python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retornar «< 6. Funções de ordem superior p.2 - Continue lendo »> 8. Closures e contexto de variáveis

7. Nossa primeira biblioteca de funções

O objetivo desse tópico é construir uma gama de funções simples para que possamos exercitar tudo que aprendemos. Vamos fazer 6 funções legais de usar e que podem ajudar em muitos casos. Partiu? Vem comigo.

tail()

Vamos tentar fazer uma função que pode ou não ser uma HOF ou não? Parecida com aquelas do python, que recebem `key` ?

```
def tail(seq, n=1, key=None):
    return seq[-n:] if not key else key(seq[-n:])
```

Pode parecer uma função extremamente simples, mas ela é bem legal. Dá pra conseguir muitos resultados legais com ela:

```
'Por padrão vai retornar só o ultimo'
tail([1,2,3,4]) # [4]

'Aqui usamos n, que nesse caso retorna os ultimos 3 elementos'
tail([1,2,3,4], 3) # [2, 3, 4]

'O resultado reverso'
tail([1,2,3,4], 3, reversed) # <list_reverseiterator at xpto>
```

```
'Uma aplicação complexa'
list(tail([1,2,3,4], 2, partial(map, lambda x: x**2))) # [9, 16]
```

head()

Aqui vamos fazer a mesma coisa, só para o começo da sequência:

```
def head(seq, n=1, key=None):
    return seq[:n] if not key else key(seq[:n])
```

Se você parar pra pensar, a única coisa que mudou aqui foi o slice (`[-n:] -> [:n]`). Então, vamos fazer as mesmas operações:

```
'Por padrão vai retornar só o primeiro'
head([1,2,3,4]) # [1]

'O resultado reverso'
head([1,2,3,4], 3, reversed) # <list_reverseiterator at xpto>

'Uma aplicação complexa'
list(head([1,2,3,4], 2, partial(map, lambda x: x**2))) # [1, 4]
```

take()

A função take é uma função muito legal implementada na lib fn.py e eu gosto muito dela. Vamos ver como ela funciona?

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

Deu pra perceber que ela é bem poderosa, não? Em qualquer sequência podemos pegar n elementos. Uma coisa legal é que podemos consumir parcialmente os geradores e isso é uma coisa linda:

```
def gen():
    yield 1
    yield 2
    yield 3
```

```
list(take(2, gen())) # [1, 2]

func_gen = gen()
list(take(2, func_gen)) # [1, 2]
list(take(2, func_gen)) # [3, 4]
```

Vamos implementar uma função dessa?

```
def take(n, seq):
    """
    Pega n elementos de um sequência

    Args:
        - n: Número de elementos retirados da sequência
        - seq: Sequência da qual os números serão removidos
    """
    _iter = iter(seq)
    for el in range(n):
        yield next(_iter)

list(take(2, [1, 2, 3])) # [1, 2]
```

Agora que você já sabe tudo que foi usado nessa função (iter, yield, next) as coisas ficam tão simples de entender, não?

Vamos testar com um gerador outra vez, usando o mesmo código do exemplo passado:

```
def gen():
    yield 1
    yield 2
    yield 3
    yield 4

list(take(2, gen())) # [1, 2]

func_gen = gen()
list(take(2, func_gen)) # [1, 2]
list(take(2, func_gen)) # [3, 4]
```

exatamente dessa maneira, eles usam uma função mágica do módulo `itertools` chamada `islice` que vamos ver em um tópico futuro dedicado especialmente ao itertools. Mas vai dizer que não ficou simples?

Jaber diz: `Mas a função take() não faz a mesma coisa que a função head()???`

Sim e não. Se você pensar em sequências que aceitam slice (seq[]) ela, head(), é uma função bem legal. Resolve problemas de listas etc... Ela é uma função que pode ser legal em iterações, em pegar o primeiro elemento em alguns casos. A função take() é MUITO mais poderosa, mas também não é bala de prata. Apesar do fato de ela consumir geradores, o que aumenta seu poder, ela também retorna um gerador, o que faz com que não possamos usar o len() dela, embora você saiba o tamanho que pediu no primeiro argumento. O retorno não pode ser acessado por posição e tem que ser construído com alguma outra função, como (list(), tuple(), set() ...). Mas a diferença mais gritante é que take() consome parcialmente iteráves e a função head() é uma HOF que aceita uma função pra processar a cabeça da lista. Com isso, elas exibem retornos completamente diferentes.

drop()

Drop também é uma função que vamos pegar emprestado de fn.py, embora também não vamos usar sua implementação oficial. Vamos ver como ela funciona:

```
list(drop(4, [1,2,3,4,5,6,7,8,9])) # [5, 6, 7, 8, 9]
```

