Nesta aula...

Conteúdo

1 Definição de funções

Definição de novas funções

- Na aula passada: vimos como usar os *operadores* e *funções pré-definidas*
- Nesta aula: vamos ver como definir *novas funções*
- Podemos depois usar as novas funções tal qual as pré-definidas
- Programar: *decompor* um problema em funções cada vez mais simples até chegar às operações elementares

Definição de novas funções

```
def nome(lista de parâmetros):
    primeira instrução
    segunda instrução
    ⋮
    instrução final
```

- início e fim de bloco marcados pela *indentação*
- lista de parâmetros pode ser vazia

Exemplo

```
def refrao():
    print "Se um elefante incomoda muita gente"
    print "Dois elefantes incomodam muito mais."

def repete_refrao():
    refrao()
    refrao()
```

Fluxo da execução

1. começa na primeira instrução do programa
2. instruções são executadas uma de cada vez, de cima para baixo
3. *definição* de uma função não altera fluxo de execução do programa
4. instruções da função só são executadas se a função for chamada
5. quando uma função é *chamada*:
   (a) execução transferida para a primeira instrução da função
   (b) executa todas as instruções da função
   (c) no final regressa ao ponto de onde partiu
6. funções podem chamadas dentro de outras

Parâmetros e argumentos

- A maior parte funções precisam de *argumentos*:

```
>>> import math
>>> math.sin()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: sin() takes exactly 1 argument
(0 given)
```

- Dentro da função, o valor dos argumentos é associado a variáveis chamadas *parâmetros*

Exemplo do livro

```
def print_twice(bruce):
    print bruce
    print bruce
```

Pode ser usada com argumentos de diversos tipos:

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(5)
5
5
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Funções que calculam valores

- Uma função pode *retornar* um valor:

```
def quadrado(x):
    return x*x
```

- Pode ser usada no meio de uma expressão:

```
>>> from math import *    # para usar sqrt()
>>> quadrado(2)
4
>>> quadrado(sqrt(2))
2.0000000000000004
>>> quadrado(quadrado(2)-1)
9
```

Âmbito de variáveis

- parâmetros e variáveis definidas numa função são *locais* (invisíveis fora da função)
- atribuições a variáveis locais não modificam variáveis em âmbitos exteriores

```
>>> x = 'ola mundo'  # x é global
>>> quadrado(2)      # parâmetro x é local
4
>>> x
'ola mundo'
```

Âmbito de variáveis

Variáveis de âmbitos exteriores são visíveis dentro de uma função:

```
# taxa do imposto de valor acrescentado
taxa_IVA = 1.19

# calcula o preço com IVA
def precoFinal(total):
   return total * taxa_IVA


>>> precoFinal(100)
119.0
```

Documentação

comentários
- começam por # até ao final da linha
- colocados em qualquer parte dum programa
- destinados ao programador e outros leitores humanos

docstrings
- texto que descreve um componente
- associado a funções (mas também classes e métodos)
- usada pelo interpretador Python: `help`

Documentação

```
# definição da função precoFinal
# Pedro Vasconcelos, 2007

# taxa de imposto de valor acrescentado
taxa_IVA = 1.19

def precoFinal(total):
    "Acrescenta a taxa de IVA ao total."
    return total * taxa_IVA

# fim da função
```

Documentação

```
>>> help(precoFinal)
Help on function precoFinal in module __main__:

precoFinal(total)
    Acrescenta a taxa de IVA ao total.
```

Como decompor um problema?

Cada função deve:

1. efectuar uma tarefa
   - necessária em mais do que uma parte do programa
   - ou re-utilizável para outros problemas
2. ter um propósito claro (explicar na linha de *docstring*)
3. ter uma *interface* clara com o exterior (significado dos parâmetros e resultado)
4. ter uma definição concisa (uma página de texto no máximo)

Funções "puras"

Funções "puras": limitam-se a devolver um valor dependente dos argumentos
Análogas às funções matemáticas. Exemplos

$$\text{mdc} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \qquad \text{mdc}(24, 15) \to 3$$

$$\text{primo} : \mathbb{N} \to \mathbb{B} \qquad \text{primo}(29) \to \texttt{True}$$

Quase todas as funções que vamos escrever nesta disciplina são puras:

1. Retornam um resultado (do contra-domínio)
2. Não usam nem alteram variáveis exteriores à função
3. Não contêm instruções de leitura (`input...`) nem de impressão (`print...`)

2 Valores boleanos e condicionais

Comparações

- Operadores de comparação:

`==`	igual
`!=`	diferente
`>`	maior
`<`	menor
`>=`	maior ou igual
`<=`	menor ou igual

- Resultado: `True` ou `False` (*verdade* ou *falsidade*)

Operadores lógicos

conjunção	P `and` Q
disjunção	P `or` Q
negação	`not` P

```
>>> import math
>>> math.pi>3
True
>>> math.pi>3 and math.pi<4
True
>>> math.pi>3.5
False
>>> not (math.pi>3.5)
True
```

Valores lógicos

Verdade `True` ou um valor diferente de zero

Falso `False` ou zero

```
>>> True and 0
0
>>> 1 and True
True
>>> not 100
False
```

Execução condicional

- Forma mais simples: `if`

```
if x > 0:
  print x, "positivo"
```

- Com alternativa: `if...else`

```
if x%2 == 0:
  print x, "par"
else:
  print x, "impar"
```

- expressão após `if` é a *condição*
- bloco indentado após `if` é executado se a condição for verdadeira
- bloco indentado após `else` é executado se a condição for falsa

Condições embricadas

```
if x == y:
  print x, "e", y, "são iguais"
else:
  if x < y:
    print x, "menor que", y
  else:
    print x, "maior que", y
```

- a indentação indica a estrutura das condições
- mais do que dois níveis: difícil de ler

Condições embricadas: alternativa

```
if x == y:
  print x, "e", y, "sao iguais"
elif x < y:
  print x, "menor que", y
else:
  print x, "maior que", y
```

- `elif` substitui o `else...if`
- um nível de indentação: leitura mais fácil
- caso geral:
  1. um `if`
  2. um ou mais `elif`
  3. um `else`