Introdução ao uso de funções em Python: variáveis locais, globais e aninhamento de funções

Nesta seção apresento um conceito mais "sutil" de programação, para entendê-lo é essencial que você copie os códigos apresentados, "cole" em seu editor preferido e os teste até que esteja tudo claro. Experimente alterar os exemplos e criar outros: grave-os e os *rode*. **Para prender programação é preciso programar!**

Se você precisar de um *algoritmo* com um objetivo muito bem definido (e.g. um *algoritmo* para encontrar o menor valor dentro uma *lista* dada) é melhor **encapsulá-lo** em um bloco de modo a poder invocá-lo sempre que precisar desse *algoritmo*. Geralmente todas as linguagens de programação possibilita esse *encapsulamento* e isso é feito na forma de uma função (ou de um procedimento). A grande vantagem dessa abordagem é deixar o seu código melhor organizado, além de permitir que a função seja utilizada em outros trechos do código sem a necessidade de codificá-la novamente!

Para ampliar as vantagens do uso de *funções* as *linguagens de programação* permitem: o *aninhamento* de funções (funções dentro de funções) e a declaração de *variáveis locais*.

Cuidado com a sintaxe de função (erro de recorrência não desejada)

Um erro que as vezes ocorre é usar de modo equivocado o conceito de *devolução* do resultado, provocando uma recorrência infinita. Como a cada chamada da função todas suas variáveis locais precisam ser alocadas na memória, a cada chamada um novo espaço de memória é reservado, logo, em algum tempo o seu computador **não mais "aguentará"** e lançará uma mensagem de erro indicando algo como **recursão excedeu profundidade máxima**. O código abaixo ilustra esse engano, experimente copiá-lo e colá-lo em seu editor preferido e rode-o.

```
def fat (n) :
  f = 1; i = 2;
  while (i<=n) :
    f *= i;
    i += 1;
  return fat(n); # Erro aqui! RuntimeError: maximum recursion depth exceeded

print(fat(5));
```

Cód. 1. Exemplo de engano implicando em recorrência "infinita".

Para corrigir o engano do código 1, troque a linha de *devolução* por `return f;`. Isso eliminará as recorrências. Por outro lado, é possível implementar uma versão correta e recursiva, veja este texto sobre recursividade.

Sobre o aninhamento de funções na linguagem *Python*

Em várias linguagens de programação, *Python* em particular, é possível declarar função dentro de função (**aninhar**). Por exemplo, dentro de uma função de nome *funcao2*, pode-se declarar

outra função de nome *funcao3*. O resultado é que, dentro da primeira função, pode-se invocar a segunda, mas em nenhum outro local do código seria possível invocar esssa função *funcao3*.

Mas vale destacar que esse recurso só é efetivamente útil se a função interna (*funcao3*) só fizer sentido dentro da função que a contém. Pois em caso contrário, você poderia invocar a função interna em outros contextos. Um exemplo dessa situação é ilustrado no parágrafo seguinte, a função *fatorial* poderia ficar melhor se declarada fora da função *combinacao*.

Um exemplo simples de função contendo outra função, poderia ser o caso do cálculo da combinação de *n*, tomados *k*-a-*k*: $C_{n,k} = n!/(k!(n-k)!)$. Como para o calculo de $C_{n,k}$ é necessário invocar o cálculo do fatorial 3 vezes, podemos desejar fazer um implementação que deixe essa dependência clara, definindo uma função de nome *combinacao* e, dentro dela, declarar a função *fatorial*. Vide exemplo 1.

Porém, como anteriormente comentado, como a função *fatorial* tem potencial de aplicação mais amplo, ela ficaria melhor se declarada **fora** da função *combinacao* (vide a crítica a esse organização de código logo após o exemplo 1).

