python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retorne «< 2. Iteráveis e iteradores - Continue lendo »> 4. Funções de redução/mapeamento

3. Consumindo iteráveis

Você achou que o tópico anterior não ia servir de nada, não?

Na verdade, você estava enganado. Muito enganado pra dizer a verdade. Agora que você já está familiarizado com os iteráveis, você já sabe como as funções funcionam. Por que não relacionar tudo e fazer um código mais limpo e funcional? (sacou a piada?)

Retomando um pouco, concordamos anteriormente que o laço “for” (e ele é foreach) itera sobre os elementos:

```
for x in [1, 2, 3, 4, 5]:
    print(x)

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

Porém, esse “for”, aos meus olhos, só vale pra printar coisas, não? Pense comigo. Toda vez que realmente iteramos em alguma sequência, nós queremos os valores, queremos modificá-los, querendo incrementar, criar novos objetos e por ai vai...

Existem muitas maneiras diferentes e até mais eficientes, não briguem comigo, de iterar sobre sequências. Uma delas é uma list comprehensions.

List o quê?

pode ser reduzido a uma única e estilosa linha de código:

```
[elemento for elemento in [1, 2, 3, 4, 5]] # [1, 2, 3, 4, 5]
```

Fala aí, quanta marra, não? Tá, mas isso me retornou a mesma lista que eu tinha antes. Ok, você ainda não entendeu. Vamos tentar outra vez:

```
[elemento + 2 for elemento in [1, 2, 3, 4, 5]] # [3, 4, 5, 6, 7]
```

Agora fez sentido? Nós criamos uma nova lista, partindo de uma lista já existente e que foi processada por uma expressão `elemento + 2` . Ou seja, para cada elemento presente na lista foi aplicada a expressão. Tá, vamos tentar fazer código feio, só pra você entender.

```
lista = [1, 2, 3, 4, 5]
lista_mais_2 = [] # Olha só, uma lista vazia, que medo

for elemento in lista:
    lista_mais_2.append(elemento + 2)
```

Você consegue olhar pra esse código com bons olhos? Imagina que esse “for” está dentro de uma função:

```
lista = [1, 2, 3, 4, 5]
lista_mais_2 = [] # Olha só, uma lista vazia, que medo

def funcao_que_nao_devo_fazer():
    """
    Função que gera muitos efeitos colaterais
    """
    for elemento in lista:
        lista_mais_2.append(elemento + 2)
```

Imagine-se executando essa função que não retorna nada e ainda gera efeitos colaterais sem limites. Imagine executar essa função uma vez, ok. Mas seu programinha vai executar ele 5.415 vezes. Você consegue imaginar o tamanho dessa lista? Eu não.

Podemos deixar a função um pouco melhor, vamos tentar:

```
    """
    Função que gera muitos efeitos colaterais
    """
    lista_mais_2 = []

    for elemento in lista:
        lista_mais_2.append(elemento + 2)

    return lista_mais_2

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

funcao_que_nao_devo_fazer_p2(lista) # [3, 4, 5, 6, 7]
```

Tá, você pode até não ter gostado das list comprehensions:

> Jaber diz: 'Não gosto, achei muito feia a sintaxe'

Ok, Jaber. Mas vamos reconhecer, ela é muito mais simples e eficiente. Em contrapartida, temos a função `map`, que também é mais bonita:

```
def soma_2(x):
    return x + 2

map(soma_2, [1, 2, 3, 4, 5]) # <map at xpto>
```

Olha só, muito mais elegante que a list comprehensions, não? Não. Pense o trabalho de escrever uma função que poderia levemente ser uma expressão. Podemos usar funções anônimas também, vai...

```
map(lambda x: x+2, [1, 2, 3, 4, 5]) # <map at xpto>
```

Olha, temos uma linha também, porém, eu acho a list comprehensions muito mais bonita. Mas, pra você que achou que falar sobre iteráveis e iteradores era besteira, essa sequência que o map retorna é uma sequência padrão `__iter__`, ou seja, ela só pode ser usada uma vez. UHULLL, economizamos memória, olha até que não foi de todo mal ter escrito esse lambda.

querido Jaber.

```
(x+2 for x in [1, 2, 3, 4, 5]) # <generator object <genexpr> at xpto>
```

Acaba-se de reproduzir o estado de amor Pythonico agora. Você foi funcional! Tá, vou parar com as piadas... Mas você usou a cabeça, economizou memória, foi estiloso. Ou seja, um deus do código.

Tá, agora você já sabe o poder que tem nas mãos. Nunca mais declare uma lista vazia para receber appends, combinado?

Agora que você é um entendedor da iterações, vamos complicar um pouco e falar sobre functors, monads e catamorphismo. Tá, mas sem palavras difíceis, os seus coleguinhas não gostam de nomenclaturas haskelianas. Mas você vai poder esnobar seus conhecimentos teóricos sobre Teoria das Categorias. Até o próximo tópico.

Retorne «< 2. Iteráveis e iteradores - Continue lendo »> 4. Funções de redução/mapeamento