Computadores e Programação

2009–2010 1º semestre — aula 1

Helmut Wolters

helmut@coimbra.lip.pt

2009-09-25

O que é o Python?

- O Python é uma VHLL (Very High Level Language).
- O próprio criador da linguagem dá dela a seguinte definição: *Python is an interpreted, interactive, object-oriented programming language. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing.*
- Guido van Rossum (Univ. Amesterdão), que criou esta linguagem de programação em 1991, teve como fonte de inspiração para o nome a série britânica de humor da década de 70 "Monty Python's Flying Circus", de que é fã incondicional...

O que é o Python?

- O Python ainda está em desenvolvimento por uma grande equipa de colaboradores, liderada por GvR. A licença de utilização é do tipo GPL.
- Está disponível gratuitamente (em código fonte, C) para a maioria dos sistemas operativos em `http://www.python.org/`.

Programação em Python

- Estrutura de um programa em Python:
  - Os programas são compostos por **módulos**
  - Os módulos contêm **instruções** e **expressões**
  - As instruções e as expressões criam e processam **objectos**.

O que é um objecto?

- É difícil dar, nesta altura, uma definição rigorosa deste conceito. De uma forma simples, os objectos correspondem a uma certa região da memória do computador à qual está associada um endereço único e na qual estão armazenados:
  - dados
  - informações sobre os dados
  - funções que actuam sobre estes dados.

O que é um objecto?

- A instrução básica em Python consiste em criar um objecto e dar-lhe um nome; a esta instrução dá-se o nome de atribuição (de nome!).
- Exemplo:

```
>>> x = 123
```

cria o objecto 123, algures na memória do computador e dá-lhe o nome `x`. Diz-se também que `x` é uma referência para o objecto 123 ou que "aponta" para o objecto 123, tudo com o mesmo significado.

O que é um objecto?

- Depois de criados, os objectos passam a ser referidos através dos seus nomes. Por exemplo, a instrução:

  ```
  >>> print x
  ```

  imprime no ecrã o *valor* do objecto cujo nome é `x`. Nem todos os objectos têm um valor, mas todos os objectos têm um *tipo* e um *endereço* únicos.
- Para obter o tipo de um objecto, utiliza-se a instrução `type(nome)`, neste caso `type(x)`.

  ```
  >>> x = 123
  >>> type(x)
  <type 'int'>
  >>>
  ```

O que é um objecto?

- Para obtermos o endereço de um objecto utiliza-se a instrução `id(nome)`. Assim:

```
>>> x = 1.23456
>>> print x
1.23456
>>> type(x)
<type 'float'>
>>> id(x)
135625436
>>>
```

O que é um objecto?

- É possível dar mais do que um nome a um mesmo objecto. A esta operação dá-se o nome de *aliasing* e pode revelar-se útil em certos casos.

```
>>> x = 45
>>> y = 45
>>> id(x)
135363888
>>> id(y)
135363888
>>>
```

O que é um objecto?

- Pode-se ainda fazer este "baptismo duplo" na mesma linha:

```
>>> x = y = 45
>>> id(x)
135363888
>>> id(y)
135363888
>>>
```

O que é um objecto?

- No entanto, um mesmo nome não pode ser usado por mais do que um objecto ao mesmo tempo!

```
>>> x = 20
>>> x = 43
>>> print x
43
>>>
```

- Prevalece sempre a última instrução de atribuição... O objecto 20 deixou de ter nome e o Python encarregar-se-á de apagar os objectos sem nome atribuído da memória do computador (*recolha de lixo*).

Tipos e categorias de objectos

Cada objecto tem um *tipo* e uma *categoria*. Existem vários tipos de objectos:

- os *números*, dos quais há os seguintes tipos: inteiros, inteiros longos, números em vírgula flutuante, números complexos.
- as *colecções*: *sequências* e *mapas*. Nas sequências temos os tipos listas, tuplas e cadeias de caracteres. Nos mapas, existe um tipo único, o dicionário.
- funções, classes e métodos
- ficheiros
- etc...

Tipos e categorias de objectos

- Cada objecto pertence a uma das categorias: *mutável* ou *imutável*.
- **Um objecto imutável não pode ser alterado, uma vez criado só lhe podemos dar ou mudar o nome, ou destruí-lo. Um objecto mutável pode sofrer alterações.**
- **Os números, as cadeias de caracteres e as tuplas são objectos imutáveis.**

Tipos e categorias de objectos

- Um objecto pode destruir-se com a instrução `del`.

```
>>> x = 123
>>> print x
123
>>> del(x)
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Algumas notas sobre os tipos numéricos

- inteiros: representação em complementos de 2 usando 32 *bits*.
- inteiros longos: quando o número inteiro está fora do intervalo $[-2147483648, 2147483647]$, utiliza o número de *bits* necessário, dentro da disponibilidade de memória do computador.
- vírgula flutuante: representação IEEE754 de dupla precisão (64 *bits*.)
- complexos: a parte real e imaginária são codificadas em 64 *bits*. Os símbolos j e J representam a unidade imaginária. Exemplo: z = 2+3j.

Algumas notas sobre os tipos numéricos

- vírgula flutuante: representação IEEE754 de dupla precisão (64 *bits*.)

