Aula 6 - Definindo funções

Karl Jan Clinckspoor

15 de julho de 2018

Sumário

1 Introdução

É muito útil consolidar código num pacote, de modo que ele possa ser chamado externamente, e o usuário não ter que saber dos detalhes do funcionamento do código. Para isso, são definidas funções. Como visto na primeira aula, funções podem receber de 0 até arbitrariamente quantos argumentos sejam necessários.

2 Declaração

De modo a criar uma função, é necessário utilizar a palavra chave **def**, seguido do nome da função e, entre parênteses, os nomes dos parâmetros. Depois, vem um ':' significando o início de um bloco de código. Coloque o código e, caso deseje que a função retorne um valor, utilize a palavra chave **return** seguido das variáveis que a função irá retornar.

```
In [2]: def soma_1(x):
            soma = x + 1
            return soma

        def _soma_1(x): # mais compacto
            return x + 1
```

```
soma_1(2)
```

```
Out[2]: 3
```

Funções não precisam ter argumentos, nem retornar alguma coisa. Por padrão, uma expressão *return None* é adicionado no final, caso não esteja presente, mas não se preocupe com esse detalhe.

```
In [3]: def dizer_oi():
            nome = input('Qual é o seu nome? ')
            print(f'Oi {nome}.')

        dizer_oi()
```

```
Qual é o seu nome? Karl
Oi Karl.
```

3 Exercício - Carregando dados

A função a seguir carrega os dados da aula 4. A função recebe o nome do arquivo a ser carregado como argumento e retorna a primeira e a segunda colunas do arquivo

```
In [4]: def carregar_dados(arquivo):
            fhand = open(arquivo, 'r')
            x = []
            y = []

            for line in fhand:
                if line.startswith('x'):
                    continue
                temp_x, temp_y = line.split(' ')
                temp_x = float(temp_x)
                temp_y = float(temp_y)

                x.append(temp_x)
                y.append(temp_y)

            fhand.close()
            return x, y

        x, y = carregar_dados('Consolidação1.txt')
```

Veja que, definido uma vez, é possível chamar a função em qualquer outra parte do código. Caso haja um erro na função, é necessário alterar o código em um só lugar, que as mudanças se propagarão para o resto do código.

Os dados que estão na pasta 'dados-1' possuem quase a mesma formatação do arquivo 'Consolidação1.txt'. Que alterações à função seriam necessárias para que a função seja compatível com ambos os arquivos?

Dicas:

1. Estude as diferenças de formatação dos dois arquivos, abrindo-os em um editor de texto.
2. Para saber em que linha a função está errando, coloque um print(line) imediatamente antes de onde o erro ocorre, de acordo com o Traceback. Essa técnica é a forma mais simples de *debugging*. Para funções mais complexas, recomenda-se utilizar IDEs como PyCharm e seu excelente debugger.
3. Utilizar funções de strings para converter a linha em algo que a função já consiga lidar. Considere o que fazer com os espaços, e o que as funções *replace*, *rstrip* e *lstrip* fazem.

```
In [5]: import glob
        dados = glob.glob('./dados-1/*.dat')
        x, y = carregar_dados(dados[0])


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-5-9e3763a6d681> in <module>()
          1 import glob
          2 dados = glob.glob('./dados-1/*.dat')
    ----> 3 x, y = carregar_dados(dados[0])


        <ipython-input-4-274c871b1a09> in carregar_dados(arquivo)
          7         if line.startswith('x'):
          8             continue
    ----> 9         temp_x, temp_y = line.split(' ')
         10         temp_x = float(temp_x)
         11         temp_y = float(temp_y)


        ValueError: too many values to unpack (expected 2)
```

```
In [ ]: %load ./respostas/Func_carregar.py
```

4 Parâmetros opcionais

É possível que as funções tenham parâmetros opcionais, fornecidos como keywords, como já visto. Para implementar isso, só coloque as keywords **depois** dos parâmetros obrigatórios.

```
In [6]: def somar(x, num=1):
            return x + num

        print(somar(5))
        print(somar(5, num=5))
        # Como não há ambiguidade sobre quem é o segundo argumento, não é estritamente
        # necessário fornecer num como um keyword. Tome cuidado, pois isso é menos
```

```
        # claro no código.
        print(somar(5, 5))
```

```
6
10
10
```

5 Documentação

Quando você criar funções mais complexas, é de bom grado colocar uma *docstring*, ou um texto que descreve o que a função faz, o que ela retorna e como ela funciona, em termos gerais. É útil tanto para quem for utilizar seu código, quanto o seu eu futuro. Isso é feito colocando-se, logo na primeira linha, uma string com três ". Esse tipo de string pode ser quebrada por linhas novas, e só termina mesmo quando encontra o "final.

```
In [7]: def func_complexa(*args, **kwargs):
            """Essa função é muito muito complexa, pois calcula e retorna o significado da
            vida, do universo, e tudo mais"""
            return 42

        help(func_complexa)
        func_complexa(37)
```

```
Help on function func_complexa in module __main__:

func_complexa(*args, **kwargs)
    Essa função é muito muito complexa, pois calcula e retorna o significado da
    vida, do universo, e tudo mais
```

```
Out[7]: 42
```

Se você se lembra da Aula 3, o operador asterisco, `*`, desempacota os valores de listas e `**` desempacota os valores de dicionários. Logo, é possível criar funções que aceitam um número arbitrário de argumentos obrigatórios e opcionais utilizando o `*args, **kwargs`.

