python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retornar «< 3. Consumindo iteráveis - Continue lendo »> 5. Funções de ordem superior

4. Funções de redução/mapeamento

Você já achou que estava super avançado, eu sei. Mas porém, contudo, entretanto, todavia, agora que você já sabe como os iteráveis funcionam, nós podemos avançar mais e fazer melhor uso de funções embutidas do python. Como:

- any()
- all()
- len()
- sum()
- zip()
- reversed()
- enumerate()
- map()

Mas primeiro, vamos reduzir tudo a categorias. Como assim?

any(), all(), len() e sum() (vamos lembrar, não estamos importando nada, AINDA), são funções de redução. Dando uma prévia, são funções que consomem iteráveis e retornam um único valor.

Por outro lado, as outras funções (zip(), enumerate(), map() e reversed()) produzem uma nova coleção, porém modificada.

Tá bom, vamos explicar detalhadamente.

4.1 Funções de redução

Funções de redução recebem um iterável e retornam um único elemento. Ok, já disse isso,

4.1.1 any()

Vamos exemplificar com a função any

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

any(lista) # True
```

Viu, nem doeu nada. Mas ok, nós vamos explicar. A função any itera sobre o iterável, que nesse caso é uma lista e verifica se seu valor é `True` . Tá, mas como ela faz isso?

```
for elemento in lista:
    print(elemento, bool(elemento))

# 1 True
# 2 True
# 3 True
# 4 True
```

Ou seja, ele executa a função bool() para cada elemento da lista.

Tá, mas como ele decide o que é Verdadeiro ou Falso?

A documentação do python diz que 'qualquer objeto pode ser testado(...) e que os seguintes valores são considerados falsos:

- None
- False
- Zeros
  - 0 (int)
  - 0.0 (float)
  - 0j (complex)

- Sequências vazias
  - `[]` (lista)
  - `''` (string)
  - `{}` (dicionários/conjuntos)

Se, você acabou de aprender um truque grátis e nem doeu.

```
if var:
    (...)
```

Você nunca mais vai precisar perguntar se algo é True, por exemplo.

```
# um péssimo exemplo
if var == True:
    (...)
```

Bom, agora que você já sabe como o any() verifica os valores, vamos continuar a explicar sua função.

any() retorna True se qualquer elemento da seqûencia for True (usando `bool(elemento)`) para todos os itens da sequência. Ou seja, a única maneira de ele ser False é que todos os elementos no iterável sejam falsos.

```
lista = [[],'', {}, 0]

any(lista) # False
```

```
lista = [[1],'', {}, 0]

any(lista) # True
```

Olha que legal, você acabou de aprender a lidar com sua primeira função de redução. Mas não vamos parar por aí.

Jaber diz: `Por quê??? Agora que eu tinha entendido tudo. Estava tudo tão fácil, por favor. Pare.`

Só mais um pouco, eu sei que você consegue. Vamos lá.

4.1.2 all()

bool() retornarem True.

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

all(lista) # True
```

```
lista = [0, 1, 2, 3, 4, 5]

all(lista) # False
```

Viu, foi tão simples. Agora, vamos a mais uma de redução.

4.1.3 len()

len(), diferente das outras funções, efetua uma soma da quantidade de valores existentes em uma sequência. Vamos tentar implementar um len()?

```
def _len(sequencia):
    """
    Vamos usar _len pois len é uma palavra reservada
    """
    contador = 0
    for x in sequencia:
        contador += 1

    return contador

_len([1,2,3,4]) # 4
```

De maneira bem simplista, o len vai acumular todos os valores existentes em qualquer tipo de sequência, e veremos como o objeto faz isso daqui a pouco. Uma coisa interessante de mencionar é que o len não conta sequências de sequência, `[[1,2,3]]`. A somatória da quantidade de objetos nesse caso é um, pois a lista só armazena uma outra lista dentro de sí.

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

```
len(string) # 6
```

Vale lembrar que para que nossas sequências, sabe, aqueles que a gente mesmo implementa? Elas só precisam ter o método `__len__()`.

O interpretador python pega o nosso objeto do tipo sequência e chama o seu método especial.