Se você analisar de perto, drop() tem um comportamento muito parecido com a função take(). Ela também consome parcialmente um iterável, porém ela nos retorna valores após o primeiro argumento:

```
drop(n, seq)
```

`n` é o valor que queremos ignorar de `seq`. Em uma chamada `n=5` de uma sequência de 6 valores, só o último valor será retonado. O que faz o comportamento da função ser exatamente inverso ao take() que nos retornaria os primeiros cinco valores. Tá bom, falamos muito, vamos tentar implementar:

```
def drop(n, seq):
    _iter = iter(seq)
    for i, el in enumerate(seq):
```

```
            next(_iter)
        else:
            yield next(_iter)
```

Se você olhar com carinho vai perceber que drop() usa enumerate() para fazer uma iteração explícita (já falamos disso) e usa o valor do index para saber em qual posição da iteração estamos no `if`. Um ponto importante a ser analisado é que os valores de `seq` são percorridos, porém não são devolvidos ao final da iteração.

Um ponto negativo da função drop() é que caso ela seja usada em uma sequência que contém sequências, `[[1], [2]]`, e essas sequências sejam muito grandes (as internas no caso), elas vão ser processadas da mesma maneira internamente na função, embora o resultado seja exatamente o esperado. Para resolver esse ponto, existe na biblioteca `itertools` uma função chamada islice() que torna a saída lazy, porém vamos ver isso em outro tópico e assim chegaremos na versão verdadeira, a de fn.py, de ambas as funções (take() e drop()).

pipe()

pipe() é uma função emprestada da biblioteca toolz, que assim como fn.py trabalha em trazer muitos aspectos funcionais ao python. Existem muitas bibliotecas com esse propósito e vou deixar um vídeo que talvez seja o maior motivador desse curso Programación funcional con Python - Jesús Espino em que Jesús explana entre muitos conceitos da programação funcional e também sobre as bibliotecas existentes. Mas chega de enrolar, vamos entender o conceito agregado no pipe(). Pra quem vem do mundo linux, pipes fazem parte do dia a dia de um bom uso do terminal.

Jaber diz: `Porque dentre todas as funções que falamos até agora, você quer filosofar nessa???`

Tá bom, vamos pensar que o conceito de pipe(), na programação funcional é uma coisa muito importante. Streams são feitos a base de pipes. Mas afinal, o que é um pipe? Não ele não é um cano, mas é, no fundo é.

Imagina pegar o resultado de uma função e atribuir a entrada de outra função? É basicamente isso. O terminal do linux foi citado, mas todo o mundo UNIX usa esse conceito de maneira magnífica. Vamos olhar um exemplo no terminal:

```
# cat - mostra o conteúdo do arquivo na tela, porém mostrar na tela é uma sa
# oi.txt - é um arquivo de texto que contém as linhas (oi Jaber \n oi Eduard
```

```
# grep - é um comando usado para procurar uma determinada String (grep oi, o
```

```
cat oi.txt | grep Jaber

# oi Jaber
```

Dado esse exemplo, o resultado do cat, que jogaria o conteúdo todo da tela, foi jogado para o pipe e ele é responsável por fazer uma conexão entre o `cat` e o `grep`. É um cano entre o cat e o grep. Ele usou tudo que foi processado por cat e entregou ao grep. Fazendo uma conexão. Dito tudo isso, vamos analisar a nossa função pipe(), que não é nossa, foi emprestada de toolz:

```
pipe([1, 2, 3, 4], lambda x: x+2) # TypeError: can only concatenate list (not
```

Jaber diz: `Estou totalmente perdido, não estou entendo mais nada.`

Calma amiguinho, estamos chegando lá.

A função pipe() funciona em um único valor:

```
pipe(4, lambda x: x + 2) # 6
```

Tá, mas ainda assim ela não acrescenta em nada. Vamos tentar complicar um pouco as coisas:

```
soma_2 = lambda x: x + 2
soma_4 = lambda x: x + 4

pipe(4, soma_2, soma_4) # 10
```

Bom, isso também é meio babaca, poderia ser feito com `soma_4(soma_2(4))`. Mas espera. Somos todos hackers, será que não podemos hackear uma função?

Jaber diz: `Aqui é Mr. Robot RAPAZ`

Em um outro momento, vamos explicar isso com muita calma, mas agora só o gostinho pra você entender:

```
soma_2 = partial(map, lambda x: x + 2)

list(pipe([1, 2, 3, 4], soma_2)) # [3, 4, 5, 6]
```

Tá, ok. Isso é um map mais complicado, você não acha?

