

Programação e Base de Dados /Computação para Detecção Remota

Introdução à linguagem Python

Sérgio Crisóstomo,
(adaptado dos slides de Rita Ribeiro,
Pedro Vasconcelos e João Pedro Pedroso)
2022/2023

Departamento de Ciência de Computadores



1. Porquê programar?
2. A linguagem Python
3. Tipos básicos
4. Variáveis e atribuições
5. Programas completos
6. Definição de funções
7. Funções que calculam resultados

Porquê programar?

O que é a programação de computadores?

- Implementação de **métodos computacionais** para resolução de problemas
- Análise e comparação de métodos diferentes
- Conjunção de várias competências:
 - matemática** linguagens formais para especificar processos;
 - engenharia** juntar componentes para formar um sistema;
avaliar prós/contras de alternativas
 - ciências naturais** observar comportamento de sistemas
complexos; formular hipóteses; testar previsões

Porquê aprender a programar?

- Trabalhos científicos necessitam de processamento complexo de dados
- Facilita a automatização de tarefas repetitivas
- Muitas aplicações científicas são programáveis (ex: Excel, GNUplot, Matlab, Maple, Mathematica)
- Estrutura o pensamento para resolver problemas
- Desenvolve o pensamento analítico
- É um desafio intelectual
- É divertido!

Porquê aprender a programar? (cont.)

Programar desenvolve competências de **resolução de problemas**:

- capacidade para descrever problemas de forma rigorosa;
- pensar de forma criativa em possíveis soluções;
- expressar as soluções de forma clara e precisa.

Linguagens de Programação

- Linguagens formais para exprimir computação
 - sintaxe:** regras de formação (gramática)
 - semântica:** significado ou operação associados
- Outras linguagens: expressões aritméticas, símbolos químicos

	sintaxe	semântica
$3 \times (1 + 2)$	ok	9
$3 \times 1 + 2$	ok	5
$\times)1 + 2 + (3$	erro	—
H_2O	ok	água
$_2zZ$	erro	—

A linguagem Python

A linguagem Python

- Linguagem de alto nível
- Sintaxe simples: fácil de aprender
- Pode ser usada em Windows, Linux, FreeBSD, Mac OS, etc... .
- Implementação distribuida como *código livre*
- Suporta programação procedimental e orientada a objectos
- Muitas bibliotecas disponíveis
- Usada no “mundo real”: Google, Microsoft, Yahoo!, NASA, Lawrence Livermore Labs, Industrial Light & Magic... .
- Sítio oficial: <http://www.python.org>

A linguagem Python

Python é implementado com um **interpretador híbrido**:

- programa Python é traduzido para um código intermédio (*byte-code*);
- o *byte-code* é executado por um interpretador especial.

Vantagens:

- fácil de usar interativamente
- fácil testar e modificar componentes
- mais eficiente do que um interpretador clássico

Desvantagem: não é tão eficiente como uma linguagem compilada tradicional (ex: C, C++, Fortran)

Utilização interativa

Executando `python3` num terminal podemos escrever comandos Python e ver os resultados imediatamente.

```
Python 3.2.3 (default, Jul  5 2013, 08:29:02)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>> 1+1
2
>>> print("Ola, mundo!")
Ola, mundo!
>>>
```

Ctrl-D ("end-of-file") ou `quit()` para terminar.

Utilização com um *script*

Em alternativa podemos escrever um **programa completo** num ficheiro de texto (*script*) e executar de uma só vez.

```
_____ programa.py _____  
print("Ola, mundo!")  
print("1 + 1 = ", 1+1)
```

Executamos no terminal “python3 programa.py” e obtemos:

```
Ola, mundo!  
1 + 1 = 2
```

Convenção: ficheiros de programas Python têm extensão `.py`

Utilização com um *script* (cont.)