Exemplo 1. Declaração aninhada de funções.

```
def combinacao (n, k) : # combinacao de n, k-a-k
  def fatorial (n) : # fatorial de n (supondo n natural)
    fat = 1;
    for i in range(2, n+1) :
      fat *= i;
    return fat;
  return fatorial(n) / (fatorial(k)*fatorial(n-k));

def main () :
  n = 5; k = 3;
  print("Combinacao %d, %d-a-%d = %d" % (n, k, k, combinacao(n,k)));
  # Problema de declarar 'fatorial()' dentro de 'combinacao()': a linha
abaixo daria erro!
  # Erro: NameError: global name 'fatorial' is not defined
  # print("fat(0)=%d, fat(1)=%d, fat(2)=%d, fat(3)=%d" % (fatorial(0),
fatorial(1), fatorial(2), fatorial(3)));

main();
```

Note que o código acima apresenta a linha comentada `print("fat(0)=%d, fat(1)=%d, fat(2)=%d, fat(3)=%d" % (fatorial(0), fatorial(1), fatorial(2), fatorial(3)));`. A razão é que ela resultaria erro, pois a função *fatorial* foi declarada dentro da função *combinacao* e portanto só pode ser usando (invocada) dentro dessa segunda!

Uma vez que o uso do cálculo de fatorial é muito comum e pensando que o código acima seria parte de um maior, então provavelmente o desenho de código usado não é bom, pois implicaria em precisarmos definir outra função *fatorial*, essa fora do contexto da função *combinacao*. Nesse sentido, poderia ser melhor usar o código (pensando que ele seja o início de um sistema maior) do exemplo a seguir.

Exemplo 2. Declaração de funções sem aninhamento.

```
def fatorial (n) : # fatorial de n (supondo n natural)
    fat = 1;
    for i in range(2, n+1) :
      fat *= i;
    return fat;
def combinacao (n, k) : # combinacao de n, k-a-k
  return fatorial(n) / (fatorial(k)*fatorial(n-k));

def main () :
  n = 5; k = 3;
  print("Combinacao %d, %d-a-%d = %d" % (n, k, k, combinacao(n,k)));
  # Note que agora pode-se invocar 'fatorial()' dentro da 'main'
  print("fat(0)=%d, fat(1)=%d, fat(2)=%d, fat(3)=%d" % (fatorial(0),
fatorial(1), fatorial(2), fatorial(3)));

main();
```

Variáveis globais e locais

Outro conceito importante relacionado com funções é o de declaração de variáveis. Assim, uma variável é **local** se declarada dentro de uma função qualquer, quer dizer, essa variável será "conhecida" (poderá ser usada) apenas dentro dessa função. Por exemplo, no *exemplo 2* acima, a variável *fat* é conhecida apenas dentro da função *fatorial*, portanto seu uso deve-se restringir a esse contexto.

Por outro lado, pode-se usar o mesmo nome para variáveis que estão em contextos distintos. Por exemplo, no código do *exemplo 1*, se fosse necessário poderíamos usar o nome de variável local *i* dentro da função *combinacao* e não haveria qualquer colisão com a variável *i* de *fatorial*.

Entretanto é necessário um critério que elimine *ambiguidades*, isto é, que seja possível identificar unicamente cada uma das variáries. Para isso usa-se o **princípio da proximidade** (ou da **localidade**), quer dizer, ao usar uma variável considera-se como sua declarção aquela mais próxima. Por isso, que poderiamos usar *i* tanto em *combinacao*, quanto em *fatorial*,

