IBGE
Instituto Brasileiro de Geografia e Estatística

Escola Nacional de
Ciências Estatísticas

Introdução à Programação

Aula 04: Programação em Python:
Operadores Relacionais e Lógicos

Prof. Eduardo Corrêa

Data 19/03/2024

Tópicos da Aula

- Temas desta aula:
  - Conhecendo as funções built-in
  - Operadores Relacionais
  - Operadores Lógicos

Funções Built-in – Introdução (1/8)

- Além das funções que estão dentro de módulos (como o módulo math), existem algumas poucas funções que você pode usar **sem precisar importar** nenhum **módulo**.
- Elas são chamadas de **funções built-in**.
  - Nos próximos slides, vamos comentar apenas 4 delas (existem outras, mas deixaremos para aulas futuras):
    - round()
    - int()
    - float()
    - str()

Funções Built-in – Introdução (2/8)

```
# int() converte um valor str para inteiro
# float() converte um valor str para real(float)

x  = '1010'
y  = '0.5'
print(int(x), float(y))
```

Saída:

```
1010  0.5
```

- Se o valor armazenado na string não puder ser convertido, ocorrerá um erro e o programa encerrará.

```
x  = 'abcd'
print(float(x))
```

Saída:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: 'abcd''
```

Funções Built-in – Introdução (3/8)

```
# round(x, d): retorna um real x arredondado para d casas
# decimais. Se d for omitido, arredonda p/ inteiro

x  = 2.76543
print(round(x,3)); print(round(x,2)); print(round(x))
print()
print(round(1.49)); print(round(1.5))
```

Saída:

```
2.765
2.77
3

1
2
```

- Função importante, pois o módulo math **na versão do Python instalada no laboratório** não possui uma função que realize essa operação.
  - Existe: math.ceil(), math.floor() e math.trunc(), que não são iguais.

Funções Built-in – Introdução (4/8)

- A conversão é **necessária** quando realiza-se a entrada de via **input()** de dados que serão tratados como números no programa.
  - idade
  - salário
  - taxa de juros
  - nota em uma prova
  - ...

- Isso porque, todo dado que entra via **input()** é recebido como uma string.
  - E **não** é possível executar operações aritméticas sobre strings !!!!

Funções Built-in – Introdução (5/8)

- Exemplos:

CERTO:

```
n = int(input("Digite um número inteiro: "))
print("O dobro é:", n * 2)
```

```
>>>
Digite um número inteiro: 5
O dobro é: 10
```

CERTO TAMBÉM:

```
n = input("Digite um número inteiro: ")
n = int(n)
print("O dobro é:", n * 2)
```

```
>>>
Digite um número inteiro: 5
O dobro é: 10
```

Funções Built-in – Introdução (6/8)

- Exemplos (*cont...*):

ERRADO !!! (sem conversão da entrada):

```
n = input("Digite um número inteiro: ")
print("O dobro é:", n * 2)
```

```
>>>
Digite um número inteiro: 5
O dobro é: 55
```

- Por que deu errado?
  - Como *n* não foi convertido de string para int, a operação *n * 2* foi realizada como uma operação de **repetição de strings** (e não multiplicação de números)
  - O valor de *n* foi repetido 2 vezes, gerando como resultado a string '55'

Funções Built-in – Introdução (7/8)

- A função **str()** faz o contrário. Ela converte um valor numérico (int ou float) para uma string (str).
- É necessária, por exemplo, quando você desejar realizar a operação de **concatenação**, uma vez que só é possível concatenar strings (e nunca uma string com um número).

CERTO:

```
nome = "Jane"; idade = 41
msg = "A idade de " + nome + " é " + str(idade)
```

ERRADO:

```
nome = "Jane"; idade = 41
msg = "A idade de " + nome + " é " + idade
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Funções Built-in – Introdução (8/8)

- Existem cerca de **70 funções** built-in no Python.
  - Em aulas futuras, apresentaremos **apenas as mais importantes**.
  - Você já conhece 6 delas: print(), input(), round(), int(), float() e str().
  - A função print() também será melhor detalhada em uma aula futura, pois ela possui muitos recursos úteis que ainda não foram mostrados.