- A forma interativa é usada para testar pequenas partes de código.
- Devemos escrever programas com mais do que algumas linhas num *script*.
- *Ambientes de desenvolvimento* como o IDLE e o Pyzo combinam:
 - uma janela para testes interativos;
 - uma ou mais janelas para *scripts*(segue-se uma demonstração. . .)

Usar Python como uma calculadora

- Operadores aritméticos básicos:

adição e subtração + -

multiplicação e divisão * /

exponenciação **

parênteses ()

- Números inteiros e fracionários: 42 -7 3.1416
- Expressões incorretas: `SyntaxError`

Usar Python como uma calculadora (cont.)

Prioridade entre os operadores (ordem de cálculo):

1. parêntesis ()
2. exponenciação **
3. multiplicação e divisão * /
4. soma e subtração + -

Operadores da mesma prioridade **agrupam à esquerda**.

Exemplos:

```
>>> 1+2-3+4
```

```
4
```

```
>>> 1+2-(3+4)
```

```
-4
```

```
>>> 2**3*4+1
```

```
33
```

```
>>> 2**3*(4+1)
```

```
40
```

Funções matemáticas

Muitas funções e constantes matemáticas estão disponíveis no módulo `math`.

Para usar devemos começar por **importar** o módulo.

```
>>> import math
```

Os nomes das funções começam com prefixo “`math`”:

```
>>> math.sqrt(2)
1.4142135623730951
>>> math.pi
3.141592653589793
```


Funções matemáticas (cont.)

Algumas funções e constantes do módulo `math`:

raiz quadrada	<code>sqrt</code>
funções trigonométricas	<code>sin, cos, tan</code>
funções trigonométricas inversas	<code>asin, acos, atan, atan2</code>
exponencial e logaritmos	<code>exp, log, log10</code>
e , π	<code>e, pi</code>

Para obter mais informação:

```
>>> help(math)
```

```
>>> help(math.log)
```

informação geral
específica sobre uma função

Tipos

Os valores em Python são classificados em diferentes **tipos**.

Algumas operações só são possíveis com determinados tipos:

```
>>> "Ola mundo!" + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str
```

Tipos básicos

	tipo	exemplos
inteiros	int	1 -33 29
vírgula-flutuante	float	1.0 -0.025 3.14156
cadeias de texto	str	"Ola mundo!" 'ABC' '1.23.99'

Tipo de um resultado

No interpretador de Python podemos usar `type(...)` para obter o tipo dum resultado:

```
>>> (1+2+3)*5-1  
29
```

```
>>> type((1+2+3)*5-1)  
<class 'int'>
```

```
>>> type(1.234)  
<class 'float'>
```

```
>>> type('ABC')  
<class 'str'>
```

Tipos básicos

Tipos numéricos

Em Python distinguimos números **inteiros** e **fracionários** (*vírgula-flutuante*) associando-lhes **tipos distintos**.

	tipo	exemplos
inteiros	int	42 -7
vírgula-flutuante	float	42.0 -7.0 -0.0254

Tipos numéricos (cont.)

As operações aritméticas funcionam com ambos os tipos:

```
>>> 1+2          int + int => int
3
```

```
>>> 1.0+2.0      float + float => float
3.0
```

Também podemos usar tipos diferentes numa operação; o resultado será um `float`:

```
>>> 1 + 2.5      int + float => float
3.5
```

Tipos numéricos (cont.)

Divisão entre inteiros dá um número fracionário¹:

```
>>> 17/5  
3.4
```

Podemos obter o *quociente* e o *resto* da divisão inteira com os operadores `//` e `%`:

```
>>> 17//5          quociente da divisão inteira  
3  
>>> 17%5          resto da divisão inteira  
2
```

¹ Isto é diferente nas versões de Python anteriores a 3.0

Erros de arredondamento

Números inteiros podem ser representados de forma exata no computador.²

Números em vírgula-flutuante são **aproximações finitas** dos números reais:

```
>>> 8/3  
2.6666666666666665
```

As operações sucessivas sobre estes números podem fazer acumular **erros de arredondamento**.