6 Lidando com erros e exceções

De vez em quando é possível que você encontre um erro que acontece em situações específicas. Por exemplo, imagine que você faça uma função de ajuste, e informe vários arquivos para a função ajustar. É possível que ela não consiga convergir para algum parâmetro em algum desses arquivos. Ao invés de você ter que criar uma exceção específica, ou remover o arquivo da lista manualmente, é possível utilizar um conjunto de **try**, **except** para lidar com esse erro.

A sintaxe é a seguinte:

```
try:
    <código>
except:
    <mais código>
```

Para o except, é muito pouco recomendado que seja um Except sozinho, sem nada, pois isso mascara erros que podem aparecer no seu programa. Ao invés disso, é útil colocar os nomes dos erros. Você pode não saber, mas você viu várias mensagens de erro informando o nome do erro. Por exemplo, *TypeError* e *ValueError*. É possível colocar uma exceção genérica, e depois imprimir o que a exceção fala.

É possível também mandar os seus próprios erros utilizando a palavra chave *raise*. Veja os exemplos.

```
In [9]: try:
            int('abc')
        except ValueError:
            print('Não é possível converter strings para ints!')

        try:
            int('abc')
        except Exception as e:
            print(f'Mensagem: {e}')

        def quero_raise():
            raise ValueError('Agora!')

        try:
            quero_raise()
        except ValueError as e:
            print(f'{e}')
```

```
Não é possível converter strings para ints!
Mensagem: invalid literal for int() with base 10: 'abc'
Agora!
```

7 Escopo de variáveis

Um tópico um pouco chato de se estudar, e que pode dar muita dor de cabeça, é o de escopo de variáveis. Basicamente, escopo significa a região que uma variável pode ser chamada. Há variáveis *globais* e *locais*. Variáveis locais não podem ser chamadas por funções fora de seu escopo, já variáveis globais podem ser chamadas de qualquer lugar. Veja o exemplo.

```
In [17]: global1 = '--- global1 é uma variável global.'

         def func1():
             local1 = '--- local1 é uma variável local'
```

```
        print('Func1 consegue ver global1?', global1)
        print('Func1 consegue ver local1?', local1)

    def func2():
        print("Func2 consegue ver local1?", local1)

    def func3():
        local2 = '--- local2 também é uma variável local'
        def ninho1():
            print(local2, global1)
        ninho1()

    func1()
    func3()
    try:
        func2()
    except NameError:
        print('Func2 não consegue ver a variável local1, pois ela foi devinida somente em
              ' apesar de func1 já ter sido executado')
```

```
Func1 consegue ver global1? --- global1 é uma variável global.
Func1 consegue ver local1? --- local1 é uma variável local
--- local2 também é uma variável local --- global1 é uma variável global.
Func2 não consegue ver a variável local1, pois ela foi devinida somente em func1 apesar de func
```

8 Exercício - calculando derivadas numéricas

Frequentemente no tratamento de dados, devemos calcular a derivada de um conjunto numérico. Porém, ao invés de funções, temos somente valores discretos, então é necessário criar aproximações para derivadas. Neste exercício, crie as seguintes funções.

1. **der_simples**: criar uma lista com as derivadas utilizando a inclinação entre o ponto *n* e o ponto *n+1*.

$$d_n = \frac{y_{n+1} - y_n}{x_{n+1} - x_n}$$

Obs: Preste atenção com os valores nas pontas! Quantas derivadas existem para um conjunto de n pontos? Essa função é equivalente a *np.diff*.

2. **der_média**: criar uma lista de derivadas utilizando a média das inclinações de *n* com *n-1* e *n+1*.

$$d_n = \frac{\left(\frac{y_{n+1}-y_n}{x_{n+1}-x_n} + \frac{y_n-y_{n-1}}{x_n-x_{n-1}}\right)}{2}$$

Obs: Novamente preste atenção com os valores das pontas. Quantas derivadas existem para um conjunto de n pontos? Essa função é equivalente à função de derivada inclusa no *Origin*.

```
In [18]: x = list(range(0, 20, 1))
         y = [i ** 2 - 10 * i + 10 for i in x]

         def der_simples(x, y):
             pass

         def der_media(x, y):
             pass
```

```
In [ ]: %load ./respostas/Exercicio_derivadas.py
```