python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retornar «< 5. Funções de ordem superior - Continue lendo »> 7. Nossa primeira biblioteca de funções

6. Funções de ordem superior p.2

Segundo Steven Lott, podemos criar três tipos de HOFs diferentes:

1. Funções que aceitam funções como argumento
2. Funções que retornam uma função ou uma classe com `__call__`
3. Funções que aceitam e retornam funções (Geralmente são decoradores)

Contudo, vamos exercitar o fato de criar funções de ordem superior apenas. Vamos tentar copiar algumas do escopo e vamos nos divertir. Pronto?

6.1 Funções que aceitam funções

Essa vocês já estão matando no peito, eu sei. Vamos ao código então:

```
def map_clone(func, sequencia):
    """
    Função geradora clone do map

    Args:
        - func: Função que será aplicada a cada elemento da sequência
        - sequencia: Iterável a ser consumido pela função
    """
    for el in sequencia:
        yield func(el)
```

Olha, eu sei que parecia que já tínhamos falado sobre tudo, mas esse é o melhor momento

6.2 Funções geradoras

Funções geradoras 'trocam' o `return` por `yield`. Você só precisa disso para que sua função seja um gerador e retorne lazy como as funções embutidas do python. Só que existe um ponto, as funções retornam iteráveis, ou seja, teremos que usar os mesmos construtores (`list()`, `tuple()`, etc..) de objetos que usamos antes:

```
map_clone(lambda x: x**2, [1, 2, 3]) # <generator object map_clone at 0x7faee

# chamando o construtor list()
list(map_clone(lambda x: x**2, [1, 2, 3])) # [1, 4, 9]
```

Vamos explanar mais a ideia de funções geradoras em outro tópico, mas o entendimento básico é necessário agora. No nosso exemplo com yield usamos um laço for, mas vamos tentar outra abordagem:

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

gen = f_geradora()

next(gen) # 1
next(gen) # 2
next(gen) # 3
next(gen) # 4
next(gen) # StopIteration

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

A função se transforma em um iterável, um comportamento diferente de todas as funções que vimos até agora.

Vamos tentar entender... yield funciona como um break. Tá, vamos tentar de novo. É como

```
def f_geradora():
    print('aqui vai o primeiro valor')
    yield 1 # pausa
    print('Segundo chegando')
    yield 2 # pausa
    print('Terceiro, tá quase acabando')
    yield 3 # pausa
    print('Quarto e último')
    yield 4 # pausa

    # StopIteration

gen = f_geradora()

next(gen)
# aqui vai o primeiro valor
# 1
next(gen)
# Segundo chegando
# 2
next(gen)
# Terceiro, tá quase acabando
# 3
next(gen)
# Quarto e último
# 4
next(gen)
# StopIteration

gen = f_geradora()
list(gen)
# aqui vai o primeiro valor
# Segundo chegando
# Terceiro, tá quase acabando
# Quarto e último
# [1, 2, 3, 4]
```

é para. Olhe, eu juro, a gente vai um pouco mais fundo e voltamos as HQ's.

```
def gen_test():
    for x in [1, 2, 3]:
        print(x)

    yield 'primeiro laço'

    for x in [4, 5, 6]:
        print(x)

    yield 'segundo laço'
list(gen_test())
# 1
# 2
# 3
# 4
# 5
# 6
#['primeiro laço', 'segundo laço']
```

A função executa exatamente o que tem que ser executado e nos retorna apenas o valor do yield. Esse tipo de implementação é a base pra entender as co-rotinas em python, mas isso é assunto pra outra hora e nem vamos falar sobre isso, pois foge do nosso escopo. Porém, você está avisado, pode pesquisar depois sobre a relação de `contextmanager` e `yield` .

Você está o bixão do mundo python já, então vamos complicar esse `yield` usando mais um amiguinho dele chamado `yield from` . Tá, tava tudo legal, mas você vai aprender o que é uma monad agora. Eu juro.

```
def gen():
    for el in [[1, 2, 3], [4, 5, 6], [7, 8, 9]]:
        yield el

def gen_flat():
    for el in [[1, 2, 3], [4, 5, 6], [7, 8, 9]]:
        yield from el

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

```
Seu coração vai chorar agora
"""
list(gen_flat()) # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

CARACAS, como assim? O que aconteceu aqui? Vamos pensar que um map() normal faria o que gen() faz. Mas um flatmap faria o que gen_flat() faz.

Quando usamos yield em um laço, ele retorna cada valor contido em uma sequência, de maneira preguiçosa. Só que a sequência contém outra sequência, ele vai retornar a sequência, pois cada uma é um elemento independente da sequência, mesmo sendo uma sequência. O `yield from` vai retornar um iterável preguiçoso dessa nova sequência, a contida na sequência anterior. Ou seja, ele vai nos retornar uma única sequência. Por isso nome 'flat', é como se a sequência de sequências fosse comprimida em uma única sequência. Para entender isso vou deixar como referência um vídeo do funfunfunctions que fala exatamente sobre isso. Vale a pena.

Agora que você já conhece mais um tipo de função, vamos voltar as nossas HOFs

6.3 Escrevendo nossas próprias HOFs

Como já sabemos e já foi dito exaustivamente, funções são objetos de primeira classe em Python. Já sabemos. Ok.