Operadores Relacionais (1/4)

- Como vimos na aula de algoritmos as **instruções de seleção** e de **repetição** tomam uma decisão a partir da avaliação de uma condição
  - Uma condição representa uma comparação ou conjunto de comparações que irá resultar sempre em VERDADEIRO (True) ou FALSO (False).
  - Os operadores básicos utilizados nas condições – denominados **operadores relacionais** – são os seguintes**:**

Operadores Relacionais	
==	**Igual**
!=	**Diferente**
>	**Maior**
<	**Menor**
>=	**Maior ou Igual**
<=	**Menor ou Igual**

Operadores Relacionais (2/4)

- Os operadores relacionais permitem que seja realizada uma comparação entre dois valores.
  - Estes valores poderão ser variáveis, expressões aritméticas ou literais (valores).

0 > 1	*resulta em False*
4 * 2 == 8	*resulta em True*
9 ** 2 != 81	*resulta em False*
5 + 2 < 7	*resulta em False*
5 + 2 <= 7	*resulta em True*
7 % 2 != 0	*resulta em True*

Operadores Relacionais (3/4)

- Como dito no slide anterior, é possível realizar a comparação de valores armazenados em variáveis.
  - De fato, é isso que você fará para implementar testes nos comandos de desvio e repetição.

media >= 7.0	*lê-se: o conteúdo da variável media é maior ou igual a 7.0?*
a > b	*lê-se: o conteúdo da variável a é maior do que o conteúdo da variável b?*
pais == 'BRASIL'	*lê-se: o conteúdo da variável pais é igual a 'BRASIL' ("pais" é uma variável string!)*

Operadores Relacionais (4/4)

- O resultado de um teste pode ser atribuído a uma variável booleana, se o programador desejar.

```
x = 0; y = 1

v1  = (x < y)          # v1 recebe True
v2  = (y % 2 == x)     # v2 recebe False
v3  = (y // 2 == x)    # v3 recebe True

print('v1: ',v1 )
print('v2: ',v2 )
print('v3: ',v3 )
```

```
>>>
v1:   True
v2:   False
v3:   True
```

Operadores Lógicos (1/4)

- Os operadores lógicos **and** (e), **or** (ou) podem ser utilizados para **combinar pares de condições**.
  - Com isto, testes mais complexos podem ser elaborados.

Operadores Lógicos and e or
and : a sentença é verdadeira se AMBAS as condições forem verdadeiras.
or : a sentença é verdadeira se UMA das condições for verdadeira.

  - Há também o operador not (não), que inverte o valor lógico de uma sentença:
    - (True -> False, False ->True)

Operadores Lógicos (2/4)

- A **tabela verdade** exprime todas as combinações possíveis entre pares de valores lógicos.

Tab. Verdade AND	*Resultado*
True and True	True
True and False	False
False and True	False
False and False	False

Tab. Verdade NOT	*Resultado*
not True	False
not False	True

Tab. Verdade OR	*Resultado*
True or True	True
True or False	True
False or True	True
False or False	False

Operadores Lógicos (3/4)

- Exemplo de programa com operadores lógicos **or** e **and**.

```
a = 10; b =5

v1 = (b < a) and (b > 0)     # v1 recebe True
v2 = (b > a) or (b < 0)      # v2 recebe False

print('v1: ',v1 ); print('v2: ',v2 )
```

```
>>>
v1:   True
v2:   False
```

Operadores Lógicos (4/4)

- Exemplo do uso do operador **not**.
- Esse operador simplesmente inverte o resultado de uma condição.
  - Ex: **not** (1 > 2) resulta em True, pois 1 > 2 é False.
  - O not é um **operador unário**, pois ao contrário do and e do or, não é aplicado sobre um par de condições, mas sim sobre uma única.

```
a  = 1; b =2

v1 = not(a < b) and (b > 0)          # v1 recebe False
v2 = not(a > b) and (not(a < 0))     # v2 recebe True

print('v1: ',v1 ); print('v2: ',v2 )
```

```
>>>
v1:   False
v2:   True
```

Resolução de Expressões Condicionais

- Para poder resolver uma expressão condicional (expressão cujo resultado final será True ou False), o Python aplica uma tabela de prioridades.

Tabela de Prioridades do Python
1. Efetuar operações embutidas entre parênteses "mais internos".
2. Efetuar funções (qualquer função, como por exemplo alguma do módulo math).
3. Resolver **
4. Resolver *, /, //, %
5. Resolver +, -
6. Resolver os operadores relacionais: ==, !=, >, <, >=, <=
7. Resolver o operador lógico not
8. Resolver o operador lógico and
9. Resolver o operador lógico or

- Essa tabela é chata de memorizar... Mas veremos que há uma saída simples!

