python-funcional

Quase um livro, quase um tutorial, quase qualquer coisa

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

Retornar «< 7. Nossa primeira biblioteca de funções - Continue lendo »> 9. Usos variados de closures

8. Closures e contexto de variáveis

Já passamos funções como argumento, já retornamos funções e até já mudamos o comportamento das mesmas. Mas tem uma coisa que ainda não fizemos: definir uma função no corpo de outra função:

```
def func_0():
    def func_1():
        pass
    pass
```

Jaber diz: `Mas isso é uma classe! Eu sabia que programação funcional era legal, mas realmente tudo é orientação a objetos.`

Tá Jaber, eu entendo seu ponto de vista, mas vou usar uma definição muito boa do livro do Mertz:

“Uma classe são dados com operações anexadas (...) Uma Closure são operações com dados anexados”

Viu? Muda totalmente o modo de ver... Vamos nos explicar de maneira simples e breve, mas vamos entender as closures e fingir que classes não existem, só por alguns minutos.

Vamos imaginar que temos que guardar um valor dentro de uma função. Essa função vai ser uma função que exclusivamente armazena um valor e uma função dentro do seu escopo:

```
    """
    A função externa recebe um valor que só vai existir dentro do seu context
    """
    def func_interna(val_2):
        """
        A função interna recebe um valor e retorna a soma dos dois valores.

        var_1 está acessível tanto no contexto da função interna quanto da
          externa, porém val_2 está acessível somente no contexto da função i
        """
        return val_1 + val_2
    # aqui quando a função externa for chamada ela retorna a função interna e
    return func_interna
```

Vamos usar esse código antes de qualquer explicação mirabolante:

```
var_func = func_externa(5)

var_func(5) # 10
```

Como dá pra notar, a função externa é atribuída a uma variável e essa variável executa a função interna. Parece complicado, mas na verdade é bem simples. Vamos recapitular algumas coisas.

Como em python as funções podem ser definidas em qualquer contexto e armazenadas em qualquer lugar, imagine que a `func_externa()` está sendo atribuída a uma variável. Em um contexto totalmente normal, como fizemos com as funções anônimas até agora. A diferença é um valor, ou uma quantidade `n` de valores, serão passadas no momento da atribuição. Esses valores vão ficar armazenados na função de maneira imutável. Vamos reaproveitar o código com alguns exemplos:

```
soma_um = func_externa(1)
soma_dois = func_externa(2)
soma_tres = func_externa(3)
soma_quarto = func_externa(4)
soma_cinco = func_externa(5)

# vamos somar tudo com 0
```

```
soma_dois(0) # 2
soma_tres(0) # 3
soma_quarto(0) # 4
soma_cinco(0) # 5
```

Bom, agora imagino que tenha ficado um pouco mais claro. Fixamos valores na `func_externa()` e armazenamos em variáveis (`soma_um()` , `soma_dois()` , `soma_tres()` , ...) cada respectiva função mostra o valor inicial da função.

Quando executamos a função `soma_um(n)` qualquer valor que for usado em `n` vai ser somado ao valor fixo na função externa `func_externa(1)` , ou seja, `1` . Vale lembrar que a soma é executada porque esse é o comportamento da função interna `func_interna()` . Vamos tentar outra vez e de uma maneira mais simples:

```
def diga_oi(saudacao):
    """
    A função diga_oi armazenada a sua saudação
    """
    def nome_pessoa(nome):
        """
        A função nome_pessoa retorna a saudação somada ao nome da pessoa
        """
        return '{} {}'.format(saudacao, nome)
    return nome_pessoa

oi_pirata = diga_oi('Ahoy!') # Definição de diga_oi (função externa) fixando

oi_pirata('Eduardo') # Ahoy! Eduardo
oi_pirata('Jaber') # Ahoy! Jaber
oi_pirata('Python') # Ahoy! Python
```

Nesse contexto a função fixou uma variável `Ahoy!` que não pode ser modificada e toda vez que é chamada responde com a união do argumento fixo com a variável passada como parâmetro.

Mas vamos tentar fixar um dicionário de idiomas pra `diga_oi()` ? Por exemplo, a função externa agora não vai receber nenhum argumento, mas vai ter dentro do seu escopo um dicionário com os idiomas:

Uma vantagem de definir esse dicionário dentro da função é que ele vai ficar isolado das variáveis globais, o que faz ele não gerar efeito colateral, e isso é muito positivo:

```
def diga_oi():
    """
    Definição de uma variável (dic) no escopo local da função.
    """

    dic = {'pirata': 'Ahoy', 'ingles':'Hello', 'portugues': 'Olá'}

    def nome_pessoa(idioma, nome):
        """
        Agora a função interna ficou com a responsabilidade dos dois parâmetr

        Mas o dicionário ainda permanece imutável dentro do escopo da chamada
        """
        return '{} {}'.format(dic[idioma], nome)
    return nome_pessoa

saudacoes = diga_oi()

saudacoes('pirata', 'Eduardo') # 'Ahoy Eduardo'
saudacoes('portugues', 'Jaber') # 'Olá Jaber'
saudacoes('ingles', 'Python') # 'Hello Python'
```