Jaber diz: `Achei muito ofensivo, deleta.`

A função functools.partial() aplica uma função parcialmente. Mas isso é assunto pra outra hora. Vamos tentar outra vez e eu juro que será a ultima:

```
from functools import partial

soma_2 = partial(map, lambda x: x + 2)
soma_4 = partial(map, lambda x: x + 4)

list(pipe([1, 2, 3, 4], soma_2)) # [3, 4, 5, 6]
list(pipe([1, 2, 3, 4], soma_4)) # [5, 6, 7, 8]

# O pulo do gato

list(pipe([1, 2, 3, 4], soma_2, soma_4)) # [7, 8, 9, 10]
```

A função aplicou soma_2() em toda a sequência e no resultado dessa sequência aplicou soma_4(). Se tivéssemos 10 funções como argumentos ele executaria as 10 funções uma no resultado da outra. Tá, mas é complicado implementar isso? Não:

```
def pipe(seq, *funcs):
    """
    O * nessa função faz com que tudo que for passado após a sequência
      faça parte da lista funcs

    Ele vai iterando na lista de funções e vai jogando o resultado na próxima
      função
    """
    for func in funcs:
        seq = func(seq)
```

Ufa, achei que não íamos acabar nunca essa função, mas no fim deu tudo certo, somos hackers de sequências e somos muito inteligentes.

twice()

Nesse momento eu presumo que você lembre que no tópico passado hackeamos a função aplicando parcialmente outra função. Porque agora vamos usar o que aprendemos, mesmo que ainda tenhamos muito pra aprender em um tópico específico sobre aplicações parciais, para hackear a função por definição.

twice() é uma função bem bacana, ela só executa duas vezes a mesma operação em uma sequência. Não é nenhum bicho de sete cabeças e eu espero que você lembre disso.

```
mul_2 = lambda x: x * 2
twice(10, mul_2) # 40
```

Vamos pensar, temos uma função que multiplica o valor passado por 2, mul_2(). Passamos `10` e ela devolve `20`. Lembra do pipe()? Então, o resultado da primeira rodada foi aplicado a segunda.

`((10 * 2) * 2)` ou seja `40`. Outra vez temos uma função que só trabalha com um único valor. Porém, twice() tem um truque na manga. Não só um, pois twice() evita aquela chamada chata `mul_2(mul_2(10))` o que já é uma coisa muito legal. Mas o que é mais emblemático nessa função nós vamos ver agora:

```
twice([1, 2, 3], mul_2) # [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Jaber: `Nossa, o que é isso? MEU DEUS, SENHOR JESUS, ARREDA DAQUI SATANÁS`

Calma, Jaber. As listas em python implementam um método chamado `__mul__`, o que faz uma multiplicação ser um pouco diferente. Vamos aplicar a só a multiplicação primeiro:

```
[1, 2, 3] * 2 # [1, 2, 3, 1, 2, 3]
```

Isso nada mais é do que repetir a sequência. É `1, 2, 3` duas vezes. O resultado de twice() parece uma coisa maluca porque fez isso duas vezes. O que fez a multiplicação parecer coisa de outro mundo. Só que a função twice() tem uma carta na manga. Um

```
twice([1, 2, 3], mul_2, True) # <map at ...>
```

Jaber: É, parei de entender a muito tempo. Tava bom, mas tava ruim também. Agora parece que piorou.

Calma amiguinho, vamos pedir uma lista disso:

```
list(twice([1, 2, 3], mul_2, True)) # [4, 8, 12]
```

Jaber: Cada vez mais misterioso está ficando esse curso, tô fora.

Você lembra do pipe hackeado? Esse True ativa o hack. Vamos ver o código:

```
def twice(val, func, _iter=False):
    from functools import partial
    if not _iter:
        return func(func(val)) # O retorno normal
    else:
        map_part = partial(map, func)
        return map_part(map_part(val)) # O retorno com hack
```

A função pipe() precisava de uma ajuda para aplicar a função parcial, o que fazia seu estado original sempre trabalhar com um único valor e com auxílio do `functools.partial()` iterar em uma sequência. Já a função twice() traz isso por definição. Vamos dizer que trabalha com a iteração por definição e pode ser ativada com o parâmetro `_iter`. Tirando isso a função twice() não tem nada de especial, ela só executa a mesma função duas vezes ao mesmo elemento e caso `_iter` seja `True` ela executa a função a uma sequência. Um ponto especial de atenção é que a função retorna o mesmo tipo de dado de entrada, caso `_iter` não esteja "ativado" a resposta é um iterável do tipo lazy pois o retorno é saída de uma função map().

Conclusões

Com isso acabamos todo nosso discurso sobre funções de ordem superior (HOFs). Se você quiser parar agora, você pode, mas eu juro que a parte mais legal vai começar a rolar agora. Vamos falar de closures, decoradores e algumas substituições modernas para map() e filter().

variáveis