python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retornar «< 8. Closures e contexto de variáveis - Continue lendo »> 10. Decoradores

9. Usos variados de closures

Vamos entender um pouco mais sobre closures e variar o uso com diversas coisas? Vamos tentar trazer alguns exemplos mais práticos do uso de closures. Mas agora vamos ver alguns modos diferentes dos vistos antes.

9.1 Closures e lambdas

Todas as closures que criamos até agora podem ser substituídas por lambdas, pois as funções internas não passam de simples expressões:

```
def diga_oi(saudacao):
    """
    Função diga_oi com um lambda interno
    """
    return lambda nome: '{} {}'.format(saudacao, nome)

ahoy = diga_oi('Ahoy')
ahoy('Jaber') # 'Ahoy Jaber'
```

Uma coisa legal de usar funções `lambda`, e ao mesmo tempo um ponto negativo desse tipo de utilização, é que não há uma maneira de sobrescrever a variável local usando `nonlocal` pois o lambda só aceita uma expressão de uma linha. Para closures como essa em que o valor deve ser só lido isso pode funcionar bem.

9.2 Métodos em closures

Sim, agora estamos falando de Python e as coisas que o mundo Pythonico nos oferece:

```
    """
    Função diga_oi com um método interno
    """
    def mudar_saudacao(nova_saudacao):
        nonlocal saudacao
        saudacao = nova_saudacao

    diga_oi.mudar_saudacao = mudar_saudacao
    return lambda nome: '{} {}'.format(saudacao, nome)

ahoy = diga_oi('Ahoy')
ahoy('Jaber') # 'Ahoy Jaber'
ahoy.mudar_saudacao('Olá')
ahoy('Jaber') # 'Olá Jaber'
```

Em tempo de execução é possível fazer monkey patch em qualquer tipo de objeto em python, e isso permite que `n` funções possam caber dentro de uma closure. Como tudo pode ser modificado, foi criado um método interno. Manipular closures dessa maneira é mais eficiente do que usar classes, mas não vamos falar sobre isso agora. O que podemos absorver disso é que sem classes ninguém vai começar a pirar, pelo menos em Python não.

9.3 Interagindo com valores `nonlocal`

Nós já fizemos isso com composições de funções, mas como as closures são exatamente o inverso das composições, vamos fazer aqui também. Caso você tenha perdido o momento em que parcialmente consumimos iteráveis, isso foi feito no tópico 7.

Vamos imaginar que nossa closure contém um iterável infinito, ele armazena uma contagem, vamos usar nosso mesmo contador Fail do tópico 8:

```
def contador(inicio):
    var = inicio
    def retorno():
        nonlocal var # implementação de nonlocal
        var += 1
        return var
    return retorno

c = contador(0)
```

```
c() # 3
c() # 4
c() # 5
```

Já fizemos isso, mas vamos tentar incrementar a closure para que ela possa ser consumida lentamente(ou preguiçosamente) e que o retorno possa ser um `range` de valores que acrescentam a contagem de formas diferentes:

```
def contador(inicio, continuo=False):
    """
    inicia nossa closure

    Args:
        - inicio: Define qual o valor de início da contagem
        - continuo: Define se os valores serão continuados ou não
    """
    var = inicio
    def retorno(quantidade):
        """
        Retorna uma quantidade x de valores contínuos a partir do último usad

        Caso seja continua, e a quantidade for (> 1) o retorno vai variar e a
        Caso contrário, retornará um range, mas não alterará o valor inicial
        """
        nonlocal var # implementação de nonlocal
        yield from range(var, var + quantidade) # aqui fica a escolha de retc

        if not continuo:
            var += 1
        else:
            var += quantidade
    return retorno
```

Isso faz com que o comportamento da closure seja diferente e temos duas maneiras de utilizar o seu retorno, mas vale lembrar que ele é sempre lazy e a resposta terá de ser construída ou 'iterada':

Modo 1:

```
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
# Independente de quantas vezes for chamado, a computação só será feita na co
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
list(c(2)) # [0, 1]
list(c(2)) # [1, 2]
list(c(2)) # [2, 3]
```

Modo 2:

```
c = contador(0, True)
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
# Independente de quantas vezes for chamado, a computação só será feita na co
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
c(2) # <generator object contador.<locals>.retorno at 0x0000000>
list(c(2)) # [0, 1]
list(c(2)) # [2, 3]
list(c(2)) # [4, 5]
list(c(2)) # [6, 7]
```

Como você pode ter notado, uma definição na função externa pode sim alterar todo o comportamento da função interna e o uso de nonlocal se fez suficiente para uma única variável, não sendo necessário o uso para as duas.