8.1 Classes vs closures

Você deve ter percebido que até agora as closures tem dois tipos de comportamentos diferentes, porém a imutabilidade permanece:

1. Fixando parâmetros para padronização de chamadas
2. O escopo da função externa é acessível para a função interna

Aqui você deve ter sacado o esquema de operações com dados. O dado passado à função externa permanece imutável sempre e inacessível a qualquer contexto externo ao da função, ou seja, o dado foi fixado e não pode ser transformado em nenhum outro valor, a não ser que seja feita outra chamada com outro valor. O que deveria ser dito sobre as classes, e que preferi postergar, é o que toca exatamente nesse ponto. Vamos fazer uma

classe **call**()

```
class diga_oi:
    """
    Classe do tipo função
    """
    def __init__(self, saudacao):
        """
        Inicializa a instância do objeto com saudacao
        """
        self.saudacao = saudacao

    def __call__(self, nome):
        """
        __call__ permite que o objeto aplique o operador (),
            seja chamado como função
        """
        return '{} {}'.format(self.saudacao, nome)
```

Esse trecho de código tem o mesmo comportamento da nossa closure pois funciona como uma função, mas o efeito colateral existe:

```
saudacao = diga_oi('Ahoy')
saudacao('eduardo') # 'Ahoy eduardo'

saudacao.saudacao = 'Olá'

saudacao('eduardo') # 'Olá eduardo'
```

Todo atributo em python é mutável, até os métodos podem ser mudados usando monkey patch. Não quero me aprofundar em orientação a objetos pois não é o assunto do curso, mas vamos só modificar essa classe para que ela não gere o efeito colateral:

```
class diga_oi:
    """
    Classe do tipo função
    """
```

```
        Inicializa a instância do objeto com saudacao sendo imutável
        """
        object.__setattr__(self, 'saudacao', saudacao)

    def __setattr__(self, *ignored):
        raise NotImplementedError

    def __call__(self, nome):
        """
        __call__ permite que o objeto aplique o operador (),
            seja chamado como função
        """
        return '{} {}'.format(self.saudacao, nome)
```

`__setattr__()` é o método da classe que 'seta' valores em atributos, se nós quebrarmos a implementação default do Python ele não vai conseguir fazer atribuições, porém, é muito mais complicado que implementar uma closure. Nessa implementação simples para a comparação as closures tem 4 linhas e as classe 7. Pra fazer a mesma coisa, acho muito mais atrativo usar uma closure, não só porque estamos falando de programação funcional, mas pela simplicidade de código ('legibilidade conta'). Mas vamos prosseguir.

8.2 Mutação das variáveis de uma closure

Diferente do que eu disse até agora, os valores podem ser alterados no escopo da função externa, mas temos uma série de limitações. Caso o objeto passado como parâmetro, ou alocado na função externa, seja mutável (listas, dicionários, conjuntos, ...), o objeto pode receber normalmente as modificações, vamos fazer um teste:

```
def contador():
    """
    Função contadora de acessos.

    Internamente mantém uma lista que é definida
        vazia no momento em que é declarada
    """
    lista = []
    def soma():
        """
        Adiciona 1 à lista toda vez que a função é chamada.
```

```
        Retorna a somatória dos valores contidos na lista
        """
        lista.append(1)
        return sum(lista)
    return soma

count = contador()
count() # 1
count() # 2
count() # 3
count() # 4
count() # 5
```

Isso é uma forma porca de fazer um contador, mas ele funciona. O mais importante disso é que a variável `lista` não está sendo modificada. Os valores estão sendo atribuídos à lista porque ela é um objeto mutável. Mas não seria possível, e vamos tentar isso agora, mudar o conteúdo da variável lista:

```
def contador():
    """
    Função contadora de acessos.

    Internamente mantém uma lista que é definida
        vazia no momento em que é declarada
    """
    lista = 0
    def soma():
        """
        Adiciona 1 à lista toda vez que a função é chamada.

        Retorna a somatória dos valores contidos na lista
        """
        lista += 1
        return lista
    return soma

count = contador()
count() # UnboundLocalError: local variable 'lista' referenced before assignm
```

mostrar coisas lindas sobre Python e que podemos usar para fazer funcional com mais empenho e graciosidade.

UnboundLocalError e escopo de variáveis

`UnboundLocalError` é um erro muito comum em Python, e o que me deixa muito surpreso foi isso ter levado tanto tempo pra acontecer nesses nossos contextos de programação funcional. Vamos falar agora sobre escopo de variáveis, porém vale lembrar que esse não é um tópico de programação funcional, e sim de comportamento específico do Python. Se você já sabe sobre tudo isso, você pode pular todos esse tópicos, porém vale a pena, para o melhor entendimento das closures, ficar atento a variáveis livres.

