e se pop() for chamado sem argumentos, ele retorna o último elemento da lista e retira ele da mesma. Nesse caso o sentinela é 3. Então ele vai desmontando a lista e gerando um novo iterável de tudo que foi removido da lista anterior.

Ou seja, vamos fazer um mapinha básico:

```
(1) lista = [1, 2, 3, 4, 5]

(2) saida = []

(3) saida.append(lista.pop())
```

Agora que o pop_append ficou claro, deu pra entender o que faz a segunda forma da função iter()? Sim, deu.

Então vamos explorar um exemplo mais eficiente, o da documentação:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        print(line) # só essa linha foi modificada
```

O método readline, quando passado sem parâmetros efetua a leitura de um único caracter. Nesse caso ele printaria uma letra do arquivo por linha. Mas, como usamos `iter(fp.readline, '')` ele vai nos retornar uma sequência de strings até que o sentinela que no caso é vazio apareça.

Ou seja, é passado um objeto com um método no lugar de uma função. O método tem suas particularidades como não precisar de argumentos e agir no objeto em si. Isso parece óbvio, porém, quando construímos nossas próprias classes, o retorno pode não ser o esperado, como nas sequências embutidas do python.

5.5 sorted()

Para os viciados em listas, como eu, o método sort da lista funciona bem, apesar de ordenar a lista e não trazer uma nova lista, o que as vezes é uma dor de cabeça.

```
lista = [1, 2, 3, 3, 2, 1]

lista.sort()

print(lista) # [1, 1, 2, 2, 3, 3]
```

Para agradar a todos, temos a função embutida sorted().Assim como as outras HOFs, temos o parâmetro opcional `key` e podemos decidir como a ordenação será feita.

Vamos pensar em uma tupla de tuplas, uma saída de um banco, por exemplo:

```
autores = (('Fernando Pessoa', 17, 'Portugal'),
           ('Carlos Drummond Andrade', 14, 'Brasil'),
```

Agora vamos ordenar:

```
sorted(autores)

#[('Carlos Drummond Andrade', 14, 'Brasil'),
# ('Fernando Pessoa', 17, 'Portugal'),
# ('Nenê Altro', 4, 'Brasil')]
```

Até então, tudo certo. Ele ordenou pelo index 0 da tupla, que nesse caso eram os nomes. Vamos tentar usar a magia da `key` agora:

```
sorted(autores, key=lambda x: x[1])

#[('Nenê Altro', 4, 'Brasil'),
# ('Carlos Drummond Andrade', 14, 'Brasil'),
# ('Fernando Pessoa', 17, 'Portugal')]
```

Nesse caso, a ordenação foi dada pelo index 1, que foi o que nós determinamos na função lambda, vamos tentar mais um caso:

```
sorted(autores, key=lambda x: x[0][-1])

#[('Fernando Pessoa', 17, 'Portugal'),
# ('Carlos Drummond Andrade', 14, 'Brasil'),
# ('Nenê Altro', 4, 'Brasil')]
```

Nesse caso ele fez a ordenação pelo index 0, só que invertido.

Mas não paramos por aí. sorted() ainda tem mais um argumento escondido, `reverse` , que por padrão vem sempre False. Mas podemos pedir o True dele:

```
sorted(autores, key=lambda x: x[0][-1], reverse=True)

#[('Nenê Altro', 4, 'Brasil'),
# ('Carlos Drummond Andrade', 14, 'Brasil'),
# ('Fernando Pessoa', 17, 'Portugal')]
```

Só pra não dizer que não falei das flores. No lugar desse lambda que não é muito bonito, existe uma função bem bonita no módulo `operator` chamada itemgetter():

```
from operator import itemgetter

sorted(autores, key=itemgetter(1))

#[('Nenê Altro', 4, 'Brasil'),
# ('Carlos Drummond Andrade', 14, 'Brasil'),
# ('Fernando Pessoa', 17, 'Portugal')]
```

Mas, teremos alguns momentos a sós com o módulo operator, calma jovenzinho. Uma hora a gente chega lá.

5.7 filter()

Bom, já estamos chegando ao final e filter() não poderia ficar de fora. A única razão pro filter() ser a última função a ser comentada por agora é única e simplesmente por fugir das definições passadas até agora.

filter() não é uma função nem de mapeamento, nem de redução. filter() é uma função de filtragem. Veja bem, só por isso ela ficou por último. Chega de enrolar, vamos ao código:

```
lista = [1, 2, 3, 4, 5]

impares = lambda x: x % 2

filter(impares, lista) # [1, 3, 5]
```

Nem doeu, né? Vale uma lembrança, aparentemente iria retornar só os pares, porém zero é False, lembra? Então o retorno foram os ímpares.

Caso você queira inverter, temos o filterfalse() do módulo itertools, que vai ser tema de outro tópico, mas fica o gostinho:

```
from itertools import filterfalse

lista = [1, 2, 3, 4, 5]
```

```
impares = lambda x: x % 2

filterfalse(impares, lista) # [2, 4]
```

Por hoje é só pessoal. No próximo tópico vamos aprender a criar nossas próprias HOFs.

Retornar «< 4. Funções de redução/mapeamento - Continue lendo »» 6. Funções de ordem superior p.2