Itaú Unibanco

Itaú

Programa de formação

# ITAú analytics

## Módulo I – Fundamentos Computacionais
## Sessão 2 - Aula 2 – Expressões e Operadores

Prof. Dr. Luiz Alberto Vieira Dias
Prof. Dr. Lineu Mialaret

# Expressões, Operadores e Precedência

- Nos slides anteriores, mostrou-se como nomes podem ser usados para se identificar objetos e como literais e construtores podem ser usados para se criar instâncias de classes pré-fabricadas (*built-in classes*)
  - Os nomes existentes podem ser combinados em expressões sintáticas maiores usando-se os operadores
- A semântica de um operador depende do tipo dos operandos envolvidos
  - Quando `a` e `b` são números, a expressão `a + b` sinaliza ser uma adição
  - Quando `a` e `b` são strings, a expressão `a + b` sinaliza uma concatenação dos operandos

# Operadores Lógicos

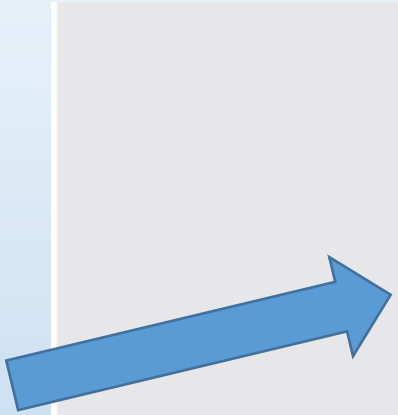
- A linguagem Python suporta os seguintes operadores para valores lógicos:

| | |
|---|---|
| **not** | unary negation |
| **and** | conditional and |
| **or** | conditional or |

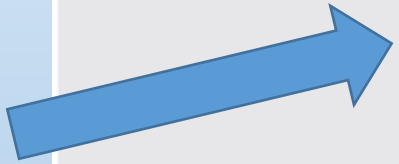
- Os operadores and e or implementam a avaliação curto-circuito (*short-circuit)*
  - Eles não avaliam o segundo operando se o resultado pode ser determinado baseado no valor do primeiro operando

# Operadores Lógicos (cont.)

- Avaliação curto-circuito
  - Na expressão (A and B), se A é falso, então a expressão é falsa e não há necessidade de se avaliar B
  - Na expressão (A or B), se A é verdadeiro, então a expressão é verdadeira e não há necessidade de se avaliar a B



| A | B | A and B |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |



| A | B | A or B |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

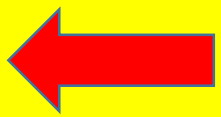
| A | not A |
|---|---|
| True | False |
| False | True |

# Operadores Lógicos (cont.)

- Avaliação curto-circuito

```
>>> x = 6
>>> y = 2
>>> x >= 2 and (x/y) > 2
True
>>> x = 1
>>> y = 0
>>> x >= 2 and (x/y) > 2
False
```

```
>>> x = 6
>>> y = 0
>>> x >= 2 and (x/y) > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```



```
count = int (input ("Enter the count:"))
sum = int (input("Enter the sum:"))
if count > 0 and sum/count > 10:
    print ("average > 10")
else:
    print ("count = 0 or average <= 10")
```

Short-circuit evaluation can be used to avoid division by zero

# Operadores de Igualdade

- A linguagem Python suporta os seguintes operadores de igualdade:

| | |
|---|---|
| **is** | same identity |
| **is not** | different identity |
| == | equivalent |
| != | not equivalent |

- A expressão `a is b` tem resultado `True` quando os identificadores `a` e `b` são sinônimos (*alias*) para o mesmo objeto
- A expressão `a == b` testa uma noção mais geral de equivalência
  - Se `a` e `b` apontam para o mesmo objeto, o resultado é `True`
  - A expressão `a == b` também resulta em `True` quando os identificadores referem-se a objetos diferentes que contém valores que podem ser considerados equivalentes

# Operadores de Igualdade (cont.)

- A noção precisa de equivalência depende do tipo de dado
  - Duas strings são equivalentes se elas são iguais caractere a caractere
  - Dois conjuntos são equivalentes se eles têm o mesmo conteúdo, independente da ordem

# Operadores de Igualdade (cont.)

- Testes de igualdade
  - Se as variáveis de referência apontam para o mesmo objeto, o operador `is` resulta em `True`
  - A expressão `a == b` também resulta em `True` quando os identificadores referem-se a objetos diferentes que contém valores que podem ser considerados equivalentes

```
>>> a = 12
>>> id(a)
1573240176
>>> b = 12
>>> id(b)
1573240176
>>> a is b
True
```

