UFCD 10793 - Fundamentos de Python

Notebook 07 - Funções e módulos - Python

Sílvia Martins

AEFHP 2022

Funções

Uma função é um bloco de código que só é executado quando é chamado.

Podemos passar dados,(pelos parâmetros), para uma função.

Uma função pode retornar dados como resultado, para isso usa-se a instrução `return` .

Em Python uma função é definida usando a palavra chave *def* .

Sintaxe:

```
def <nome_da_funcao> (p1,p2, ..., pn):
    <instruções>
    [return <qq_coisa>]
```

A `chamada de uma função` é efetuada por uma instrução com o nome da função seguida dos parêntesis curvos, onde poderão ou não ser introduzidos valores para parâmetros.
`

Criação de uma função, de nome *ola*, sem parametros e que não devolve nenhum dado, seguida da sua chamada:

In [2]:

```
#criação da função 'ola'
def ola ():
  print('Olá mundo!')
#chamada da função 'ola'
ola()
```

```
Olá mundo!
```

Criação de uma função com um parâmetro (x) e que devolve o valor armazenado em x, seguida da sua chamada e passagem de valor (3) para a mesma:

In [4]:

```
#criação da função 'func'
def func(x):
    x=x*2
    return x
#chamada da função 'func'
func(3)
```

Out[4]:

```
6
```

In [5]:

```
a=5
func(a),a
print(func(a),a)
a=[4,5,6]
(func(a),a)  #tuplo
```

```
10 5
```

Out[5]:

```
([4, 5, 6, 4, 5, 6], [4, 5, 6])
```

Como esperado, a invocação da função não altera o valor do objeto a dado como *input*. Mas o caso poderia ser (inesperadamente?) diferente se a função usasse métodos de uma classe mutável, como é o caso das listas, como no exemplo abaixo:

In [8]:

```
def nfunc (a,b):
    a.append(b)
    return a
lista=[1,2,3]
valor=5
print(nfunc(lista,valor))
print(lista)
```

```
[1, 2, 3, 5]
[1, 2, 3, 5]
```

Na linguagem Python é como se o mecanismo de passagem de parâmetros a procedimentos funcionasse por *valor*, no caso de tipos imutáveis, e por *referência*, no caso de tipos mutáveis.
Em rigor, a passagem de parâmetros dá-se sempre por valor.

Definição de funções com argumentos variáveis
Quando não sabemos quantos argumentos serão passados para a função, adicionamos um * antes do nome do parâmetro na definição da função
=> parâmetro `*args` . Desta forma, a função receberá os argumentos em forma de *tuplo*.

Exemplos:

Função com pelo menos um argumento, mas potecialmente muitos, que devolve a sua soma:

In [19]:

```
#criação da função
def f(x,*ys):
    for y in ys:
        x=x+y
    return x
#chamada da função
f(5,6,7,10)
```

Out[19]:

```
28
```

O parâmetro *ys da definição de f é interpretado como um tuplo.
Podemos inclusive invocar a função a partir de tuplos de argumentos.

In [20]:

```
t=(1,2,3)
f(5,*t)
```

Out[20]:

```
11
```

In [21]:

```
def maior_30(*args):    #args é um nome usado pelos programadores mas poderia usar-se outro
    print(args)
    print(type(args))

    for num in args:
        if num > 30:
            print(num)

maior_30(10, 20, 30, 40, 50, 60)
```

```
(10, 20, 30, 40, 50, 60)
<class 'tuple'>
40
50
60
```

Podemos seleccionar de uma lista de números inteiros, todos os números pares.
Considera a função `pares` que recebe como argumento uma lista `w` de números inteiros e devolve a lista contendo todos os números pares em `w`.

In [22]:

```
def pares(w):
    return [y for y in w if y%2==0]

a=[1, 2, 10, 44, 33, 1, 2, 45, 71]
pares(a)
```

Out[22]:

```
[2, 10, 44, 2]
```

A função seguinte permite determinar quantos números pares existem numa lista.

In [23]:

```
def contapares(w):
    return len([y for y in w if y%2==0])
contapares(a)
```

Out[23]:

```
4
```

Nos exemplos seguintes generalizamos a função `contapares` para contar outros tipos de elementos, nos exemplos seguintes serão números inteiros:

In [10]:

```
def par(n):
    return n%2==0
print(par(2))
print(par(3))
def impar(n):
    return n%2!=0
print(impar(2))
print(impar(3))
```

```
True
False
False
True
```

Argumentos arbitrários de palavras-chave **__kwargs__.

Para criar uma função com número variado de parâmetros nomeados, utiliza-se o parâmetro **__kwargs__. Desta forma, todos os dados passados à função serão guardados nessa variável ***__kwargs__*, em formato de um dicionário.

