

Python: Funções

Abstração

- É uma técnica de programação que nos permite pensar num problema em diversos níveis
- A idéia é que quando estamos pensando num problema macroscopicamente, não estamos preocupado com minúcias
- Dividir para conquistar:
 - Um problema é dividido em diversos sub-problemas
 - As soluções dos sub-problemas são combinadas numa solução do problema maior

Programação Estruturada

- É uma disciplina de programação que incorpora o princípio de “Dividir para Conquistar”
 - (Programação Orientada a Objetos é outra...)
- Programas são divididos em sub-programas
 - Cada sub-programa é invocado por meio de um identificador e uma lista de entradas
 - Permite especificar como um problema pode ser resolvido *em geral*
 - O mesmo sub-programa pode ser invocado para resolver diversos problemas de mesma natureza mas com valores específicos diferentes
 - Os resultados computados por um sub-programa pode ser combinado com os de outros sub-programas

Definindo funções

- Em Python, sub-programas têm o nome de *funções*

- Formato geral:

```
def nome (arg, arg, ... arg):  
    comando  
    ...  
    comando
```

- Onde:

- *nome* é o nome da função
- *args* são especificações de argumentos da função
 - Uma função pode ter 0, 1 ou mais argumentos
- *comandos* contêm as instruções a ser executadas quando a função é invocada

Resultado de funções

- Uma função tipicamente computa um ou mais valores
- Para indicar o valor a ser devolvido como o resultado da função, usa-se o comando `return` que tem o formato `return expressão`
 - onde a *expressão* é opcional e designa o valor a ser retornado
- Ao encontrar o comando `return`, a função termina imediatamente e o controle do programa volta ao ponto onde a função foi chamada
- Se uma função chega a seu fim sem nenhum valor de retorno ter sido especificado, o valor de retorno é `None`

Exemplo

```
>>> def f():  
    return
```

```
>>> print f()  
None
```

```
>>> def f():  
    return "Oi"
```

```
>>> print f()  
Oi
```

```
>>> def f(nome):  
    return "Oi, "+nome+"!"
```

```
>>> print f("Joao")  
Oi, Joao!
```

Variáveis locais e globais

- Variáveis definidas em funções são *locais*, isto é, só podem ser usadas nas funções em que foram definidas
- Variáveis definidas fora de funções são conhecidas como variáveis globais
 - É possível no código de uma função ler o conteúdo de uma variável global
 - Para alterar uma variável global, ela precisa ser declarada no corpo da função usando o comando `global`

Exemplo

```
>>> def f():  
    print a
```

```
>>> a = 1
```

```
>>> f()
```

```
1
```

```
>>> def f():  
    a = 5
```

```
>>> f()
```

```
>>> print a
```

```
1
```

```
>>> def f():  
    global a  
    a = 5
```

```
>>> f()
```

```
>>> print a
```

```
5
```


Argumentos de funções

- Argumentos (ou parâmetros) são como variáveis que recebem seus valores iniciais do chamador
- Essas variáveis, assim como outras definidas dentro da função são ditas *locais*, isto é, só existem no lugar onde foram definidas
 - Ao retornar ao ponto de chamada, as variáveis locais são descartadas
- Se uma função define n argumentos, valores para todos eles devem ser passados pelo chamado
 - Exceção: argumentos com valores default

Exemplo

```
>>> def f(x):  
        return x*x
```

```
>>> print f(10)  
100
```

```
>>> print x
```

```
.....
```

```
NameError: name 'x' is not defined
```

```
>>> print f()
```

```
.....
```

```
TypeError: f() takes exactly 1 argument (0  
given)
```

Argumentos *default*

- É possível dar valores *default* a argumentos
 - Se o chamador não especificar valores para esses argumentos, os defaults são usados
- Formato:
`def nome (arg1=default1, ..., argN=defaultN)`
- Se apenas alguns argumentos têm default, esses devem ser os *últimos*
 - Se não fosse assim, haveria ambigüidade na passagem de argumentos

Exemplo

```
>>> def f(nome,saudacao="Oi",pontuacao="!!"):  
    return saudacao+", "+nome+pontuacao
```

```
>>> print f("Joao")
```

```
Oi,Joao!!
```

```
>>> print f("Joao","Parabens")
```

```
Parabens,Joao!!
```

```
>>> print f("Joao","Ah","...")
```

```
Ah,Joao...
```