Simulação (Execução de Programa)

- Resolução de expressão lógica em Python.
- Qual será o valor de V2?

```
x = -3
y = 9
v1 = False

v2 = (x ** 2 == y) or (not v1) and (x > y % 3 + 1)

print(v2)
```

Tabela de Prioridades
1: parênteses.
2: funções.
3: **
4: *, /, //, %
5: +, -
6: ==, !=, >, <, >=, <=
7: not
8: and
9: or

Simulação (Execução de Programa)

- Seguindo a tabela de prioridades, o Python tentará resolver primeiro o que está entre parênteses.

```
x = -3
y = 9
v1 = False

v2 = (x ** 2 == y) or (not v1) and (x > y % 3 + 1)

print(v2)
```

Tabela de Prioridades
1: parênteses.
2: funções.
3: **
4: *, /, //, %
5: +, -
6: ==, !=, >, <, >=, <=
7: not
8: and
9: or

Simulação (Execução de Programa)

```
x = -3
y = 9
v1 = False

v2 = (x ** 2 == y) or (not v1) and (x > y % 3 + 1)

print(v2)
```

Tabela de Prioridades
1: parênteses.
2: funções.
3: **
4: *, /, //, %
5: +, -
6: ==, !=, >, <, >=, <=
7: not
8: and
9: or

Como há 3 expressões entre parênteses, 1º resolve-se a que está mais à esquerda, sempre seguindo a tabela de prioridades.

```
v2 =(x ** 2 == y) or (not v1) and (x > y % 3 + 1)
v2 =(-3 ** 2 == 9) or (not v1) and (x > y % 3 + 1)
v2 =(-3 ** 2 == 9) or (not v1) and (x > y % 3 + 1)
v2 =(9 == 9) or (not v1) and (x > y % 3 + 1)
v2 = True or (not v1) and (x > y % 3 + 1)
```

Simulação (Execução de Programa)

```
x = -3
y = 9
v1 = False

v2 = True or (not v1) and (x > y % 3 + 1)

print(v2)
```

Tabela de Prioridades
1: parênteses.
2: funções.
3: **
4: *, /, //, %
5: +, -
6: ==, !=, >, <, >=, <=
7: not
8: and
9: or

O que estava no 1º parênteses foi resolvido. Agora o Python resolve o 2º

```
v2  = True or (not v1) and (x > y % 3 + 1)
v2  = True or (not False) and (x > y % 3 + 1)
v2  = True or (not False) and (x > y % 3 + 1)
v2  = True or True and (x > y % 3 + 1)
```

Simulação (Execução de Programa)

```
x = -3
y = 9
v1 = False

v2 = True or True and (x > y % 3 + 1)

print(v2)
```

Tabela de Prioridades
1: parênteses.
2: funções.
3: **
4: *, /, //, %
5: +, -
6: ==, !=, >, <, >=, <=
7: not
8: and
9: or

E agora a terceira expressão entre parênteses. Sempre respeitando ordem da tabela de prioridades.

```
v2 = True or True and (x > y % 3 + 1)
v2 = True or True and (-3 > 9 % 3 + 1)
v2 = True or True and (-3 > 9 % 3 + 1)
v2 = True or True and (-3 > 0 + 1)
v2 = True or True and (-3 > 1)
v2 = True or True and False
```

Simulação (Execução de Programa)

```
x = -3
y = 9
v1 = False

v2 = True or True and False

print(v2)
```

Tabela de Prioridades
1: parênteses.
2: funções.
3: **
4: *, /, //, %
5: +, -
6: ==, !=, >, <, >=, <=
7: not
8: and
9: or

Agora só existem operadores booleanos a serem resolvidos. O Python vai resolver primeiro o AND e depois o OR (*veja a tab. de prioridades!!!*)

```
v2 = True or True and False
v2 = True or True and False
v2 = True or False
v2 = True
```

Resolução de Expressões Condicionais

- * * RECOMENDAÇÃO * *: como a tabela de prioridades é difícil de decorar, use sempre parênteses para explicitar ao Python a forma com que se deseja resolver a expressão.

- Ex.: Suponha que um aluno é aprovado se:

  - Tem nota na prova maior ou igual a 6.5 e nota do teste maior ou igual a 7.0. (*as duas coisas tem que ser verdade!!*)

  - ou se possui nota na prova maior do que 7.0 independentemente da nota no teste.

  - Você pode montar o teste dessa forma:

```
((prova >= 6.5) and (teste >= 7.0)) or (prova > 7.0)
```