

# Programação em Python

Tipos, Variáveis, Expressões, Funções

---

2023

Departamento de Ciência de Computadores



1. Tipos
2. Tipos básicos
3. Variáveis e atribuições
4. Programas completos
5. Definição de funções
6. Funções que calculam resultados

# Tipos

---

# 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	<code>int</code>	42 -7
vírgula-flutuante	<code>float</code>	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 fracionário<sup>1</sup>:

```
>>> 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
```

---

<sup>1</sup>Diferente nas versões de Python anteriores a 3.0

# Erros de arredondamento

Números inteiros podem ser representados de forma exata no computador.<sup>2</sup>

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.

<sup>2</sup>Apenas 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** `str + str ⇒ str`

**Repetição** `int * str ⇒ 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 acentos<sup>3</sup>
- 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	

## Variáveis (cont.)

Exemplos de nomes válidos para variáveis:

```
nome  idade  Preço_Max  área2
```

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

```
76trombones  more$  class
```

---

<sup>3</sup>Só 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*

**import** math

raio = 2.5

perimetro = 2\*math.pi\*raio

---

Executa correctamente, mas não mostra resultados!

# Comandos de entrada e saída de dados

`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*

**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*

- 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` 

`t = t + 10    # 10s extra de tempo` 

# Definição de funções

---

# Definição de funções

- Anteriormente vimos como usar as funções matemáticas pré-definidas
- Vamos agora ver como definir **novas funções**

## **Programação estruturada**

Decompor um problema em componentes mais simples até chegar às operações elementares.

# 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**

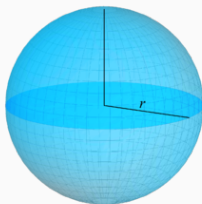
---

# 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)
```

---



## Documentação (cont.)

- 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