Passando argumentos com nomes

- É possível passar os argumentos sem empregar a ordem de definição desde que se nomeie cada valor passado com o nome do argumento correspondente
- Ex.:

```
>>> def f(nome,saudacao="Oi",pontuacao="!!"):  
        return saudacao+" "+nome+pontuacao  
  
>>> print f(saudacao="Valeu",nome="Joao")  
Valeu,Joao!!
```

Alterando parâmetros

- É possível alterar parâmetros?
 - Sim e não
 - Como o parâmetro é uma variável local, ele pode ser alterado sem problemas
 - Entretanto, se um parâmetro recebe um valor que vem de uma variável global, esta não é alterada

■ Ex.:

```
>>> def f(x):  
        x = 5  
>>> a = 1  
>>> f(a)  
>>> print a  
1
```

Alterando parâmetros

- Note que quando passamos uma variável do tipo lista como parâmetro, estamos passando uma *referência* para um valor do tipo lista
 - Nesse caso, alterar o parâmetro pode influenciar no “valor” da variável global
 - Na verdade, o “valor” da variável do tipo lista é uma referência que *não muda*
 - Este caso é idêntico a termos duas variáveis se *referindo* ao mesmo valor

Exemplo

```
>>> def f(x):  
    x[:] = [5]
```

```
>>> a = [1]
```

```
>>> f(a)
```

```
>>> a
```

```
[5]
```

```
>>> b = a
```

```
>>> b[:] = [7]
```

```
>>> a
```

```
[7]
```


Documentando Funções

- Ao invés de usar comentários para descrever o que uma função, é mais vantajoso usar *docstrings*
 - Uma constante string escrita logo após o cabeçalho da função (comando def)
 - Permite o acesso à documentação a partir do interpretador, usando a notação *função* . `__doc__`

```
>>> def fat(n):  
...     "Retorna o fatorial de n."  
...     for i in range(n-1,1,-1): n*=i  
...     return n  
...  
>>> fat(4)  
24  
>>> print fat.__doc__  
Retorna o fatorial de n.
```

Lista de parâmetros variável

- Se o último argumento de uma definição de função começa com *, os todos os valores passados a partir daquele são postos numa tupla

- Ex.:

```
>>> def imprime(nome,*atributos):  
...     print nome,atributos  
...  
>>> imprime ('a',1,2,'b')  
a (1, 2, 'b')  
>>> def media(*valores):  
...     total=0.0  
...     for x in valores: total+=x  
...     return total/len(valores)  
...  
>>> media(1,2,3,4)  
2.5
```

Lista de parâmetros variável (2)

- Se o último argumento de uma definição de função começa com `**`, os todos os valores passados usando chaves a partir daquele são postos num dicionário

- Ex.:

```
>>> def f(a,b,**c):  
        print a, b, c
```

```
>>> f(1,2,3)
```

```
...
```

```
TypeError: f() takes exactly 2 arguments (3 given)
```

```
>>> f(1,2,x=3)
```

```
1 2 {'x': 3}
```

Lista de parâmetros variável (3)

- É possível passar os valores de uma tupla para preencher parâmetros posicionais de uma função bastando para isso precedê-la de *
- Um dicionário podem ser usado para preencher parâmetros por chave bastando para isso precedê-lo de **
- É preciso tomar cuidado para não abusar!
- Ex.:

```
>>> def f(a,b,*c,**d):  
        print a,b,c,d
```

```
>>> f(*[1,2,3,4,5])
```

```
1 2 (3, 4, 5) {}
```

```
>>> f(**{"a":1,"b":2,"c":3,"d":4})
```

```
1 2 () {'c': 3, 'd': 4}
```

```
>>> f(1,2,3,**{"d":1})
```

```
1 2 (3,) {'d': 1}
```

```
>>> f(1,2,3,**{"a":1})
```

```
...
```

```
TypeError: f() got multiple values for keyword argument 'a'
```

Passando funções

- Nomes de funções podem ser manipulados como variáveis e mesmo como argumentos de funções
 - Para saber se um nome se refere a uma função, use o predicado `callable()`

■ Ex.:

```
>>> def f(g):  
    return g(5)
```

```
>>> def h(x):  
    return x*x
```

```
>>> f(h)  
25
```

```
>>> m = h
```

```
>>> callable(m)
```

```
True
```

```
>>> f(m)  
25
```

