Computadores e Programação

2009–2010 1º semestre — aula 3

Helmut Wolters

helmut@coimbra.lip.pt

2009-10-09

Funções: Definição

- As funções em Python são de certa forma semelhantes às funções da linguagem C e às funções/procedimentos do Pascal.
  Sintaxe:

```
def nome([arg1,arg2,...,argn]):
    <statements>
    return [value1,value2,...valuen]
```

- Notas:
  - a instrução `def` cria um objecto do tipo função e atribui-lhe um nome
  - `return` devolve o(s) resultado(s) para a instrução que chamou a função
  - para chamar uma função, depois de criado este objecto, basta invocá-la pelo seu nome.

Funções: Utilidade

- Reutilização do código
- Decomposição de uma tarefa complexa numa série de tarefas mais simples
- Facilita a leitura e futuras modificações do programa

Funções: Exemplos

- Exemplo 1:

```
>>> def produto(x,y):
>>>     return x*y
>>>
>>> print produto(2,3)
>>> 6
>>> z =2
>>> y = produto(9,z)
>>> print y
>>> 18
>>> frase = 'aa'
>>> z = produto(frase,2)
>>> print z
>>> 'aaaa'
```

Funções: Exemplos

- Exemplo 2:

```
>>> def intersect(s1,s2):
>>>     res = []
>>>     for x in seq1:
>>>         if x in seq2:
>>>             res.append(x)
>>>     return res
>>>
>>> intersect([1,2,3],[1,7,2])
>>> [1,2]
```

Funções: Documentação

- As funções definidas em Python podem conter texto de ajuda e documentação.
  Sintaxe:

```
def nome([arg1,arg2,...,argn]):
    'Texto de ajuda e documentação.'
    <statements>
    return [value1,value2,...valuen]
```

- A ajuda pode ser obtida com o comando:

```
>>>help(nome)
```

Funções: Documentação

- Este comando pode ainda ser usado para obter ajuda sobre as funções pré-definidas do Python:

```
>>>import math
>>>help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(...)
    sqrt(x)

    Return the square root of x.
```

Funções: passagem de parâmetros

- A passagem de argumentos é feita por referência, ou seja por atribuição de um nome no espaço de nomes local da função. Isto significa que objectos imutáveis não podem ser alterados dentro de uma função (nem fora dela), mas os objectos mutáveis (ex. listas, dicionários) sim.
- Por exemplo com a função:

```
def try_to_change(n):
    n='A'
```

obter-se-à:

```
>>>name='B'
>>>try_to_change(name)
>>>name
'B'
```

Funções: passagem de parâmetros

- Já com a função:

```
def change(n):
    n[0]='A'
```

obter-se-à:

```
>>>name=['B','C']
>>>change(name)
>>>name
['A','C']
```

Funções: Tipos de parâmetros

- Parâmetros posicionais
- Parâmetros chave-valor
- Parâmetros chave-valor e valores por omissão
- Listas arbitárias de parâmetros

Funções: Parâmetros posicionais

- Todos os exemplos que vimos até agora são de parâmetros do tipo posicional.
- A ordem pela qual estes parâmetros são dados à função é importante. Com a função:

```
def operation(x,y,z):
return x/y+z
```

teremos por certo que operation(x,y,z) é em geral diferente de operation(y,x,z).

Funções: Parâmetros tipo chave-valor

- Suponha a função:

```
def hello(greeting,name)
    print greeting+", "+name+"!"
```

- Usando parâmetros do tipo posicional pode-se normalmente chamar esta função como:

```
>>> hello('Hello','world')
Hello, world!
```

- Pode contudo ser importante designar o nome das variáveis:

```
>>> hello(greeting='Hello',name='world')
Hello, world!
```

  Dizemos que estamos a usar uma chamada com parâmetros do tipo chave-valor.

