# python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retornar «< 0. Saindo da zona de conforto - Continue lendo »> 2. Iteráveis e iteradores

# 1. Funções

Como nem tudo são flores, vamos começar do começo e entender algumas características das funções do python (o objeto função) e dar uma revisada básica em alguns conceitos de função só pra gente não se perder no básico depois. Então o primeiro tópico vai se limitar a falar da estrutura básica das funções em python, sem entrar profundamente em cada um dos tópicos. Será uma explanação de código e abrir a cabeça para novas oportunidades de código mais pythonicos e que preferencialmente gere menos efeito colateral. Mas calma, não vamos ensinar a fazer funções, você já está cheio disso.

## 1.1 Funções como objeto de primeira classe

Funções como objeto de primeira classe, são funções que se comportam como qualquer tipo nativo de uma determinada linguagem. Por exemplo:

```
# uma lista

lista = [1, 'str', [1,2], (1,2), {1,2}, {1: 'um'}]
```

Todos esses exemplos são tipos de objetos de primeira classe em Python, mas no caso as funções também são. Como assim? Pode-se passar funções como parâmetro de uma outra função, podemos armazenar funções em variáveis, pode-se definir funções em estruturas de dados:

```
# Funções como objeto de primeira classe

func = lambda x: x # a função anônima, lambda, foi armazenada em uma variável
```

```
    return x + 2

lista = [func, func_2] # a variável que armazena a função foi inserida em uma

lista_2 = [lambda x: x, lambda x: x+1] # aqui as funções foram definidas dent
```

Como é possível notar, em python, as funções podem ser inseridas em qualquer contexto e também geradas em tempo de execução. Com isso nós podemos, além de inserir funções em estruturas, retornar funções, passar funções como parâmetro (HOFs), definir funções dentro de funções(closures) e assim por diante. Caso você tenha aprendido a programar usando uma linguagem em que as funções não são objetos de primeira classe, não se assuste. Isso faz parte da rotina comum do python. Preferencialmente, e quase obrigatoriamente, é melhor fazer funções simples, pequenas e de pouca complexidade para que elas não sofram interferência do meio externo, gerem menos manutenção e o melhor de tudo, possam ser combinadas em outras funções. Então vamos lá!

## 1.2 Funções puras

Funções puras são funções que não sofrem interferência do meio externo. Vamos começar pelo exemplo ruim:

```
valor = 5

def mais_cinco(x):
    return x + valor

assert mais_cinco(5) == 10 # True

valor = 7

assert mais_cinco(5) == 10 # AssertionError
```

`mais_cinco()` é o exemplo claro de uma função que gera efeito colateral. Uma função pura deve funcionar como uma caixa preta, todas as vezes em que o mesmo input for dado nela, ela terá que retornar o mesmo valor. Agora vamos usar o mesmo exemplo, só alterando a linha do return:

```
valor = 5

def mais_cinco(x):
    return x + 5

assert mais_cinco(5) == 10 # True

valor = 7

assert mais_cinco(5) == 10 # True
```

Pode parecer trivial, mas muitas vezes por comodidade deixamos o meio influenciar no comportamento de uma função. Por definição o Python só faz possível, e vamos falar disso em outro tópico, a leitura de variáveis externas. Ou seja, dentro do contexto da função as variáveis externas não podem ser modificadas, mas isso não impede que o contexto externo a modifique. Se você for uma pessoa inteligente como o Jaber deve saber que nunca é uma boa ideia usar valores externos. Mas, caso seja necessário, você pode sobrescrever o valor de uma variável no contexto global usando a palavra reservada `global`. O que deve ficar com uma cara assim:

```
valor = 5

def teste():
    global valor # aqui é feita a definição
    valor = 7

print(valor) # 7
```

Só lembre-se de ser sempre coerente quando fizer isso, as consequências podem ser imprevisíveis. Nessa linha de funções puras e pequeninas, podemos caracterizar, embora isso não as defina, funções de ordem superior, que são funções que recebem uma função como argumento, ou as devolvem, e fazem a chamada das mesmas dentro do contexto da função que a recebeu como parâmetro. Isso resulta em uma composição de funções, o que agrega muito mais valor caso as funções não gerem efeitos colaterais.

## 1.3 Funções de ordem superior (HOFs)

Funções de ordem superior são funções que recebem funções como argumento(s) e/ou

superior, como: `map`, `filter`, `zip` e praticamente todo o módulo functools `import functools` . Porém, nada impede de criarmos novas funções de ordem superior. Um ponto a ser lembrado é que map e filter não tem mais a devida importância em python com a entrada das comprehensions (embora eu as adore), o que nos faz escolher única e exclusivamente por gosto, apesar de comprehensions serem mais legíveis (vamos falar disso em outro contexto), existem muitos casos onde elas ainda fazem sentido. Mas sem me estender muito, vamos ao código:

```
func = lambda x: x+2 # uma função simples, soma mais 2 a qualquer inteiro

def func_mais_2(funcao, valor):
    """
    Executa a função passada por parâmetro e retorna esse valor somado com do

    Ou seja, é uma composição de funções:

    Dado que func(valor) é processado por func_func:
        func_mais_2(func(valor)) == f(g(x))
    """
        return funcao(valor) + 2
```