O controlo destes erros na computação é estudado em Análise Numérica.

²Apenas limitados pela memória disponível.

Erros de arredondamento

Usando álgebra exata:

$$\left(\frac{100}{3} - 33\right) \times 3 = 100 - 33 \times 3 = 1$$

Contudo, usando operações vírgula-flutuante obtemos resultados diferentes:

```
>>> (100.0/3.0 - 33.0) * 3.0
1.0000000000000007
>>> 100.0 - 33.0*3.0
1.0
```

O erro de arredondamento foi

$$1.0000000000000007 - 1 \approx 7 \times 10^{-15}$$

Conversão automática entre tipos numéricos

`int + int ⇒ int`

`float + float ⇒ float`

`int + float ⇒ float`

`float + int ⇒ float`

Também com os operadores aritméticos `-`, `*` e `**`.

A divisão (em Python 3) é um caso especial:

`int/int ⇒ float`

`int//int ⇒ int`

`int%int ⇒ int`

Conversão explícita entre tipos

```
>>> int(2.71)
```

```
2
```

```
>>> str(-3.134)
```

```
'-3.134'
```

```
>>> round(2.71)
```

```
3
```

```
>>> float("3.14")
```

```
3.14
```

```
>>> float(-33)
```

```
-33.0
```

```
>>> float("trinta e três")
```

```
ValueError
```

Nota:

`int(...)` faz a **truncatura**;

`round(...)` faz o **arrendondamento**.

Cadeias de caracteres

As cadeias de caracteres são valores de tipo `str` (*string*).

Escrevemos o texto entre **aspas simples ou duplas**:

```
>>> "Olá mundo!"
```

```
'Olá mundo!'
```

```
>>> 'abracadabra'
```

```
'abracadabra'
```

```
>>> type('abracadabra')
```

```
<class 'str'>
```

Cadeias de caracteres (cont.)

Podemos usar **três aspas** para introduzir cadeias de caracteres com várias linhas.

```
>>> '''Bom dia!  
--- Ola, mundo!'''  
'Bom dia!\n--- Ola, mundo!'
```

Operações sobre cadeias de caracteres

Concatenação $\text{str} + \text{str} \Rightarrow \text{str}$

Repetição $\text{int} * \text{str} \Rightarrow \text{str}$

```
>>> 'Olá'+' '+'Mundo'  
'Olá Mundo'
```

```
>>> 3*'Olá'+' Mundo!'  
'OláOláOlá Mundo!'
```

```
>>> 3*'Olá ' + 'Mundo!'  
'Olá Olá Olá Mundo!'
```

Variáveis e atribuições

Variáveis

- Nomes simbólicos para representar **quantidades** ou **propriedades** dum problema
- Começam com uma letra, seguido de letras, números ou sublinhado
- Podem ter letras com acentos³
- Não podem ter espaços ou tabulações
- Não podem ser *palavras reservadas* de Python:

and	def	exec	if	not	return
assert	del	finally	import	or	try
break	elif	for	in	pass	while
class	else	from	is	print	yield
continue	except	global	lambda	raise	

Exemplos de nomes válidos para variáveis:

```
nome  idade  Preço_Max  área2
```

Exemplos de nomes que **não podemos** usar:

```
76trombones  more$  class
```

³Só nas versões de Python a partir de 3.0.

Atribuições

Associa o valor de uma **expressão** a uma **variável**:

nome = expressão

```
>>> import math  
>>> raio = 1
```

<code>math.pi</code>	→	3.14159...
	⋮	
<code>raio</code>	→	1

Atribuições (cont.)

Depois de definir uma variável, podemos usá-la em cálculos seguintes:

```
>>> perimetro = 2*math.pi*raio  
>>> perimetro  
6.2831853071795862
```

math.pi	→	3.14159...
	⋮	
raio	→	1
perimetro	→	6.2831...