- Neste caso a ordem não interessa:

```
>>> hello(name='world',greeting='Hello')
Hello, world!
```

Funções: Parâmetros chave-valor e valores por omissão

- O tipo de parâmetros chave-valor pode também ser usado na definição das funções para especificar valores que o parâmetro pode tomar se o utilizador não o utilizar (valores por omissão). Exemplo:

```
def add_tax(x,iva=20):
...     """Adds the IVA tax (given as percent)
...        to value x.
...     """
...     return x*(1+iva/100.)
...
print add_tax(100)
120.0
print add_tax(100,5)
105.0
print add_tax(100,iva=7)
107.0
```

Funções: Parâmetros chave-valor e valores por omissão

- Quando omitido, o argumento `iva` toma o valor por omissão (20%). O exemplo ilustra ainda a possibilidade de chamar explicitamente por nome um argumento da função, o que ajuda, por vezes, à legibilidade do programa. Chama-se a atenção para o seguinte: na chamada de uma função, a seguir a um argumento dado explicitamente por nome, todos os outros argumentos que lhe seguem também terão que ser dados da mesma forma, pelo que a seguinte chamada da função é ilegal:

```
print add_tax(buy=200,5)
SyntaxError: non-keyword arg after keyword arg
```

Notar a *string* de documentação a seguir à instrução `def`.

Funções: Parâmetros chave-valor e valores por omissão

- É ainda de notar que a resolução dos nomes dados por omissão na definição de uma função só é avaliada *uma* vez, precisamente quando a função é definida, e não quando a função é chamada; este detalhe pode dar origem a erros subtis!

```
taxa = 20
def add_tax(x,iva=taxa):
...     return x*(1+iva/100.)
...
print add_tax(100)
120.0
taxa = 21
print add_tax(100)
120.0
```

Funções: Parâmetros chave-valor e valores por omissão

- Um outro resultado inesperado:

```
def save(x,L=[]):
...     L.append(x)
...     print L
save(1)
[1]
save(2)
[1, 2]
L =['a','b']
save(3,L)
['a', 'b', 3]
save(3)
[1, 2, 3]
```

Funções: Listas arbitrárias de parâmetros

- É por vezes desejável escrever funções que possam aceitar um número variável de argumentos. Para o efeito, os símbolos `*args` e `**keyw` utilizados como argumentos de uma função (na sua definição ou chamada) significam um número arbitrário de argumentos posicionais (`*args`) e de argumentos com valor por defeito (`**keyw`). No primeiro caso `args` é uma tupla que contém os argumentos posicionais e no segundo `keyw` um dicionário contendo os argumentos e os valores por omissão.

Funções: Listas arbitrárias de parâmetros

```
def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if not x in res:
                res.append(x)
    return res

a = [1,2,3]; b =[2,7,8]; c =[0,3]
print union(a,b,c)
[1, 2, 3, 7, 8, 0]
L = [a,b,c]
print union(*L)
[1, 2, 3, 7, 8, 0]
```

Funções: Recursão

- Em Python, tal como na maioria das linguagens de programação modernas, as funções podem ser definidas de forma recursiva (i.e., que incluem na própria definição referências à própria função.
- Por exemplo, a seguinte função calcula $n!$ de forma recursiva.

```
def fact(n):
...     if n == 1:
...         return 1
...     else:
...         return n*fact(n-1)
...
fact(3)
6
```

Funções: Recursão

- A sequência de passos que o programa efectua quando executa esta função é o seguinte:

```
fact(3) -> 3*fact(2) (reserva memória no stack)
fact(2) -> 2*fact(1) (reserva memória no stack)
fact(1) -> 1
volta para trás:
fact(2) <- 2*1
fact(3) <- 3*2*1 = 6
```

Funções: Recursão

- A definição de funções de forma recursiva é expressiva, mas a sua execução (em termos de tempo de cálculo e, nalguns casos, de gasto de memória intermédia — *stack*) pode ser dispendiosa. Há contudo algoritmos que podem ser expressos de forma muita simples recursivamente e que, expressos de outra forma são muito complicados. Há também muitos casos em que o tempo de execução da forma recursiva de uma função não lhe é desfavorável. . .