Escopo

- Escopo é o nome que se dá ao conjunto de nomes acessíveis de um determinado ponto de um programa
 - Também é chamado de *espaço de nomes* ou *namespace*
- Um programa começa em um escopo (chamado escopo global) enquanto que cada função acrescenta um escopo próprio (local)
 - Módulos e classes também definem escopos
- Ao se fazer acesso a um nome, todos os escopos, do mais interno para o mais externo, são consultados.
 - Isto explica por que definir uma variável numa função pode fazer com que uma variável global deixe de ser acessível

Função *vars()*

- O dicionário obtido com a função `vars()` pode ser usado para ter acesso a todas as variáveis definidas num escopo. Ex.:

```
>>> vars()
{'__builtins__': <module '__builtin__' (built-in)>,
 '__name__': '__main__', '__doc__': None}
>>> def f():
        x = 1
        print vars()
```

```
>>> vars()
{'f': <function f at 0xb6e7f56c>, '__builtins__':
 <module '__builtin__' (built-in)>, '__name__':
 '__main__', '__doc__': None}
>>> f()
{'x': 1}
```

Funções definidas em funções

- Funções podem ser definidas dentro de funções
- Se uma função `g` é definida dentro de uma função `f`, ela tem acesso ao seu próprio escopo (em primeiro lugar) e também ao escopo de `f`

■ Ex.:

```
>>> def f(x):  
        def g(y): return x*y  
        return g(2)
```

```
>>> print f(4)
```

```
8
```


Funções definidas em funções (2)

- Observe que, se uma função *g* foi definida dentro de outra função *f*, então, se *g* é armazenada numa variável ou transmitida para outra função ela carrega com si os valores do escopo de *f* (mas não o escopo global). Ex:

```
>>> x = 2
>>> def f(y):
def g(z): return x*y*z
return g
```

```
>>> h = f(3)
>>> print h(1)
6
```

```
>>> x = 3
>>> print h(1)
9
```

Formas *lambda*

- São pequenas funções “sem nome”
 - Tipicamente criadas para serem passadas para outras funções
 - Historicamente, são relacionadas com o estilo “funcional” de programar (LISP)
- Em Python, funções *lambda* estão restritas a retornar unicamente uma expressão
 - Formato: `lambda arg,...arg: expressão`
 - Exemplo:

```
>>> f = lambda x: 2*x*x+1
>>> f(1)
3
>>> f(3)
19
```

Programação Funcional - filter

- Além de formas lambda, Python provê algumas ferramentas clássicas usadas em programação funcional
- `filter` (*função, seqüência*)
 - Retorna outra seqüência com os elementos *x* de *seqüência* tais que *função(x)* é verdadeiro
 - Se *seqüência* é uma tupla ou string, o resultado é também uma tupla ou string, caso contrário, o resultado é uma lista
- Exemplos:

```
>>> filter (lambda x: x%2==0, [1,2,3,4,5,6,7,999])
[2, 4, 6]
>>> filter (lambda x: str(x)==x, [1,2,3,'abc',4])
['abc']
>>> filter (lambda c: c not in 'aeiou', "consoantes")
'cnsnts'
```

Programação Funcional - map

- `map` (*função*, *seqüência*)

- retorna outra seqüência cujos elementos y_i são computados por $função(x_i)$ onde x_i são os elementos de *seqüência*

- Exemplos:

```
>>> map(lambda x: x+1, [1,2,3,4])  
[2, 3, 4, 5]
```

```
>>> def f(x):  
    if x%2==0: return x*x  
    else: return x
```

```
>>> map (f, [1,2,3,4])  
[1, 4, 3, 16]
```

Programação Funcional - reduce

- `reduce (função, seqüência [,inicializador])`
 - Retorna o valor acumulado obtido aplicando a função binária *função* a pares consecutivos de elementos em *seqüência*
 - Ex.: `reduce (f, [a, b, c, d])` é equivalente a `f(f(f(a, b), c), d)`
 - Se o argumento opcional *inicializador* é especificado, ele determina um valor a ser considerado antes dos elementos da *seqüência*
 - Ex.: `reduce (f, [a, b, c, d], 0)` é equivalente a `f(f(f(f(0, a), b), c), d)`

Exemplo

```
>>> reduce (lambda a,b: a+b, [1,2,3,4,5])  
15
```

```
>>> reduce (lambda a,b: a+b, [])  
Traceback (most recent call last):  
  File "<pyshell#9>", line 1, in ?  
    reduce (lambda a,b: a+b, [])  
TypeError: reduce() of empty sequence with no  
initial value
```

```
>>> reduce (lambda a,b: a+b, [], 0)  
0
```