Atribuições (cont.)

Note que a atribuição é um **comando**, não é uma **equação**.

Exemplo: `perimetro` não muda se mudarmos o `raio`.

```
>>> raio = 2
>>> perimetro
6.2831853071795862
```

<code>math.pi</code>	→	3.14159...
	⋮	
<code>raio</code>	→	2
<code>perimetro</code>	→	6.2832...

Atribuições (cont.)

Podemos sempre re-calcular o perímetro voltando a executar a atribuição:

```
>>> perimetro = 2*math.pi*raio  
>>> perimetro  
12.566370614359172
```

math.pi	→	3.14159...
	→	:
raio	→	2
perimetro	→	12.5663...

Ordem de atribuições

A **ordem das atribuições** é importante!

Exemplo: vamos anotar os valores de p e n após cada instrução.

$p = 1$

$n = 2$

$p = p * n$

$n = n + 1$

$p = 1$

$n = 2$

$n = n + 1$

$p = p * n$

Ordem de atribuições

A **ordem das atribuições** é importante!

Exemplo: vamos anotar os valores de p e n após cada instrução.

$p = 1$ $p \rightarrow 1$

$n = 2$

$p = p * n$

$n = n + 1$

$p = 1$ $p \rightarrow 1$

$n = 2$

$n = n + 1$

$p = p * n$

Ordem de atribuições

A **ordem das atribuições** é importante!

Exemplo: vamos anotar os valores de p e n após cada instrução.

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$p = p * n$	
$n = n + 1$	

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$n = n + 1$	
$p = p * n$	

Ordem de atribuições

A ordem das atribuições é importante!

Exemplo: vamos anotar os valores de p e n após cada instrução.

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$p = p * n$	$p \rightarrow 2, n \rightarrow 2$
$n = n + 1$	

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$n = n + 1$	$p \rightarrow 1, n \rightarrow 3$
$p = p * n$	

Ordem de atribuições

A **ordem das atribuições** é importante!

Exemplo: vamos anotar os valores de p e n após cada instrução.

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$p = p * n$	$p \rightarrow 2, n \rightarrow 2$
$n = n + 1$	$p \rightarrow 2, n \rightarrow 3$

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$n = n + 1$	$p \rightarrow 1, n \rightarrow 3$
$p = p * n$	$p \rightarrow 3, n \rightarrow 3$

Ordem de atribuições

A ordem das atribuições é importante!

Exemplo: vamos anotar os valores de p e n após cada instrução.

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$p = p * n$	$p \rightarrow 2, n \rightarrow 2$
$n = n + 1$	$p \rightarrow 2, n \rightarrow 3$

No final: $p \rightarrow 2, n \rightarrow 3$

$p = 1$	$p \rightarrow 1$
$n = 2$	$p \rightarrow 1, n \rightarrow 2$
$n = n + 1$	$p \rightarrow 1, n \rightarrow 3$
$p = p * n$	$p \rightarrow 3, n \rightarrow 3$

No final: $p \rightarrow 3, n \rightarrow 3$

Programas completos

perimetro.py

Calcular o perimetro de uma circunferência

Pedro Vasconcelos, 2013

import math

raio = 2.5

perimetro = 2*math.pi*raio

Executa correctamente, mas não mostra resultados!

