Programação I
Aula 5 — Ciclos e condicionais

Pedro Vasconcelos
DCC/FCUP

Nesta aula

Na aula passada: vimos como repetir instruções usando ciclos *for*.

Nesta aula: vamos ver mais em detalhe diferentes formas de ciclos e execução condicional.

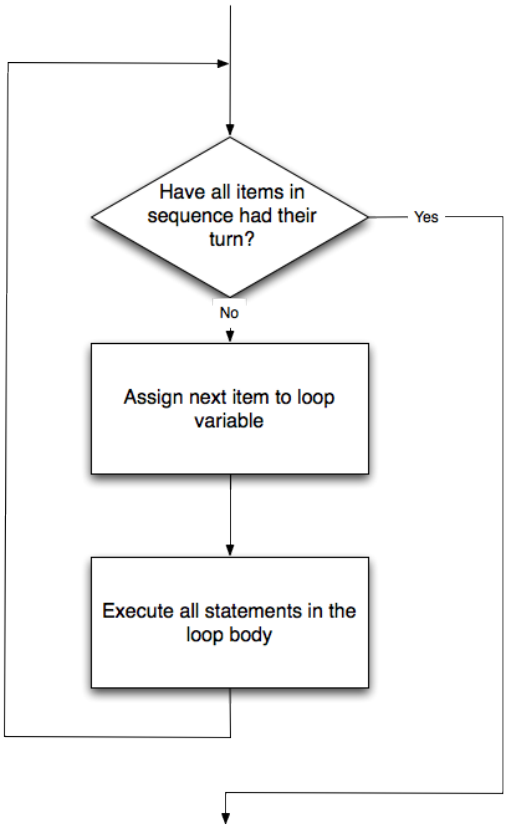
1. Ciclos for
2. Execução condicional
3. Ciclos while

Ciclos *for*

```
for variável in lista de valores:
    instrução 1
    instrução 2
    ⋮
    instrução n
resto do programa
```

1. O corpo do ciclo está indentado
2. Enquanto não percorremos todos os valores:
   - a variável toma o próximo valor na lista;
   - executamos o corpo do ciclo.
3. Depois do último valor: a execução continua no resto programa

Fluxo de execução de um ciclo *for*



Exemplos

```
amigos = ["Ana","João","Pedro","Beatriz"]
for nome in amigos:
    mensg = "Olá, " + nome + "!"
    print(mensg)
```

```
Olá, Ana!
Olá, João!
Olá, Pedro!
Olá, Beatriz!
```


Exemplos (cont.)

```
import math
for x in [0,1,2,3,4,5]:
    print(x, math.sqrt(x))
```

```
0 0.0
1 1.0
2 1.4142135623730951
3 1.7320508075688772
4 2.0
5 2.23606797749979
```

Função `range`

Muitas vezes queremos efectuar um ciclo sobre valores numéricos em progressão aritmética (exemplo: $0, 1, 2, 3, \ldots$)

A função *range* permite facilmente gerar valores desta forma.

Função `range` (cont.)

```
for x in range(5):   # 0, 1, 2, 3, 4
    print(x)

for x in range(10): # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    print(x)

for x in range(3,10): # 3, 4, 5, 6, 7, 8, 9
    print(x)

for x in range(3,10,2): # 3, 5, 7, 9
    print(x)
```

Função `range` (cont.)

range(n)	valores inteiros de 0 até $n-1$ inclusivé;
range(i,n)	valores inteiros de i até $n-1$ inclusivé;
range(i,n,d)	valores inteiros i, $i+d, i+2d, \ldots$ inferiores a n.

Note que `range(n)` inclui o zero mas não inclui o n.

Os programadores preferem contar do zero!