Podemos ver, como já foi dito antes, todas as classes de sequência tem que ter um `__iter__()` ou um `__getitem__()`. Mas, se você parar para notar, todas as sequências implementam `__len__()` também. Vamos conferir:

```
'__len__' in dir([1,2,3]) # True
'__len__' in dir('string') # True
'__len__' in dir((1,2,3)) # True
'__len__' in dir({1,2,3}) # True
'__len__' in dir(4) # False
```

Viu só, ele está em toda sequência e não está em objetos que não podem ser iterados, como por exemplo um número inteiro.

4.1.4 sum()

Bom, todas as funções que vimos até agora envolvem alguma operação, uma chamada de função ou algo do gênero (`bool()`, `__len__()`, ...). A função embutida sum() executa uma somatória de todos os elementos da sequência. Por exemplo, se você tiver uma lista de números (int, float, complex, ...) ele vai fazer uma soma de elemento por elemento:

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

# no caso da lista anterior
((((1 + 2)+3)+4)+5) # 15
```

Então ele vai pegar o primeiro elemento e somar com o segundo, o resultado disso vai ser somado com o terceiro valor e assim por diante...

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

```
sum(lista) # 15
```

Uma coisa muito interessantes do sum é que ele não é sempre iniciado em 0, espera. Ele é sempre iniciado em 0, porém o segundo parâmetro de sum pode ser o seu valor inicial. Vamos tentar:

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

sum(lista, 1) # 16
```

Embora sum() faça a soma de valores, ele não usa o `__add__()` do objeto. No caso, várias strings não podem ser concatenadas com sum(). Existe um método das strings mais legal para fazer isso que é o `''.join()` , mas esse não é o nosso foco. Isso só serve para exemplificar que assim como usamos soma de strings `'a' + 'b' # ab` não funciona nesse caso.

sum() só executa a função de soma, tá isso é meio óbvio. Existe uma função muito famosa em python chamada `functools.reduce()` que executa uma função de concatenação igual ao sum só que aplica qualquer tipo de operação, mas vamos falar mais dela quando estivermos falando da biblioteca functools.

Pronto, agora você está preparado para aprender mais um pouco sobre funções de redução. Existem outras funções, mas esse não é momento para falarmos delas. Talvez depois de funções de ordem superior.

4.2 Funções de mapeamento

As funções de mapeamento padrões da biblioteca padrão (zip(), enumerate() ,reversed()) são maneiras super interessantes de trabalhar com iteráveis, vamos lá.

4.2.1 zip() e reversed()

A função zip não é uma função de compressão, como pode parecer. Ela funciona como um zipper, sabe, aquele da sua calça jeans? É tipo isso.

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

list(zip(lista, lista_reversa)) # [(1, 5), (2, 4), (3, 3), (4, 2), (5, 1)]
```

que aprende: programação funcional é te fazer bem, eu disse.

Como podemos ver, a função zip() juntou os elementos de mesmo index de cada sequência e os agrupou em tuplas. Como podemos ver:

```
(lista[0], lista_reversa[0]) # TypeError
```

Jaber diz: `Você não disse que elas eram acessadas pelo index? Não acredito mais em você!`

Calma Jaber, lembra quando eu disse que uma sequência podia ser acessada usando `__iter__()` ? Pense comigo. Como fazer para consumir um iterador preguiçoso? É, é isso mesmo, é assim que chamamos esse tipo de iterador que em python não pode ser acessado por index.

Isso explica quase tudo sobre a linha `list(zip(..., ...))` . Como o zip produz um iterador preguiçoso, não conseguimos acessar nenhum ítem pelo index, mas sim chamando a função embutida next(). Com isso podemos consumir um iterador sem acessar ele pelo index.

Mas por que estamos falando disso outra vez? Todas as nossas funções de mapeamento retornam esse tipo de sequência preguiçosa.

Então, caso a gente queira ver o que tem na sequência, a sua representação, precisamos construir uma sequência não preguiçosa:

```
print(zip(lista, lista_reversa)) # <zip object at xpto>
print(reversed(lista)) # <list_reverseiterator object at xpto>
```

A resposta é exatamente essa. Quer dizer que não existe uma representação 'visível' desse tipo de iterável.