```
>>> x= 1e10
>>> print x
10000000000.0
>>> x=1e131
>>> print x
1e+131
>>> x=123456789012345678901.1234567890
>>> print x
1.23456789012e+19
>>> x=1e9999
>>> print x
inf
>>> type(x)
<type 'float'>
```

limitações em **precisão** e em **valor absoluto** máximo/mínimo

Expressões aritméticas

As expressões aritméticas habituais podem ser usadas com objectos do tipo numérico. A conversão de inteiro $\longrightarrow$ longo inteiro $\longrightarrow$ vírgula flutuante funciona do modo esperado, excepto no caso da divisão de inteiros, que é entendida como a **divisão inteira**.

```
>>> n = 1
>>> z = 2*n
>>> print z
2
>>> x = 1.23456
>>> print 2*3.456+5*x
13.0848
>>> print z/3
0
```

Expressões aritméticas

Estão disponíveis os seguintes operadores aritméticos binários:

Operador	Significado	Precedência
+	soma	0
-	subtracção	0
*	multiplicação	1
/	divisão	1
//	divisão inteira	1
%	resto da divisão inteira	1
**	potência	2

Expressões aritméticas

Na avaliação de uma expressão aritmética o operador potência tem o mais elevado grau de precedência, seguindo-se a multiplicação, divisão e resto da divisão (todas ao mesmo nível) e por último as somas e subtracções.

```
>>> print 2**3+2*2
12
```

Quando se pretende impor uma ordem de avaliação diferente da imposta pelos graus de precedência dos operadores devem utilizar-se parêntesis:

```
>>> print 2**(3+2*2)
128
```

Estruturas de controlo

- O fluxo de um progama (sequência e encadeamento das instruções) é ditado pelas *estruturas de controle*, que em Python são as seguintes:
  - `if ... elif ...else`
  - `while...else`
  - `for...else`
  - `raise`
  - `break`
  - `continue`

Estruturas de controlo

- Comparado com outras linguagens de programação, o número de estruturas de controle é reduzido, o que facilita a aprendizagem. As estruturas de controle foram desenhadas para serem poderosas e de uso genérico. Por exemplo, a instrução

  `for ... else`

  permite percorrer qualquer objecto iterável e não se limita a *arrays* (listas homogéneas de objectos ocupando uma porção contígua da memória), como em `C` ou `Pascal` e na maioria das outras linguagens de programação.

Selecção: `if...elif...else`

```
if <test>:
    <statements>
[elif <test>:
    <statements>]
[else:
    <statements>]
```

- Nota importante:
  O teste de igualdade entre `x` e `y` é `x == y` e não `x = y`!

Selecção: if...elif...else

Exemplo:

```
if choice == 'eggs':
    print 'Eggs for lunch.'
elif choice == 'ham'
    print 'Ham for lunch'
elif choice == 'spam'
    print 'Hum, spam for lunch'
else:
    print "Sorry, unavailable choice."
```

Verdadeiro e falso

- A interpretação do valor lógico "verdadeiro" (`True`) ou "falso" (`False`) de um objecto em Python segue as seguintes regras:
  - Um objecto é considerado verdadeiro se for diferente de zero, caso seja um número, ou se o objecto não estiver "vazio".
  - Um objecto é considerado falso se não for verdadeiro, ou seja nos seguintes casos: número zero, objecto vazio ou o objecto `None`.
- O valor lógico de um objecto é calculado pela função `bool`; o valor devolvido é `True` ou `False`, os dois únicos objectos do tipo `bool`(eano).

Verdadeiro e falso

- Regras para expressões:
  - As comparações (`==` `!=` `>` `<` `>=` `<=`) devolvem o valor `True` ou `False` consoante sejam falsas ou verdadeiras.
  - `not a` é `False` se a for verdadeiro e `True` se a for falso.
  - `x or y` devolve o primeiro dos objectos que for verdadeiro; se nenhum for verdadeiro, devolve o último objecto.
  - `x and y` devolve o primeiro dos objectos que for falso; se nenhum for falso, devolve o último objecto.
- Os valores obtidos de acordo com estas regras são os esperados da lógica de Boole.

Ciclos de repetição: `while...else`