Então como já entendemos tudo isso, vamos só usar alguns exemplos de funções que recebem funções. OBS: Vamos criar algumas funções meio especialistas agora, isso não é muito bom. Mas serve como base de aprendizado. No proxímo tópico vamos explorar mais funções simples e que são de grande utilidade em muitos contextos, porém...

Vamos trabalhar em outra frente então:

1. Mapear sequências mais complexas

Vamos supor, que temos uma lista de tuplas:

```
# Sim, já vimos algo parecido no tópico anterior

# Hora, minuto, segundo
tempo = [(13, 17, 50),
         (17, 28, 51),
```

```
        (12, 2, 23),]
```

E vamos trabalhar nessa sequência que é um pouco mais complexa do que as que usamos até agora.

Vamos supor que esse horário que está no padrão que vai de 00:00:00 até 24:59:59. E a resposta que nós esperamos é um horário am/pm que vai de 01:00:00 até 12:59:59. Só que a saída terá que ser uma nova tupla, com quatro elementos `(H, M, S, (am ou pm))` . Para isso, a nossa função de mapeamento terá que ser um pouco mais inteligente

```
hora = lambda x: (x[0] % 12, 'pm') if x[0] > 12 else (x[0] % 12, 'am')
formato = lambda x, y: (y[0], x[1], x[2], y[1])


def func_map(seq, *funcs):
    for el in seq:
        yield funcs[1](el, funcs[0](el))

list(func_map(tempo, hora, formato)) # [(1, 17, 50, 'pm'), (5, 28, 51, 'pm'),
```

De brinde você acabou de fazer uma função curry, mas não vamos nos atentar agora a esse detalhe, vamos focar no que aconteceu.

A função anônima `hora()` devolve uma simples tupla com `am` ou `pm` usando aritmética modular. Se for menor que doze ele nos retorna uma tupla com `(hora, 'am')` , se for maior nos retorna `(hora, 'pm')` . Simples não?

Agora a função `formato()` recebe dois argumentos de sequência e só organiza o posicionamento `(hora, minuto, segundo, am_ou_pm)` .

Sobre a função `func_map()` eu inverti a ordem dos argumentos propositalmente pois o `*` só pode ficar depois dos argumentos fixos. Neste caso o `*` não é muito importante, mas serve pra gente acumular 'n' argumentos e eles se tornam uma lista dentro do escopo da função. Por isso chamamos `funcs[0]` e `funcs[1]` .

Não iteramos pela lista de funções, e sim pela sequência. Iterar por uma sequência de funções aproveitando os resultados é um conceito chamado de `streaming` mas vamos dedicar um tópico exclusivamente a isso num futuro próximo.

Jaber diz: `Não ficou muito claro, esse exemplo fugiu das listas básicas. Tô meio perdido`

Vamos um mais simples pra sintetizar:

Essa função vai fazer o clássico algoritmo de map/reduce, sim aquele que conta quantas palavras tem em um texto. Mas nós vamos nos limitar a letras, pois é mais simples de demonstrar.

Vamos entrar com uma string `abacaxi` e a função vai ter que retornar {'a': 3, 'b': 1, 'c': 1, 'x': 1, 'i': 1}.

```
def map_reduce(map_func, reduce_func, seq):
    return reduce_func(map_func(seq))

map_func = lambda x: ((el, 1) for el in x)


def reduce_func(seq):
    dicio = {}
    for chave, val in seq:
        if chave not in dicio:
            dicio[chave] = val
        else:
            dicio[chave] += val

    return dicio

map_reduce(map_func, reduce_func, 'abacaxi') # {'a': 3, 'b': 1, 'c': 1, 'x':
```

Viu, essa foi simples como roubar doce de criança, tá... Ok, roubar doce de criança é bem difícil, mas nossa implementação é bem simples.

A função de mapeamento pega elemento por elemento e o transforma em uma tupla com o valor 1 `(elemento, 1)` e a função de redução tem um dicionário que usa o elemento como chave e o valor 1 vai sendo somado cada vez que ele aparece no dicionário. Então tudo foi mapeado (para transformação em tupla) e foi reduzido em um dicionário. Olha, tudo é muito simples, você já está muito avançado.

agrupamento por palavras.

```
map_reduce(map_func, reduce_func, 'abacaxi verde limão verde como coco verde'

# {'abacaxi': 1, 'verde': 3, 'limão': 1, 'como': 1, 'coco': 1}
```

Olha que mágico, é uma HOF realmente útil... Não, ela não é. Sabe por que?

```
from collections import Counter

Counter('abacaxi') # Counter({'a': 3, 'b': 1, 'c': 1, 'i': 1, 'x': 1})


Counter('abacaxi verde limão verde como coco verde'.split()) # Counter({'abac
```

Tá, vai... A gente tentou e você aprendeu. SUAHSUAHUSHA.

Com isso, no próximo tópico, vamos construir nossa propria lib de HOFs simples e que servem para tudo. Um abraço.

Retornar «< 5. Funções de ordem superior - Continue lendo »> 7. Nossa primeira biblioteca de funções