```
'__str__' in reversed(lista) # False
'__repr__' in reversed(lista) # False
```

Não temos um método para representar a saída do nosso objeto. Então, como faço pra ver?

```
# construtor de uma lista
```

```
# construtor de um conjunto
print(set(reversed(lista))) # {5, 4, 3, 2, 1}

# construtor de uma lista
print(tuple(reversed(lista))) # (5, 4, 3, 2, 1)
```

A essa altura do campeonato, você já deve ter notado o que faz a função `reversed()`. Ele devolve um iterável preguiçoso e invertido do que entrou.

```
list(reversed([1,2,3])) # [3, 2, 1]
list(reversed('String')) # ['g', 'n', 'i', 'r', 't', 'S']
```

Então... depois desse grande devaneio, vamos voltar ao zip(). Não esqueci dele, só precisava te ensinar o que ele vai retornar, vai que você se assusta. Não?

Então... o zip retorna um iterável preguiçoso que casa valores de iteráveis.

```
zip([1,2,3], [4,5,6]) # [(1, 4), (2, 5), (3, 6)]

# E você achando que ele só zipava dois? Estava enganado
zip([1,2,3], [4,5,6], [7,8,9]) # [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Não é muito complicado de entender o único problema do zip(), que em todas as sequências tem que ter o mesmo len() (Nossa, você já está entendendo tudo, eu sei). Vamos voltar a falar mais sobre essa quantidade de argumentos infinitos, só que mais tarde. Agora é o enumerate().

4.2.2 enumerate()

A função enumerate() faz uma coisa muito parecida com o zip, só que ele gera a sequência a ser zipada pra você. Olha que legal:

```
list(enumerate([1,2,3])) # [(0, 1), (1, 2), (2, 3)]
```

Olha só que lindo, você não esperava por isso, né? Toda a sua sequência é agrupada com o número do index dela. Fantástico.

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

# iteração
for x in lista:
    print(x)

# 1
# 2
# 3
# 4
# 5
```

Era exatamente esse trecho de código e a gente comparou com o código em C, que usa o index para navegar pelo vetor. Só que às vezes não precisamos de um foreach, a gente quer mesmo é os valores. Então segue uma outra dica bonita:

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

# iteração
for x, y in enumerate(lista):
    print(x, y)

# 0 1
# 1 2
# 2 3
# 3 4
# 4 5
```

É explícito? Não. Mas tem os index, as vezes a gente precisa deles.

Bom, enumerate() é bem simples. Mas temos uma função lá no começo que deixamos de falar. Na verdade vamos só dar uma olhada nela, pois ela é a ponte entre esse e próximo tópico. Enfim, map().

Vamos, falta pouco pra acabar por hoje, você aguenta.

4.2.3 map()

grande segredo será revelado apenas agora.

map() é uma função de MAPeamento, é, essa não foi tão difícil. Porém, ela é um pouco diferente das nossas funções usadas agora. Ela é uma função que chamamos de **Função de ordem superior**. Isso não é um conceito complicado, quer dizer que ela é uma função que recebe uma função como argumento. Existem muitos casos de como usar map(), mas uma coisa é certa, a função usada por map() só pode receber um argumento. Vamos ao código:

```
def func(x):
    """
    Já usamos essa função, lembra?
    """
    return x + 2

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

Também já contei pra você que ela executa a função em todos os elementos da sequência. Tá, você já sabe como usar o map(), eu sei.

Porém você já pensou que falamos hoje da função bool()? Já pensou usar ela em todos os elementos da sequência?

```
map(bool, [0, 1, 2]) # [False, True, True]
```

Era só nesse ponto que eu queria tocar, todas as funções embutidas do python que recebem só um único argumento podem ser usadas com map(). Mas o gostinho das funções que recebem funções ficou na pontinha da língua? Então até o próximo tópico.

Retornar «< 3. Consumindo iteráveis - Continue lendo »> 5. Funções de ordem superior