```
>>> conj1 = {1,2,3}
>>> conj2 = {1,2,3}
>>> id(conj1)
2129595788872
>>> id(conj2)
2129595788200
>>> conj1 ==conj2
True
```

# Operadores de Comparação

- Os tipos de dados já vistos podem definir um ordenamento natural por meio dos seguintes operadores relacionais:

| | |
|---|---|
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |

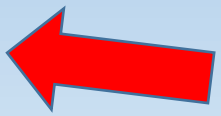
  - Estes operadores possuem um comportamento esperado para tipos numéricos e são definidos lexicograficamente e sensíveis ao caixa (*case-sensitively*) para strings

# Operadores de Comparação (cont.)

- Comparação de Strings
  - Exceções surgem quando os operandos são de tipos incompatíveis
  - Pode-se comparar strings e elas são comparadas de forma lexicográfica, ou seja, usando-se o valor ASCII dos caracteres

```
>>> "tim" == "tie"
False
>>> "free" != "freedom"
True
>>> "arrow" > "aron"
True
>>> "right" >= "left"
True
>>> "teeth" < "tee"
False
>>> "yellow" <= "fellow"
False
>>> "abc" > ""
True
```

```
>>> 5 < 'hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```



# Operadores Aritméticos

- A linguagem Python suporta os seguintes operadores aritméticos:

| | |
|---|---|
| + | addition |
| – | subtraction |
| * | multiplication |
| / | true division |
| // | integer division |
| % | the modulo operator |

- A utilização dos operadores de adição, subtração e multiplicação é trivial
  - Se os operandos são inteiros, o resultado será int
  - Se um dos operandos é do tipo float, o resultado será float

# Operadores Aritméticos (cont.)

- O operador / designa a divisão verdadeira (*true division*), retornando um resultado de ponto flutuante da operação
  - A expressão 27 / 4 resulta no valor em ponto flutuante 6.75
- O Python também suporta os operadores // e % para realizar os cálculos da divisão da seguinte forma
  - 27 // 4 resulta no valor int 6 (o valor da função piso do quociente)
  - 27 % 4 resulta no valor int 3, o resto da divisão inteira

```
>>> 27/4  # true division
6.75
>>> 27//4 # integer division
6
>>> 27%4 # module operator
3
>>> (27//4)*4 + (27%4)
27
```

# Operadores Bit a Bit

- O Pythom disponibiliza os seguintes operadores (*bitwise operators*) para operações em bits de operandos inteiros:

| | |
|---|---|
| ~ | bitwise complement (prefix unary operator) |
| & | bitwise and |
| \| | bitwise or |
| ^ | bitwise exclusive-or |
| << | shift bits left, filling in with zeros |
| >> | shift bits right, filling in with sign bit |

```
>>> a = 60
>>> print ("{0:b}".format(a))
111100
>>> a>>2
15
>>> b = a>>2
>>> print ("{0:b}".format(b))
1111
>>> c = b<<2
>>> print ("{0:b}".format(c))
111100
```

# Operadores para Sequências

- Os tipos sequenciais str, tuple, e list suportam as seguintes sintaxes:

| | |
|---|---|
| s[j] | element at index *j* |
| s[start:stop] | slice including indices [start,stop) |
| s[start:stop:step] | slice including indices start, start + step, start + 2*step, …, up to but not equalling or stop |
| s + t | concatenation of sequences |
| k * s | shorthand for s + s + s + ... (k times) |
| val in s | containment check |
| val not in s | non-containment check |

```
>>> str = "axaxaxaxaxax"
>>> x = str[1]
>>> print (x)
x
>>> print (str[0:2])
ax
>>> print (str[0:10:2])
aaaaa
>>> print (str[0:10:1])
axaxaxaxax
>>> str1 = "xxxxzzzz"
>>> print (str + str1)
axaxaxaxaxaxxxxxzzzz
>>> print (2*str1)
xxxxzzzzxxxxzzzz
>>> "x" in str1
True
>>> "a" not in str1
True
>>> "x" not in str1
False
```

# Operadores para Sets

- No Phyton, sets (e frozensets) suportam os seguintes operadores:

| | |
|---|---|
| key **in** s | containment check |
| key **not in** s | non-containment check |
| s1 == s2 | s1 is equivalent to s2 |
| s1 != s2 | s1 is not equivalent to s2 |
| s1 <= s2 | s1 is subset of s2 |
| s1 < s2 | s1 is proper subset of s2 |
| s1 >= s2 | s1 is superset of s2 |
| s1 > s2 | s1 is proper superset of s2 |
| s1 \| s2 | the union of s1 and s2 |
| s1 & s2 | the intersection of s1 and s2 |
| s1 – s2 | the set of elements in s1 but not s2 |
| s1 ^ s2 | the set of elements in precisely one of s1 or s2 |

```
>>> conj1 = {1,2,3,4}
>>> conj2 = {1,2}
>>> conj3 = {1,2,3,4}
>>> "1" in conj1
False
>>> "5" not in conj1
True
>>> conj1 == conj2
False
>>> conj2 <= conj1
True
```

# Operadores para Dictionaries

- Os operadores suportados para objetos do tipo dict são os seguintes:

| | |
|---|---|
| d[key] | value associated with given key |
| d[key] = value | set (or reset) the value associated with given key |
| **del** d[key] | remove key and its associated value from dictionary |
| key **in** d | containment check |
| key **not in** d | non-containment check |
| d1 == d2 | d1 is equivalent to d2 |
| d1 != d2 | d1 is not equivalent to d2 |

```
>>> d1 = {"mike":41, "bob":3}
>>> d2 = {"bob":3, "mike":41}
>>> "mike" in d1
True
>>> "mike" not in d1
False
>>> d1 == d2
True
>>> print (d1["mike"])
41
```