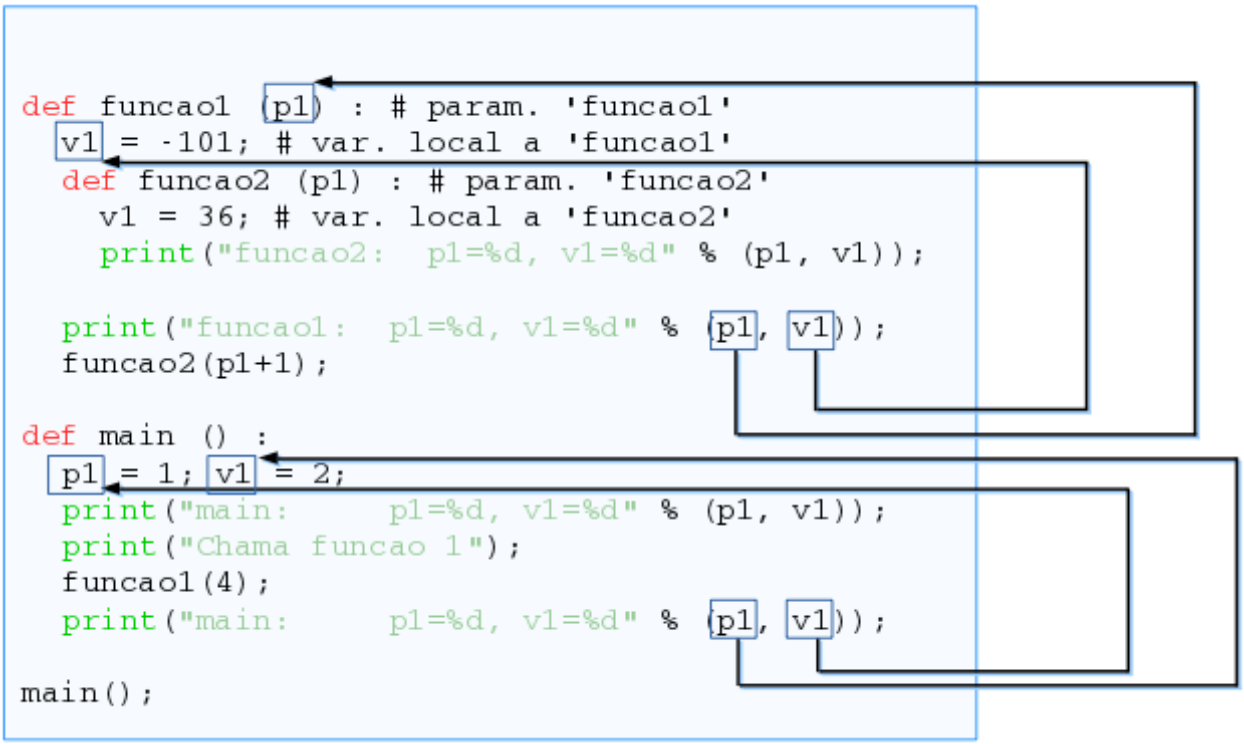

Fig. 1. Código ilustrando escopo de variáveis: `v1` e `v2` correspondem à declaração mais próxima.

main: p1=1, v1=2 Chama funcao 1 funcao1: p1=4, v1=-101 funcao2: p1=5, v1=36 main: p1=1, v1=2*Cód. 1. Saídas para o código da figura 1.*

Já uma variável **global** pode ser acessada em qualquer ponto do código. Mas por essa mesma razão deve ser evitada, sendo necessária apenas para casos excepcionais, sempre com um nome significativo, que evite qualquer confusão.

Em *Python* existem dois modos para se declarar uma variável **global**, ou seja, uma variável que pode ser acessada em qualquer ponto do código (em qualquer função). Ela pode estar no código "principal" (quer dizer não estar dentro de qualquer função) ou sendo declarada dentro de uma função usando a diretiva especial **global**, como ilustrado no próximo exemplo.

Exemplo 3. Declaração de variável global.

```
var_global1 = "essa variavel e' uma global (1)";
def funcao1 () :
  print("funcao1: var_global1 = %s" % var_global1);

def funcao2 () :
  global var_global2; # define 'var_global2' como global
  var_global2 = "essa variavel e' outra global (2)"; # atribui um primeiro
valor a 'var_global2'
  print("funcao2: var_global1 = %s" % var_global1);
  print("funcao2: var_global2 = %s" % var_global2);

def main () :
  print("main   : var_global1 = %s" % var_global1);
```

```
  #Erro: print("main   : var_global2 = %s" % var_global2); # NAO pode usar
essa linha pois 'var_global2' ainda NAO foi definida
  funcao1();
  funcao2();
  print("main   : var_global2 = %s" % var_global2); # NAO pode usar essa
linha pois 'var_global2' ainda NAO foi definida

main();
```

Aninhando funções e variáveis locais e globais

Deve-se notar que é possível aninhar tantas funções quantas forem necessárias e o mesmo para variáveis locais. O exemplo abaixo ilustra esses aninhamentos e o princípio da proximidade. Examine-o com atenção, copie-o em seu computador, utilizando um interpretador *Python* e altere seu código até que esteja certo de ter assimilado os conceitos.

