08-Funções de Alta Ordem

July 31, 2020

1 Introdução

Funções de alta ordem são funções que podem receber funções como parâmetros e/ou retornar outras funções. Isto é possível pois o Python permite trabalhar com funções da mesma forma que outros valores e objetos. Observe o código abaixo onde uma função, chamada **soma**, é atribuída a uma variável, **minha_soma**, e passada como argumento para a função **print**.

[1]:
```
def soma(a, b):
    return a + b

minha_soma = soma
print(minha_soma)
print(soma)
```

```
<function soma at 0x7f226c0613a0>
<function soma at 0x7f226c0613a0>
```

Observe que estamos imprimindo o objeto da função soma. O valor hexadecimal (iniciado em 0x) corresponde ao endereço de memória do computador no qual o objeto da função está armazenado. Esse valor pode ser diferente em diferentes execuções do programa, uma vez que é o Sistema Operacional o responsável por fazer o armazenamento.

2 Funções que recebem outras funções

As funções **map**, **filter** e **reduce** (vistas anteriormente) são exemplos de funções de alta ordem que recebem uma função como parâmetro.

Como exemplo, vamos implementar nossa própria versão da função **filter**:

[2]:
```
def filtro(fn, it):
    it2 = list()
    for valor in it:
        if fn(valor):
            it2.append(valor)
    return it2

lista1 = [1, 2, 3, 4, 5, 6, 7, 8]
lista2 = filtro(lambda x: x % 2 == 0, lista1)  # mantemos os números pares
print(lista2)
```

```
[2, 4, 6, 8]
```

Observe que diferentemente da função **filter**, nossa função **filtro** já retorna uma lista representada pela variável it2.

Observe também que a variável **fn** é usada como uma função local no escopo da função **filtro**. Atenção como é feita a chamada da função na linha:

```
if fn(valor):
```

Essa linha irá executar a função passada como parâmetro retornando **True** se a variável valor for par ou False se for ímpar. Caso o resultado seja True, valor é inserido em it2.

3 Funções que retornam funções

Uma função, além de poder receber uma função como um argumento, pode retornar outras funções. No exemplo abaixo, a função **criar_verificador(tipo)** retorna uma função dependendo do argumento **tipo**:

[3]:
```
def criar_verificador(tipo):  # função externa
    tipo = tipo.lower()
    if tipo == 'par':
        return lambda x: x % 2 == 0  # função interna/retornada que é anônima
    elif tipo == 'impar':
        return lambda x: x % 2 == 1
    elif tipo == 'positivo':
        return lambda x: x > 0
    elif tipo == 'negativo':
        return lambda x: x < 0
    else:
        return lambda x: False

ver = criar_verificador('par')
print("par, 1:", ver(1))
print("par, 2:", ver(2))
ver = criar_verificador('negativo')
print("negativo, -1:", ver(-1))
print("negativo, 1:", ver(1))
ver = criar_verificador('invalido')
print("invalido, 1:", ver(1))
```

```
par, 1: False
par, 2: True
negativo, -1: True
negativo, 1: False
invalido, 1: False
```

As funções retornadas podem ser do tipo **lambda** (anônimas) como mostrado acima ou também serem funções com nome, veja abaixo:

[4]:
```
def somador():  # função externa
    def soma(a, b):  # função interna
        return a + b
    return soma  # a função externa retorna a interna

fn = somador()
print(fn(2, 3))
```

```
5
```

3.1 Escopo da função retornada

Uma característica muito útil é que a função retornada (interna) "lembra" os argumentos da função externa.

[5]:
```
def criar_multiplicador(x):  # função externa
    def mult(y):  # função interna
        return x * y
    return mult

dobro = criar_multiplicador(2)
triplo = criar_multiplicador(3)

print(dobro(4))
print(triplo(4))
```

```
8
12
```

No exemplo acima, duas instâncias da função **mult** são criadas. A primeira instância, atribuída a variável **dobro**, tem o valor de x como 2. A segunda, atribuída a variável **triplo**, tem o valor de x como 3.

Observe como a função **mult** acessa tanto o valor do seu próprio argumento, **y**, como o argumento **x** da função **criar_multiplicador**.

Veja abaixo que a função retornada (interna) tem o acesso não somente ao valores do argumento da função externa, mas também a qualquer variável local à função externa:

[6]:
```
def criar_fn(x):  # função externa
    x_quad = x**2
    def fn(a, b):  # função interna
        return a*x_quad + b*x
    return fn

fn2 = criar_fn(2)
fn3 = criar_fn(3)

print(fn2(2, 3))
print(fn3(2, 3))
```

```
14
27
```

4 Bibliografia

John Hunt. **A Beginners Guide to Python 3 Programming**. Undergraduate Topics in Computer Science. Springer, 2019.

Kent D. Lee. **Python Programming Fundamentals**. Undergraduate Topics in Computer Science. Springer, 2014.