# Operadores de Atribuição Estendidos

- Python suporta o operador de atribuição estendida para a maioria dos operadores binários

  +=5 é um atalho para count = count + 5

  - Para tipos de dados imutáveis, esta sintaxe não altera o valor do objeto e sim cria um novo objeto apontado pelo identificador

```
>>> count = 10
>>> id (count)
1573240112
>>> count +=5
>>> id (count)
1573240272
>>> count -=3
>>> id (count)
1573240176
```

# Operadores de Atribuição Estendidos (cont.)

- Entretanto é possível para um tipo alterar o objeto com este operador
- Classe list faz isto

```
>>> alpha = [1, 2, 3]
>>> id(alpha)
1957522902408
>>> beta = alpha    # an alias for alpha
>>> print (beta)
[1, 2, 3]
>>> beta += [4, 5]     # extends the original list with two more elements
>>> print (beta)
[1, 2, 3, 4, 5]
>>> id(beta)
1957522902408
>>> beta = beta + [6, 7] # reassigns beta to a new list [1, 2, 3, 4, 5, 6, 7]
>>> print (beta)
[1, 2, 3, 4, 5, 6, 7]
>>> id(beta)
1957522900616
>>> print(alpha)    # will be [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

# Precedência de Operadores

- As linguagens de programação têm regras claras para a ordem com que expressões tais como 5 + 2 * 3 são avaliadas
- A ordem formal de precedência para operadores no Python é apresentada ao lado
  - Operadores com maior precedência são avaliados antes dos com mais baixa precedência

| Operator Precedence | | |
|---|---|---|
| | **Type** | **Symbols** |
| 1 | member access | expr.member |
| 2 | function/method calls<br>container subscripts/slices | expr(...)<br>expr[...] |
| 3 | exponentiation | ** |
| 4 | unary operators | +expr, −expr, ~expr |
| 5 | multiplication, division | *, /, //, % |
| 6 | addition, subtraction | +, − |
| 7 | bitwise shifting | <<, >> |
| 8 | bitwise-and | & |
| 9 | bitwise-xor | ^ |
| 10 | bitwise-or | \| |
| 11 | comparisons<br>containment | **is**, **is not**, ==, !=, <, <=, >, >=<br>**in**, **not in** |
| 12 | logical-not | **not** expr |
| 13 | logical-and | **and** |
| 14 | logical-or | **or** |
| 15 | conditional | val1 **if** cond **else** val2 |
| 16 | assignments | =, +=, −=, *=, etc. |

# Precedência de Operadores (cont.)

- A expressão

  5 + 2 * 3 é avaliada como 5 + (2*3) = 11

- Operadores dentro de uma categoria são avaliados da esquerda para a direita
  - 5 – 2 + 3 = 6
- Exceções a regra incluem o operadores unários e exponenciação
  - Avaliados da direita para esquerda

| Operator Precedence | | |
|---|---|---|
| | **Type** | **Symbols** |
| 1 | member access | expr.member |
| 2 | function/method calls<br>container subscripts/slices | expr(...)<br>expr[...] |
| 3 | exponentiation | ** |
| 4 | unary operators | +expr, −expr, ~expr |
| 5 | multiplication, division | *, /, //, % |
| 6 | addition, subtraction | +, − |
| 7 | bitwise shifting | <<, >> |
| 8 | bitwise-and | & |
| 9 | bitwise-xor | ^ |
| 10 | bitwise-or | \| |
| 11 | comparisons<br>containment | **is**, **is not**, ==, !=, <, <=, >, >=<br>**in**, **not in** |
| 12 | logical-not | **not** expr |
| 13 | logical-and | **and** |
| 14 | logical-or | **or** |
| 15 | conditional | val1 **if** cond **else** val2 |
| 16 | assignments | =, +=, −=, *=, etc. |

# Precedência de Operadores (cont.)

- A linguagem Python permite a atribuição encadeada (*chained assignement*)

  x = y = 0

- Permite também o encadeamento de operadores de comparação

  1 <= x + y <= 10 é avaliada como (1<=x+y) e (x+y<=10)

```
>>> # chained assignment
>>> x = y = 0
>>> print (x,y)
0 0
>>> # Python code to illustrate
... # chaining comparison operators
... x = 5
>>> print(1 < x < 10)
True
>>> print(10 < x < 20 )
False
>>> print(x < 10 < x*10 < 100)
True
>>> print(10 > x <= 9)
True
>>> print(5 == x > 4)
```

# Precedência de Operadores (cont.)

- Tomar cuidado com a atribuição encadeada!

```
>>> # chained assignment
>>> a = b = []
>>> a.append(5)
>>> print(b)
[5]
>>> a is b
True
>>> print (a)
[5]
```