Exemplo 4. Declaração de variáveis locais, usando globais e aninhamento de funções.

```
glob1 = "glob1: variavel global, conhecida em qualquer local";

def funcao1 () :
  loc1 = "loc1: conhecida apenas dentro da funcao 'funcao1'";
  glob1 = "funcao1.glob1: declarada dentro de 'funcao1' => uma variavel local
`a funcao 'funcao1'";
  print("funcao1: loc1 = %s" % loc1); # note que: pode haver m
  print("funcao1: glob1 = %s\n" % glob1);

def funcao2 (param1) :
  def funcao3 () : # essa funcao so' e' conhecida dentro de 'funcao2'!
    loc1 = "loc1: conhecida apenas dentro da funcao 'funcao3' - NAO sou a
'funcao1.loc1' e nem a 'funcao2.loc1'!";
    glob1 = "funcao3.glob1: declarada dentro de 'funcao3' => var. local `a
funcao 'funcao3' - NAO e' global 'glob1' e nem 'funcao1.glob1'!";
    print("funcao3: loc1 = %s" % loc1); # note que: e' local com outra com
esse nome; "principio do mais proximo" em acao!
    print("funcao3: loc2 = %s" % loc2); # note que: e' local `a funcao
'funcao2' - variavel declarada fora de funcao3!
    print("funcao3: glob1 = %s" % glob1);
  global glob2;
  glob2 = "glob2: variavel global, conhecida em qualquer local (mas declarada
em 'funcao2')"; # mas depois desse comando ser executado!
  loc1 = "loc1: conhecida apenas dentro da funcao 'funcao2 - NAO sou a
'funcao1.loc1'!'";
  loc2 = "loc2: conhecida apenas dentro da funcao 'funcao2'";
  print("funcao2: param1 = %s" % param1);
  print("funcao2: loc1 = %s" % loc1); # note que: e' local com outra com esse
nome; "principio do mais proximo" em acao!
  print("funcao2: loc2 = %s" % loc2); # note que: e' local
  print("funcao2: glob1 = %s" % glob1);
  print("funcao2: chama funcao3\n");
  funcao3();
  print("funcao2: chama funcao1\n");
  funcao1();

def funcao4 () :
```

```
  print("funcao4: glob2 = %s" % glob2);

def main () :
  loc1 = "loc1: conhecida apenas dentro da funcao 'main'";
  print("main   : loc1 = %s" % loc1); # note que: e' local com outra com esse
nome; "principio do mais proximo" em acao!

  print("main   : glob1 = %s" % glob1);

  # A linha abaixo NAO pode ser usada, pois apesar de glob2 ser declarada
como global (usando a diretiva 'global')
  # isso e' feito apenas dentro da funcao 'funcao2', entao ela so' ficara'
disponivel apos 'funcao2' ser executada!
  # Se tirar o comentario da linha abaixo, resultaria o erro: NameError:
global name 'glob2' is not defined
  # print("main   : glob2 = %s" % glob2);

  print("main   : chama funcao1\n");
  funcao1();
  print("main   : chama funcao2\n");
  funcao2("parametro efetivo passado para a funcao 'funcao2'");

  # Atencao: como a funcao 'funcao3' foi declarada dentro da funcao
'funcao2', entao ela so' pode se invocada em 'funcao2'
  # Por exemplo, a linha abaixo resultaria no erro: NameError: global name
'funcao3' is not defined
  # funcao3();

  # Mas nesse ponto a global 'glob2' ja' foi definida
  print("main   : glob2 = %s" % glob2);

  print("main   : chama funcao4\n");
  funcao4(); # note que 'funcao4' tambem pode usar a global 'glob2'
(declarada dentro da 'funcao2'

main();
```

Leônidas de Oliveira Brandão
http://line.ime.usp.br

Alterações:

Alterações