Exemplos:

In [25]:

```
def dados_pessoa(**kwargs):
    print(type(kwargs))

    for chave, valor in kwargs.items():
        print(f"{chave}: {valor}")

dados_pessoa(nome='João', idade=35, profissao='programador')
```

```
<class 'dict'>
nome: João
idade: 35
profissao: programador
```

In [18]:

```
# concatenar
def concatenar(**palavras):
    result = ""
    for arg in palavras.values():
        result += arg
    return result

print(concatenar(a="O ", b="Python ", c="é ", d="formidável", e="!"))
```

```
O Python é formidável!
```

In [32]:

```
def pessoa(a,**kwargs):
    a.append(kwargs.items())

pessoal=[]
print ("INSIRA OS DADOS DOS 3 FUNCIONÁRIOS")
for i in range(3):
    n=input("Nome->")
    id=int(input("Idade->"))
    pessoa(pessoal,nome=n, idade=id)
print(pessoal)
```

```
INSIRA OS DADOS DOS 3 FUNCIONÁRIOS
Nome->s
Idade->3
[dict_items([('nome', 's'), ('idade', 3)])]
Nome->f
Idade->4
[dict_items([('nome', 's'), ('idade', 3)]), dict_items([('nome', 'f'), ('ida
de', 4)])]
Nome->g
Idade->9
[dict_items([('nome', 's'), ('idade', 3)]), dict_items([('nome', 'f'), ('ida
de', 4)]), dict_items([('nome', 'g'), ('idade', 9)])]
[dict_items([('nome', 's'), ('idade', 3)]), dict_items([('nome', 'f'), ('ida
de', 4)]), dict_items([('nome', 'g'), ('idade', 9)])]
```

A ordem dos argumentos deve ser respeitada na definição: primeiro os parâmetros fixos, depois `*args` e finalmente `**kwargs` .

Palavra reservada *global* : aplicada a um nome dentro duma função, esse nome manterá o seu valor fora da função.

No exemplo abaixo, o nome w foi precedido pela palavra reservada global, pelo que, será mantido o seu valor fora do procedimento, apesar de w representar um valor do tipo int que não é iterável.

In [3]:

```
def escrevepercentagem(tot,val):
    global w
    w=int(100*val/tot)
    print(w,"%")

escrevepercentagem(25,11)
print(w,"%")
```

```
44 %
44 %
```

Funções recursivas

A recursividade é a capacidade que uma linguagem tem de permitir que uma função possa invocar-se a ela própria. É uma forma de implementar um ciclo através de chamadas sucessivas à mesma função.

REGRAS PARA A ESCRITA DE FUNÇÕES RECURSIVAS

- 1º – A 1ª instrução de uma função recursiva deve ser a implementação do critério de paragem, isto é, qual a condição ou condições que se devem verificar para que a função pare se se invocar a ela própria.
- 2º – Só depois de escrito o critério de paragem é que se deverá escrever a chamada recursiva da função, sempre relativa a um subconjunto.
  Por exemplo potência de base X e expoente N, a sua chamada recursiva é realizada com N-1 e critério de paragem será para expoente 0):

potencia(3, 4) =3 * potencia(3,3)

potencia(3, 3) =3 * potencia(3,2)

potencia(3, 2) =3 * potencia(3,1)

potencia(3, 1) =3 * potencia(3,0)

potencia(3, 0) = 1

Exemplos:

In [4]:

```
def potencia(x,N):
    if N!=0 :
        return x*potencia(x,N-1)
    else:
        return 1
print("insira a base e o expoente ")
x=int(input("Base->"))
N=int(input("Expoente->"))
print(f"POTÊNCIA DE BASE {x} E EXPOENTE {N}: {potencia(x,N)}")
```

```
insira a base e o expoente
Base->2
Expoente->4
POTÊNCIA DE BASE 2 E EXPOENTE 4: 16
```

Módulos

Os módulos são bibliotecas existentes na linguagem Python ou que criadas pelo programador. As que são criadas pelo programador têm extensão `py` e agregam conjuntos de funções definidas pelo mesmo.
Para que as funções definidas no módulo possam ser utilizadas basta fazer a sua importação:
`import <nome_da_biblio> as <alias>`

Exercício(criação de biblioteca de funções):

Cria uma biblioteca com o nome biblio.py que contenha as funções definidas acima : **f**, **contapares**, **pares** e **impares**. Utiliza a biblioteca biblio, fazendo uso de cada uma das suas funções.

Bibliografia

Caleiro.C., Ramos.J.(2016). Notebook 03 - Listas e outros tipos iteráveis Dep. Matemática -IST. Lisboa: IST
Vasconcelos, J. (2015). Python - Algoritmia e Programação Web. Lisboa: FCA
Matthes. E.(2019). Python Crash Course - A Hands-On, Project-Based Introduction to Programming. San Francisco: USA
https://www.w3schools.com/python/python_functions.asp (https://www.w3schools.com/python/python_functions.asp)
https://pythonacademy.com.br/blog/funcoes-em-python (https://pythonacademy.com.br/blog/funcoes-em-python) https://docs.python.org/pt-br/3/library/stdtypes.html#additional-methods-on-integer-types (https://docs.python.org/pt-br/3/library/stdtypes.html#additional-methods-on-integer-types)
https://www.w3schools.com/python/python_strings_methods.asp (https://www.w3schools.com/python/python_strings_methods.asp)
https://realpython.com/python-kwargs-and-args/#using-the-python-args-variable-in-function-definitions (https://realpython.com/python-kwargs-and-args/#using-the-python-args-variable-in-function-definitions)
Carvalho. Adelaide. (2021). Práticas de Python - Algoritmia e programação. Lisboa: FCA