Jaber diz: `Mas eu poderia criar um método para gerenciar aquele boolean em tempo de execução?`

Sim e isso torna tudo mais lindo, mas eu vou deixar você tentar fazer isso sozinho.

9.4 Closures que recebem funções (ou quase isso)

Esse, embora seja um exemplo óbvio do que vimos até agora, pode ser que não tenha passado na cabeça de ninguém até agora, mas vamos lá:

```
def diga_oi(saudacao, func):
    """
```

```
    return func

ahoy = diga_oi('Ahoy', lambda nome: '{} {}'.format(saudacao, nome))
ahoy('Jaber') # NameError: name 'saudacao' is not defined
```

Jaber diz: `Mas isso faz todo sentido de funcionar, por que não funciona????`

Lembra de todo aquele caso do escopo das variáveis? `saudacao` está definida dentro de `diga_oi()` e isso faz com que ela não exista fora desse contexto que é onde a função `lambda` está sendo definida. Então nesse caso passar funções como argumento que usam valores `nonlocal` não funcionam. Passar funções como argumento que não compartilham atributos é como fazer uma método estático em classes, não serve de muita coisa, só agrega funções em um mesmo lugar.

Vamos entender melhor esse tipo de atribuição ao falar dos decoradores, onde vamos decorar funções. Isso é só um problema de escopo de que precisava ficar nítido. E agora é a hora.

9.5 Decorando funções com closures

Vamos voltar ao básico e fazer somas com números e partiremos de exemplos mais complexos no próximo tópico. Vamos imaginar que temos um caso de uma função em que todos os resultados necessitam ser pares. Tá, isso é muito simples, mas vamos decorar uma função que só executa uma soma simples primeiro:

```
def soma(x, y):
    return x + y
```

Embora essa seja uma função bem simples, não consigo pensar em um exemplo mais direto para que o entendimento dos decoradores não soem como coisa de outro mundo. Vamos criar uma closure que simplesmente executa essa função, ou seja ela vai decorá-la:

```
def eh_par(func):
    """
    Nesse caso a função externa é quem nomeia o decorador.

    A função interna é quem lida com a execução da função e com todo tipo
        de interação com a função decorada.
```

```
    Args:
        - func: Nesse caso func é a função decorada
    """
    def interna(a, b):
        """
        Função que faz a mágica.

        Diferente das closures que vimos até agora, os argumentos a, b
            são os argumentos passados para a função decorada.

        Execução:
            Caso o resultado da soma seja par, o retorno vai ser uma tupla
                como (True, <valor>)
            Caso seja ímpar, vai retornar a tupla com (False, <valor>)
        """
        result = func(a, b)
        if result % 2 == 0:
            return True, result
        else:
            return False, result
    return interna
```

Alguns comentários que gostaria de fazer antes de aplicar de fato o decorador. A função externa, nesse caso 'eh_par' recebe como primeiro argumento a função que será decorada, o que na verdade é aquilo que o decorador de fato faz:

```
eh_par(soma(2, 2)) # (True, 4)
eh_par(soma(3, 2)) # (False, 5)
```

Vamos pensar que a função externa só carrega a função para o escopo da closure (aquilo que vimos como `local` / `nonlocal`). A função interna, nesse caso chamado de 'interna', é quem faz a execução da função de fato, ou seja, ela quem carrega os argumentos da função decorada. É como se a função interna incorporasse a função decorada. Vamos exemplificar com o uso:

```
@eh_par
def soma(x, y):
    return x + y
```

entender. É como se a função 'soma' fosse passada como parâmetro da função 'eh_par' e os argumentos de soma fossem passados para a função 'interna'. Pensando na execução da função 'interna', ela é quem de fato executa a função decorada 'soma'.

Vamos executar:

```
@eh_par
def soma(x, y):
    return x + y

soma(2, 2) # (True, 4)
soma(3, 2) # (False, 5)
```

Agora estamos preparados para entender especialmente os decoradores no próximo tópico, que esse último exemplo tenha sido explicativo e vamos tentar resolver o problema do 'Ahoy' no próximo tópico.

Retornar «< 8. Closures e contexto de variáveis - Continue lendo »> 10. Decoradores