Variáveis globais e locais

Em Python as variáveis podem estar em três contextos. Elas podem ser de escopo global, local ou um 'contexto livre'. Vamos tentar olhar para isso com código:

```
var_0 = 5 # Variável global, pode ser acessada em qualquer contexto

def func():
    """
    func lê a variável global e define uma variável local (var_1)
        e retorna a soma das duas
    """
    var_1 = 5 # aqui temos uma variável do escopo local de uma função
    return var_0 + var_1

print(var_0) # 5
print(var_1) # NameError: name 'var_1' is not defined
```

var_0 está presente em toda a execução, porém var_1 é uma variável do escopo local da função `func()` e fora desse contexto ela simplesmente não existe.

Outro ponto legal disso é que a variável global não pode ser modificada pela função `func()` :

```
var_0 = 5

def func():
    var_0 = 7
```

```
func()
print(var_0) # 5
```

No momento em que tentamos atribuir no escopo de `func()`, criamos uma nova `var_0` dentro desse contexto (vamos falar mais sobre isso em um tópico futuro chamado introspecção de funções), o que faz com que a variável global permaneça a mesma.

Se nós quisermos transformar o valor da variável global dentro do escopo da função, podemos usar a palavra reservada `global`:

```
var_0 = 5

def func():
    global var_0
    var_0 = 7

func()
print(var_0) # 7
```

Agora, como você pode notar, o comportamento é diferente. A palavra `global` disse ao Python que a variável usada aqui é exatamente a do escopo global. Quando o Python procura uma variável (vamos falar mais sobre isso em um tópico futuro chamado introspecção de funções (2)) ele segue uma hierarquia:

```
Existe no meu escopo local? Se sim, use. Caso não
Procure em um escopo mais amplo, se existe, use. Caso não
Procure em um escopo mais amplo ....
....
```

Ele busca até que o contexto seja o global, se a variável não existir ele vai nos retornar `NameError`. Mas isso só vale para leitura. E agora para escrever?

```
Variável criada no escopo, fica no escopo.
```

É simples não? Isso nos ajuda a prevenir muitos erros e não 'assinar' variáveis fora do nosso escopo local e gerar efeitos colaterais bizarros. Exemplo:

```
    """
    Função que recebe uma string com um diretório
    e concatena ele com nosso path atual:

    Exemplo:
    >>> dir_concat('documentos')
    /home/Jaber/Python/documentos
    """
    from os import getcwd
    return '{}/{}'.format(getcwd(), dir)
```

Você notou o que existe de errado nessa função? Ela usa a palavra reservada `dir`, que é uma função do contexto global do python `dir()`, ou seja, se os contextos locais não fossem criados durante todo o resto da execução desse programa não poderíamos fazer uso da função `dir()` pois ela foi sobrescrita.

Variáveis livres

Tá, pode ser que tudo que eu disse é tão óbvio e você faz isso sempre, mas dentro das closures tem um problema relativo a isso e vou voltar no código que iniciei esse bloco:

```
def contador():
    """
    Função contadora de acessos.

    Internamente mantém uma lista que é definida
        vazia no momento em que é declarada
    """
    lista = 0 # Variável local de contador
    def soma():
        """
        Adiciona 1 à lista toda vez que a função é chamada.

        Retorna a somatória dos valores contidos na lista
        """
        lista += 1 # lista está definida no contexto de contador
        return lista
    return soma
```

externo de de `soma()`) só que não podemos usar `global` pois `lista` não é global.

Lembra da ordem da busca por nomes?

```
Faz parte de soma? Não
Faz parte de contador? Sim
```

Então ele usa, só que responde `UnboundLocalError` pois não podemos mudar seu valor. Em Python 3, e só em Python 3, existe a palavra reservada `nonlocal`, que dá o poder de outro escopo mutar o valor fora de seu escopo e que não está no escopo global. Ou seja, resolve o nosso problema e isso é lindo. Existe uma PEP inteira detalhando esse aspecto PEP 3104 pois não quero me alongar nessa explicação, pra simplificar, vou usar o contexto do wikipedia:

```
Uma variável livre é uma variável referenciada em uma função, que não é nem u
```

No caso de nossas closures, é uma variável declarada na função externa, que pode ser acessada pela função interna e não pode ser modificada pela mesma. Porém podemos usar `nonlocal`. Vamos tentar?

```
def contador():
    """
    Função contadora de acessos.

    Internamente mantém uma lista que é definida
        vazia no momento em que é declarada
    """
    var = 0
    def soma():
        """
        Adiciona 1 à lista toda vez que a função é chamada.

        Retorna a somatória dos valores contidos na lista
        """
        nonlocal var # implementação de nonlocal
        var += 1
        return var
    return soma
```

memória com uma lista que pode ter um tamanho nada convencional. Porém isso fere o conceito de imutabilidade, mas ainda é melhor que uma classe porque quem faz acesso ao recurso `var` é somente a função interna sem que ele possa ser transformado pelo escopo externo, como é feito no caso das classes.

Agora vamos olhar para um lado mais avançado das closures, mas você vai conseguir dar mais vazão e usos derivados das mesmas.

Retornar «< 7. Nossa primeira biblioteca de funções - Continue lendo »> 9. Usos variados de closures