```
while <test>:
    <statements>
    [if <test>: break]
    [if <test>: continue]
    <statements>
[else:
    <statements>]
```

- Nota: As cláusulas `break`, `continue` e `else` são opcionais. Se o teste da cláusula `break` for verdadeiro, o programa salta para o fim do bloco `while`. Se o teste da cláusula `continue` for verdadeiro, o programa abandona o ciclo actual e continua no próximo ciclo. Se existir a cláusula `else` e se a instrução `break` não for accionada, o programa executa as instruções que se seguem a `else` no final dos ciclos.

Ciclos de repetição: `while...else`

- Exemplo 1:

```
a = 0; b = 10
while a < b:
    print a,
    a += 1  # a = a+1

Resultado:  >>> 0 1 2 3 4 5 6 7 8 9
```

- Exemplo 2:

```
x = 10
while x:
    x -= 1    # x = x-1
    if x % 2 == 0:
        print x,

Resultado: >>> 8 6 4 2 0
```

Ciclos de repetição: `while...else`

- Exemplo 3:

```
name = 'Spam'
while name:
      print name,
      name = name[1:]

Resultado: >>> Spam pam am m
```

Ciclos de repetição: `while...else`

- Exemplo 4:

```
# Guess a number game
mynumber = '123456'
while 1:
        n = input('Guess the number: ')
        if n == mynumber:
            print 'You guessed the number!'
            break;
        else:
            print 'Sorry, wrong guess.'
print 'Game is over'
```

Ciclos de repetição: `while...else`

Resultado:

```
Guess the number: 43465
Sorry, wrong guess.
Guess the number: 7161527
Sorry, wrong guess.
Guess the number: 999999
Sorry, wrong guess.
Guess the number: 123456
You guessed the number!
Game is over.
```

Ciclos de iteração: instrução `for ...else`

- Em Python a instrução `for` é utilizada (exclusivamente) para iterar sobre objectos. Na versão actual do Python todas as sequências, os dicionários e os ficheiros têm um iterador definido por defeito, que devolve cada um dos seus elementos em sequência, mas é possível definir iteradores para outros objectos.
- Sintaxe:

```
for x in object:
    <instruções>
    [if condição: break>]
[else:
    <instruções>]
```

Ciclos de iteração: instrução `for ...else`

- Se existir a cláusula opcional `else`, a instruções que se lhe seguem serão executadas apenas no caso de o ciclo atingir o último elemento do objecto sem a condição `break` ser accionada.
- Exemplos:

```
basket = ['orange','banana','apple']
for fruit in basket:
    print fruit

phrase = 'Coimbra is a nice town.'
for c in phrase:
    print c,ord(c) # ord(c) = ASCII code
```

Ciclos `for`: continuação

- Exemplo de utilização da cláusula `else`. Suponhamos que queremos testar se existe algum número negativo na lista `L`. Podemos fazê-lo com o seguinte código:

```
for x in L:
    if x < 0:
        print ‘‘There are negative numbers’’
        break
else:
    print ‘‘All numbers are non-negative’’
```

- O Python possui a função intrínseca `range` que é muito útil para criar listas de inteiros que são frequentemente usadas para iterar ciclos for:

```
range(n) -> [0,1,2, ...n-1]
range(i,j) -> [i,i+1,...j-1]
range(i,j,k) -> [i,i+k, i+2k, ...]
```

Ciclos `for`: continuação

- Exemplos:
  `range(5) = [0,1,2,3,4]`
  `range(2,5) = [2,3,4]`
  `range(1,10,2) = [1,3,5,7,9]`
  `range(0,-10,-3) = [0,-3,-6,-9]`
- Para ciclos que exijam listas muito extensas é preferível utilizar a função `xrange`, semelhante à anterior, mas que permite economizar memória, sendo a lista criada à medida que for necessário.

Ciclos `for`: continuação

- Advertência importante sobre a utilização de ciclos *for*: a lista iteradora não deve ser alterada no interior do ciclo sob pena de se obterem resultados errados!
- Exemplo (errado!):

```
for x in lista:
    if x < 0: lista.remove(x)
```

- Para o efeito devemos iterar não sobre a lista mas sobre uma *cópia* (ou *clone*) da lista:

```
for x in lista[:]:
    if x < 0: lista.remove(x)
```

Ciclos `for`: continuação

- Para iterar simultaneamente sobre várias sequências existe uma outra função muito útil: `zip`.

```
colors = ("red","green","blue")
clubs = ("Benfica","Sporting","Porto")

for club,color in zip(clubs,colors):
     print club,color
```