Introdução ao uso de funções em Python: organização do código

Nesta seção apresentamos as questões essenciais para o uso de *funções* (ou *procedimentos* - funções sem valores devolvidos): usar funções para **não** repetir códigos e declarar as funções de modo bem organizado.
Experimente alterar os exemplos e criar outros: grave-os, compile-os e os *rode*. **Para aprender programação é preciso programar!**

1. Sintaxe de funções em *Python*

Uma *função matemática* deve ter o *domínio* e o *contra-domínio*, uma função em *Python* segue o mesmo padrão, mas com uma sintaxe bem específica. O *domínio* é definido pelo tipo de dado dos *parâmetros* da *função* e o *contra-domínio* é definido pelo tipo de dado de devolução, por meio do comando `return`. A sintaxe de uma *função* em *Python* está ilustrada no código 1.

```
def nome_funcao (lista_parametros_com_tipos) :
  # o codigo (atuando sobre a "lista_parametros_com_tipos" e
devolvendo algo do "tipo"
  # geralmente deve-se terminar a funcao com comando "return
'expressao_simples'"
```

Cód. 1. Estrutura de uma função em Python.

Um exemplo em devemos utilizar o conceito de *função* é se precisarmos computar a combinação de *N, k-a-k*, ou seja, considerando *N* elementos diferentes, quantos são os conjuntos distintos com exatamente *k* elementos que é possível formar (vide aqui uma explicação, com exemplos das loterias). Isso está ilustrado no código 2, lançando mão de definir uma função fatorial.

Python

```
1  def fat (n) : # define funcao com 1 parametro
2    ft = 1; i=1;
3    while (i < n) :
4      i = i + 1; # poderiamos tambem usar "i += 1"
5      ft = ft * i; # poderiamos tambem usar "ft *= i"
6    return ft;
7    # final da funcao, entao na linha seguinte usar menos
8  espaco de "indentacao"
9  ...
10 # supondo existir neste contexto variaveis: N e k
11 print("Combinacao = %f" % (fat(N) / (fat(k) * fat(N-k))));
```

Cód. 2. Exemplo de código para computar combinação de n, k-a-k, com uma função para fatorial.

2. Procure organizar bem o seu código

A linguagem **Python**, por ser interpretada, permite péssimas organizações de código, procure fortemente se disciplinar para produzir um código direto e evitando **"bagunça-lo"**. Para ficar

claro o que **dever ser evitado** apresentarei um *contra-exemplo*, ou seja um **código "bagunçado"** e a seguir uma versão equivalente **bem organizada**. O objetivo do código *contra-exemplo* é bastante artificial (para ficar curto): implementar um programa que soma 1 e depois soma 2, usando funções.

Contra-exemplo. Exemplo de código péssimamente organizado para somar 1 e somar 2 (**não** faça desse modo!).

```
a = int(input()); # comando da linha principal de execucao
def soma1 (a) : # OPA! Declaracao de funcao em meio 'a linha...
  print("%d" % (a+1)); # Hum, NAO e' bom imprimir aqui, deveria
deixar para quem precisa de "soma1"
soma1(a); # mais um comando na linha principal de execucao
def soma2 (b): # Xi, mais uma funcao no meio da linha
principal!
  print("%d" % (b+2)); # Novamente, deveria deixar a impressao
para quem precisa de "soma2"
soma2(a); # novamente linha principal
```

Cód. 3. Exemplo de código mal estruturado, misturando declarações e imprimindo dentro de função.

O código acima emprega dois péssimos hábitos de organização, um foi misturar a declaração de função em meio ao *fluxo de execução* (esse é o pior deles e infelizmente a linguagem **Python** o permite...). O segundo foi imprimir a resposta do valor calculado dentro das funções, o que impediria a função a ser usada em outro local que não precise da impressão, é sempre preferível deixar para que invocou a função a tarefa de fornecer a saída adequada.

A segunda falha é mais sutil, pois dependendo do código a ser implementado, pode ser necessário a função ter algumas saídas de dados (e eventualmente alguma entrada). Então a regra é usar o *bom senso*, se a função tiver um objetivo muito bem definido de computar determinados valores, implemente-a sem qualquer impressão.

Agora examinemos um código muito melhor organizado, que faz a mesmo que o anterior.

Exemplo. Exemplo de código equivalente ao 3 mas este bem organizado.

```
# Primeiro declare TODAS as funcoes, primeiro as que nao
depende de outras
def soma1 (a) : # Essa funcao NAO depende de qualquer outra
  return a+1;
def soma2 (b): # Essa poderia depender de "soma1", entao
precisa vir depois de "soma1"
  return b+2; # Novamente, deveria deixar para quem precisa de
"soma2"
  # Note que com esse codigo limpo, podemos implementar "soma2"
a partir de "soma1"
  # trocando o "return b+2" pelo seguinte
```

```
  # return soma1(a)+1;
# Agora vem o "fluxo principal", com o codigo que invoca as
funcoes anteriormente declaradas
a =int(input()); # comando da linha principal de execucao
print("%d" % (soma1(a)); # mais um comando na linha principal
de execucao
print("%d" % (soma2(a)); # novo comando na linha principal
```

Cód. 4. Exemplo de código bem estruturado e que produz o mesmo resultado que o código do contra-exemplo anterior.