`input(text)` escreve texto (opcional) e lê uma cadeia de caracteres

`print(expr1, expr2, ...)` escreve valores no terminal

perimetro.py

```
# Calcular o perimetro de uma circunferência  
# Pedro Vasconcelos, 2013  
  
import math  
  
raio = float(input('Qual é o valor do raio? '))  
perimetro = 2*math.pi*raio  
print('O perimetro da circunferência é', perimetro)
```

```
# Calcular o perimetro de uma circunferência  
# Pedro Vasconcelos, 2013
```

- Começam com o símbolo # e estendem até ao fim da linha
- Permitem incluir documentação para outros programadores
- Também úteis para o próprio autor (ex: para lembrar como funciona o programa)
- Evitar comentários redundantes, e.g.:

```
t = t + 10  # adicionar 10 a t
```

NOT OK

```
t = t + 10  # 10s extra de tempo
```

OK

Definição de funções

Definição de novas funções

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

- O início e fim da função são marcados pela **indentação**
- A 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()
```

Exemplo (cont.)

Vamos experimentar estas funções no interpretador:

```
>>> refrao()
```

```
Se um elefante incomoda muita gente  
Dois elefantes incomodam muito mais.
```

```
>>> repete_refrao()
```

```
Se um elefante incomoda muita gente  
Dois elefantes incomodam muito mais.  
Se um elefante incomoda muita gente  
Dois elefantes incomodam muito mais.
```

Fluxo da execução

1. Começa na primeira instrução do programa
2. As instruções são executadas por ordem sequencial
3. A **definição** de uma função não altera fluxo de execução
4. A **invocação** de uma função
 - 4.1 executa as instruções da definição por ordem
 - 4.2 no final regressa ao ponto de onde partiu
5. Funções podem chamar outras funções

Parâmetros e argumentos

- Normalmente as funções usam **argumentos**:

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

- O valor dos argumentos é associado a variáveis chamadas **parâmetros**

Exemplo

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

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

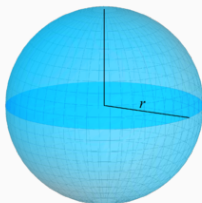
Funções que calculam resultados

Funções que calculam resultados

- As funções podem calcular resultados
- O resultado deve ser indicado com a instrução *return*
- A instrução *return* termina a função e define o resultado

Exemplo: calcular a área A de uma superfície esférica de raio r .

$$A = 4\pi r^2$$



Funções que calculam resultados (cont.)

```
import math

def area_esfera(r):
    A = 4 * math.pi * r**2
    return A
```

```
>>> area_esfera(1.0)
12.566370614359172
>>> area_esfera(1.5)
28.274333882308138
>>> area_esfera(2.0)
50.26548245743669
```

Âmbito das variáveis

- Os parâmetros duma função são **variáveis locais**—não são visíveis fora da função
- As variáveis definidas dentro da função também são locais

Exemplo:

```
>>> r = 42
>>> area_esfera(1)
12.566370614359172
>>> r
42
>>> A
NameError: name 'A' is not defined
```

Âmbito das variáveis (cont.)

As variáveis definidas fora das funções (**globais**) podem ser usadas dentro destas.

Exemplo: uma função para acrescentar a taxa de IVA a um preço.

```
taxa_IVA = 0.23
```

```
def precoFinal(valor):  
    return valor*(1+taxa_IVA)
```

É boa ideia documentar usando comentários e/ou *docstrings*.

```
taxa_IVA = 0.23 # taxa de IVA em percentagem
```

```
def precoFinal(valor):  
    '''Aumenta a taxa de IVA a um valor.  
    Usa a variável global taxa_IVA.'''  
    return valor*(1+taxa_IVA)
```

- Os comentários e *docstrings* são para quem lê o código
- As *docstrings* são também usadas pelo sistema de ajuda.

```
>>> help(precoFinal)
```

```
Help on function precoFinal in module __main__:
```

```
precoFinal(valor)
```

```
    Acrescenta a taxa de IVA a um valor.
```

```
    Usa a variável global taxa_IVA.
```

Return ou print?

return termina a execução da função e devolve um resultado

print apenas imprime um resultado

Só podemos usar um resultado se a função terminar com **return**:

```
def f(x):  
    return x*x  
  
print(f(f(3)))
```



81

```
def g(x):  
    print(x*x)  
  
print(g(g(3)))
```



9

Erro