Um ponto a tocar, e o que eu acho mais bonito, é que a função vai retornar diferentes respostas para o mesmo valor, variando a entrada da função. Nesse caso, dada a entrada de um inteiro ele será somado com 2 e depois com mais dois. Mas, vamos estender este exemplo:

```
func = lambda x: x + 2 # uma função simples, soma mais 2 a qualquer inteiro

def func_mais_2(funcao, valor):
    """
    Função que usamos antes.
    """
        return funcao(valor) + 2

assert func_mais_2(func, 2) == 6 # true

def func_quadrada(val):
```

```
    Deve o valor de entrada ao quadrado.
    """
    return val * val

assert func_mais_2(func_quadrada, 2) == 6 # true
```

### 1.3.1 Um exemplo usando funções embutidas:

Muitas das funções embutidas em python são funções de ordem superior (HOFs) como a função map, que é uma das minhas preferidas. Uma função de map recebe uma função, que recebe um único argumento e devolve para nós uma nova lista com a função aplicada a cada elemento da lista:

```
def func(arg):
    return arg + 2

lista = [2, 1, 0]

list(map(func, lista)) == [4, 3, 2] # true
```

Mas fique tranquilo, falaremos muito mais sobre isso.

## 1.4 `__call__`

Por que falar de classes? Lembre-se, Python é uma linguagem construída em classes, e todos os objetos que podem ser chamados/invocados implementam o método `__call__` :

Em uma função anônima:

```
func = lambda x: x

'__call__' in dir(func) # True
```

Em funções tradicionais:

```
def func(x):
    return x
```

```
__call__ em dir(func) # True
```

Isso quer dizer que podemos gerar classes que se comportam como funções?

SIIIIIM. Chupa Haskell

Essa é uma parte interessante da estrutura de criação do Python a qual veremos mais em outro momento sobre introspecção de funções, mas vale a pena dizer que classes, funções nomeadas, funções anônimas e funções geradoras usam uma base comum para funcionarem, essa é uma das coisas mais bonitas em python e que em certo ponto fere a ortogonalidade da linguagem, pois coisas iguais tem funcionamentos diferentes, mas facilita o aprendizado da linguagem, mas não é nosso foco agora.

## 1.5 Funções geradoras

Embora faremos um tópico extremamente focado em funções geradoras, não custa nada dar uma palinha, não?

Funções geradoras são funções que nos retornam um iterável. Mas ele é lazy (só é computado quando invocado). Para exemplo de uso, muitos conceitos precisam ser esclarecidos antes de entendermos profundamente o que acontece com elas, mas digo logo: são funções lindas <3

Para que uma função seja geradora, em tese, só precisamos trocar o return por yield:

```
def gen(lista):
    for elemento in lista:
        yield elemento

gerador = gen([1, 2, 3, 4, 5])

next(gerador) # 1
next(gerador) # 2
next(gerador) # 3
next(gerador) # 4
next(gerador) # 5
next(gerador) # StopIteration
```

Passando bem por cima, uma função geradora nos retorna um iterável que é preguiçoso.

## 1.6 Funções anônimas (lambda)

Funções anônimas, ou funções lambda, são funções que podem ser declaradas em qualquer contexto. Tá... Todo tipo de função em python pode ser declarada em tempo de execução. Porém funções anônimas podem ser atribuídas a variáveis, podem ser definidas dentro de sequências e declaradas em um argumento de função. Vamos olhar sua sintaxe:

```
lambda argumento: argumento
```

A palavra reservada `lambda` define a função, assim como uma `def`. Porém em uma `def` quase que instintivamente sempre quebramos linha:

```
def func():
  pass
```

Uma das diferenças triviais em python é que as funções anônimas não tem nome. Tá, isso era meio óbvio, mas vamos averiguar:

```
def func():
  pass

func.__name__ # func

lambda_func = lambda arg: arg

lambda_func.__name__ # '<lambda>'
```

O resultado `'<lambda>'` será o mesmo para qualquer função. Isso torna sua depuração praticamente impossível em python. Por isso os usuários de python (e nisso incluo todos os usuários, até aqueles que gostam de funcional) não encorajam o uso de funções lambda a todos os contextos da linguagem. Mas, em funções que aceitam outra funções isso é meio que uma tradição, caso a função (no caso a que executa o código a ser usado pelo lambda) não esteja definida e nem seja reaproveitada em outro contexto. Eu gosto de dizer que lambdas são muito funcionais em aplicações parciais de função. Porém, os lambdas não passam de açúcar sintático em Python, pois não há nada que uma função padrão (definida com `def`), não possa fazer de diferente. Até a introspecção retorna o mesmo resultado:

```
    pass

type(func) # function

lambda_func = lambda arg: arg

type(lambda_func) # function
```

Uma coisa que vale ser lembrada é que funções anônimas em python só executam uma expressão. Ou seja, não podemos usar laços de repetição ( `while` , `for` ), tratamento de exceções ( `try` , `except` , `finally` ). Um simples `if` com uso de `elif` também não pode ser definido. Como sintaticamente só são aceitas expressões, o único uso de um `if` é o ternário:

```
valor_1 if condicao else valor_2
```

O que dentro de um lambda teria essa aparência:

```
func = lambda argumento: argumento + 2 if argumento > 0 else argumento - 2
```

Funções lambda também podem ter múltiplos argumentos, embora seu processamento só possa ocorrer em uma expressão:

```
func = lambda arg_1, arg_2, arg_3: True if sum([arg_1, arg_2, arg_3]) > 7 els
```

Embora essa seja uma explanação inicial sobre as funções anônimas, grande parte dos tópicos fazem uso delas e vamos poder explorar melhor sua infinitude.

Mas por hoje é só e no tópico seguinte vamos discutir, mesmo que superficialmente, iteradores e iteráveis e suas relações com a programação funcional.

Retornar «< 0. Saindo da zona de conforto - Continue lendo »> 2. Iteráveis e iteradores