3. Não repetir código: use função/procedimento

Uma das grandes utilidades do emprego de *funções* é simplificar o código, evitando-se que um mesmo código aparece mais de uma vez em seu programa. O código 1 ilustra isso, se não fosse utilizado funções, seria necessário usar mais duas cópias do código para fatorial (uma para computar *fat(N)*, uma para *fat(k)* e uma última para *fat(N-k)*). Isso resultará um código "menor" (menos linhas), além de ser mais fácil mantê-lo funcionando.

Outra vantagem não tão óbvia em código pequenos é reduzir erros e facilitar a manutenção do código.
Uma vez que implementou-se uma função (procedimento) que realiza determinada tarefa, sem erros, pode-se sempre utilizar o mesmo código (apenas copiando-o para o novo programa. Isso reduz a chance de novos erros.
Além disso, por estar bem organizado, em um ponto, a manutenção do código também é simplificada (basta examinar e eventualmente alterar essa função).

Ao declarar funções em *Python*, deve-se organizar o código declarando as funções **por ordem de chamada**, ou seja, se seu código usualmente invoca primeiro `f1(.)`, depois `f2(.)` e por último `f3(.)`, então deveria declarar nessa ordem (primeiro `f1(.)` e assim por diante).
Entretanto, existem códigos maiores e mais complexos, nesses casos tente **organizar por assunto**, deixando próximas as funções que tratam do mesmo *assunto*.
Se existir algum caso de *ciclo*, uma função `f1(.)` que invoca `f2(.)` e `f2(.)` que invoca `f1(.)`, será necessário declarar os protótipos das funções. Por exemplo, se ambas as funções devolverem inteiro e a primeira tiver um único parâmetro inteiro e a segunda um inteiro e um flutuante, então deve-se declarar no início os seguintes *protótipos*: `int f1(int);` `int f2(int,float);`.

Outro item que pode ser necessário em códigos mais complexos é declarar uma função dentro do contexto de outra função, como poderia ser o caso de fazer uma função para *combinação*, tendo internamente sua função *fatorial*. Embora nesse caso seria prefarível deixar o *fatorial* disponível para todos os demais códigos (ou seja, como no código 2). Para examinar mais detalhes sobre o *aninhamento* de funções clique aqui.

Para ilustrar as duas vantagens do uso de funções apresentamos um código violando as duas regras, declaração desordenada das funções e repetição de código (que deveria estar em funções). Assim, o código 5 deve ser entendido como **não** se deve programar!

Contra-exemplo: **não** codifique duplicando código (use função) e não misture declarações

```
# Exemplo de codigo ruim:
# 1. Codigo repetido que deveria estar em uma
funcao/procedimento
# 2. Funcoes misturadas com a linha de execucao

def modos () :
  # 1. NAO duplique codigo,
  N = 3; k = 2; # usar tambem variaveis: modo, fat, fatN,
fatk, i
  modo = 2;
  if (modo==1) : # fat(N)
    fat = 1; # inicio: codigo que deveria estar como
funcao
    for i in range(2,N+1) :
      fat *= i;
    print("Fatorial de %d = %d" % (N, fat));

  elif (modo==2) : # fat(N)/fat(k);
    fatN = 1; # inicio: codigo duplicado (1)
    for i in range(2, N+1) :
      fatN *= i;
    fatk = 1; # inicio: codigo duplicado (2)
    for i in range(2, k+1) :
      fatk *= i;
    print("Arranjo %d, %d-a-%d = %d  (%d,%d)" % (N, k, k,
(fatN/fatk), fatN, fatk));
    # final do 'elif'
  # final da funcao 'modos(.)', reduza "indentacao" na
linha seguinte

# A execucao comeca por aqui, isso esta' MUITO RUIM!
Deveria estar apos declarar TODAS as funcoes!
modos();

# 2. NAO misture execucao principal com funcoes! Primeiro
declare "soma(.)"
#    Dependendo das configuracoes de seu compilador, nao
compilaria ou resultaria na advertencia dos 3 comentarios
abaixo
def soma (a, b) : #
  return a+b; #
  # final 'soma(.)'
```

```
# Mais bagunca, mais mistura da execucao principal em meio
'as declaracoes (NAO faca isso)
a = 5; b = 6;
print("%d + %d = %d" % (a,b, soma(a,b)));
```

Cód. 5. Exemplo de código para computar combinação de n, k-a-k, com uma função para fatorial, mas ordem incoveniente.

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

Alterações