Nesta aula...

Conteúdo

1 Expressões, valores e tipos

Expressões e valores

- programas calculam *expressões* para produzir *valores*
- cálculo de expressões segue a estrutura de parênteses e as *prioridades* dos operadores
- o cálculo é efectuado da esquerda para a direita entre operadores da mesma prioridade:

$$\begin{array}{rl} & (1+2+3)*5-1 \\ \Rightarrow & (3+3)*5-1 \\ \Rightarrow & 6*5-1 \\ \Rightarrow & 30-1 \\ \Rightarrow & 29 \end{array}$$

Tipos

- os valores são classificados em diferentes *tipos*:

```
int    1  -33  29  102034
float  1.0  -17.0  3.14156  -1.25e2
str    "Ola mundo!"  'ABC'  'A'  '1.23.99'
```

- algumas operações só fazem sentido com valores de determinados tipos:

```
>>> "Ola mundo!" + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str'
and 'int' objects
```

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)
<type 'int'>

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

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

Tipos e operações aritméticas

As operações aritméticas operam com diferentes tipos númericos:

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

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

É permitido *misturar tipos* numa mesma operação:

```
>>> 1 + 2.0          # int + float => float
3.0
```

Operações sobre cadeias de caracteres

concatenação `str` + `str` ⇒ `str`

repetição `int` ∗ `str` ⇒ `str`

```
>>> 'Ola' + ' ' + 'Mundo'
'Ola Mundo'

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

Conversão automática entre tipos numéricos

$$\texttt{int} + \texttt{int} \Rightarrow \texttt{int}$$

$$\texttt{float} + \texttt{float} \Rightarrow \texttt{float}$$

$$\left.\begin{array}{l}\texttt{int} + \texttt{float} \\ \texttt{float} + \texttt{int}\end{array}\right\} \Rightarrow \texttt{float}$$

Também com os outros operadores aritméticos: `-`, `*`, `/`, `**`

Conversão explícita entre tipos

```
>>> int(2.71)
2

>>> int(-3.134)
-3

>>> float(-33)
-33.0

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

NB: `int(...)` efectua a *truncatura*, não o arredondamento!

Cuidados com tipos numéricos

Divisão entre inteiros dá o *quociente inteiro*:[1]

```
>>> 17/5
3
>>> 17.0/5.0
3.3999999999999
```

`//` dá sempre o *quociente inteiro*:

```
>>> 17//5
3
>>> 17.0/5.0
3.0
```

Cuidados com tipos numéricos

Erros de arredondamento em operações de vírgula flutuante:

[1] Isto poderá mudar em versões futuras da linguagem.

```
>>> (100.0/3 -33) * 3
1.0000000000000071
>>> 100.0 - 33*3
1.0
```

Mas algebricamente:

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

Mais tipos númericos

`long`: inteiros de precisão arbitrária[2] quando um resultado é demasiado grande para um inteiro normal e.g. $[-2^{31}, 2^{31} - 1]$:

```
>>> 2**100
1267650600228229401496703205376L
```

`complex`: números complexos com parte real e imaginária em vírgula flutuante: $re+im$`j`

```
>>> (1j)**2
(-1+0j)
>>> (2+1j) + (1-1j)
(3+0j)
```

Funções matemáticas

O módulo `math` tem definições de várias funções e constantes matemáticas. Para as usar, necessitamos de executar no ínicio da sessão ou do programa:

```
>>> import math
```

As constantes e funções num módulo são referidas com *módulo.nome*:

```
>>> math.sqrt(2)
1.4142135623730951
>>> math.pi
3.1415926535897931
>>> math.sin(math.pi/4)
0.70710678118654746
```

Funções matemáticas

Alternativa: podemos importar os nomes para o âmbito principal:

```
>>> from math import *
```

[2] Apenas limitada pela memória do computador!

Nesse caso podemos referir os nomes sem especificar o módulo:

```
>>> sqrt(2)
1.4142135623730951
>>> pi
3.1415926535897931
>>> sin(pi/4)
0.70710678118654746
```

Algumas funções matemáticas

acos	asin	atan	atan2	ceil	cos	degrees
e	exp	fabs	floor	hypot	log	log10
pi	pow	radians	sin	sqrt	tan	

Para obter mais informações:

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

2 Variáveis e atribuições

Variáveis

- nomes que representam *quantidades* ou *propriedades* dum problema
- começam com uma letra, seguido de letras, números ou sublinhado, p.ex.: `nome`, `idade`, `Preco_Max`, `area2`
- não podem ter acentos ou espaços
- 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	

Atribuições

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

$$nome = expressão$$

```
>>> pi = 3.1416
>>> raio = 1
```

$$\begin{array}{rcl} \texttt{pi} & \longrightarrow & 3.14156 \\ \texttt{raio} & \longrightarrow & 1 \end{array}$$

Atribuições

Podemos usar variáveis em atribuições subsequentes:

```
>>> perimetro = 2*pi*raio
>>> perimetro
6.2832
```

`pi`	$\longrightarrow$	3.14156
`raio`	$\longrightarrow$	1
`perimetro`	$\longrightarrow$	6.2832

Atribuições

A atribuição é um *comando*, não uma *equação*:

```
>>> raio = 2
>>> perimetro
6.2832
```

`pi`	$\longrightarrow$	3.14156
`raio`	$\longrightarrow$	*2*
`perimetro`	$\longrightarrow$	6.2832

Atribuições

Re-executando a atribuição:

```
>>> perimetro = 2*pi*raio
>>> perimetro
12.5664
```

`pi`	$\longrightarrow$	3.14156
`raio`	$\longrightarrow$	2
`perimetro`	$\longrightarrow$	*12.5664*

Programas

perimetro.py

```
# Calcular o perimetro de uma circunferência
import math

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

Executa correctamente, mas não mostra o resultado...

Comandos de entrada e saída de dados

input('prompt') lê um valor do teclado

raw_input('prompt') lê uma *string*

print expr1, expr2, ... escreve valores no terminal

Programa revisto

perimetro.py

```
# Calcular o perimetro de uma circunferência
import math

raio = input('Raio=?')
perimetro = 2*math.pi*raio

print 'Perimetro=', perimetro
```

Sumário

- resultado duma expressão é um *valor*
- valores são agrupados em *tipos*
- uma *atribuição* associa um valor a um nome
- uma sequência de comandos é um *programa*