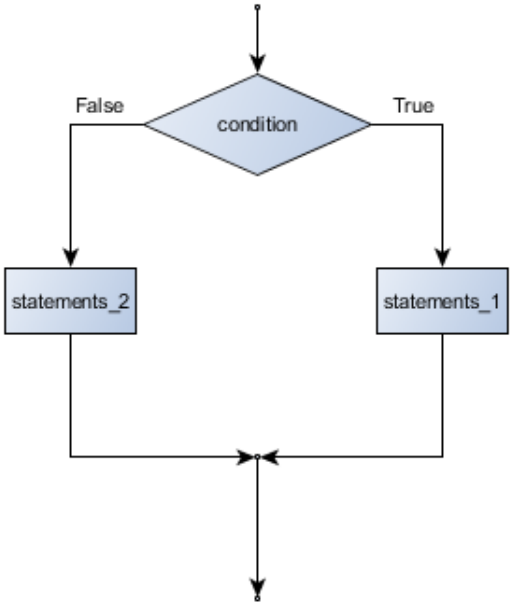
Exemplo

```
import math
for x in range(10,110,10):
    print(x, math.log10(x))
```

```
10 1.0
20 1.3010299956639813
30 1.4771212547196624
40 1.6020599913279625
50 1.6989700043360187
60 1.7781512503836436
70 1.845098040014257
80 1.9030899869919435
90 1.9542425094393248
100 2.0
```

Execução condicional

```
if condição:
    instruções 1
else:
    instruções 2
```



- A expressão na linha do `if` é a condição
- O bloco após o `if` é executado se a condição for verdadeira
- O bloco após o `else` é executado se a condição for falsa

Condições

`==`	igual
`!=`	diferente
`>`	maior
`<`	menor
`>=`	maior ou igual
`<=`	menor ou igual

O resultado é um valor lógico (`True` ou `False`).

Condições (cont.)

Exemplos:

```
>>> 1+2 == 3
True
>>> 1+2 > 2+3
False
>>> 'a' != 'A'
True
>>> 'B' < 'A'
False
```


Exemplo

```
for x in range(5):
    if x%2 == 0:
        print(x, "é par")
    else:
        print(x, "é ímpar")
```

Resultado

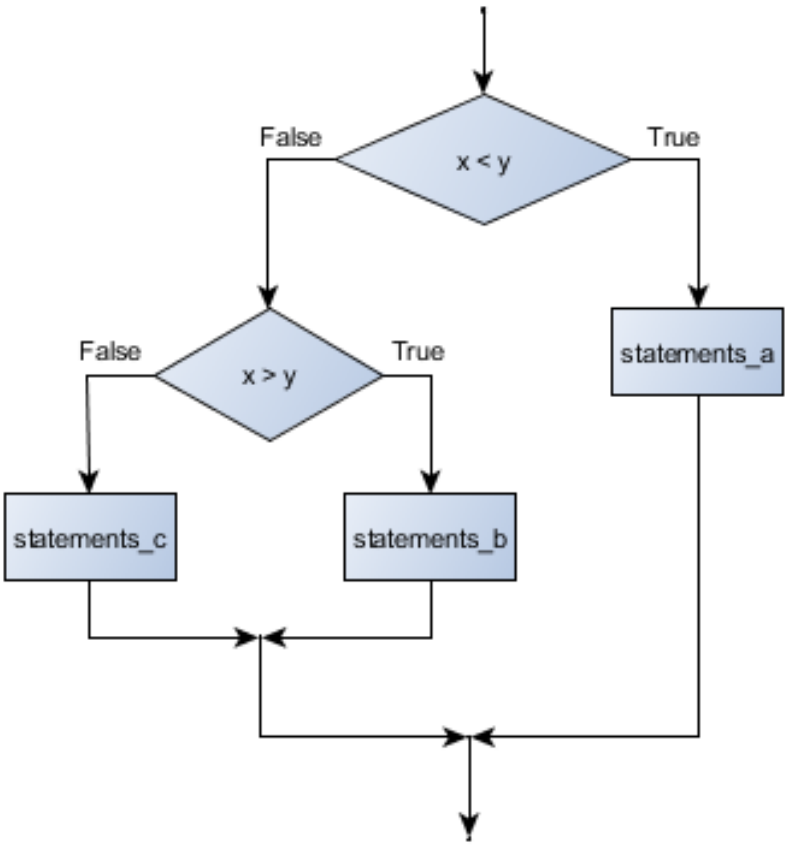
```
0 é par
1 é impar
2 é par
3 é impar
4 é par
```

If-else dentro de *if-else*

```
if x < y:
    print(x, "é menor que", y)
else:
    if x > y:
        print(x, "é maior que", y)
    else:
        print(x, "e", y, "são iguais")
```

- A indentação indica a estrutura das condições
- Pode ser difícil de ler com mais do que dois níveis

If-else dentro de *if-else* (cont.)

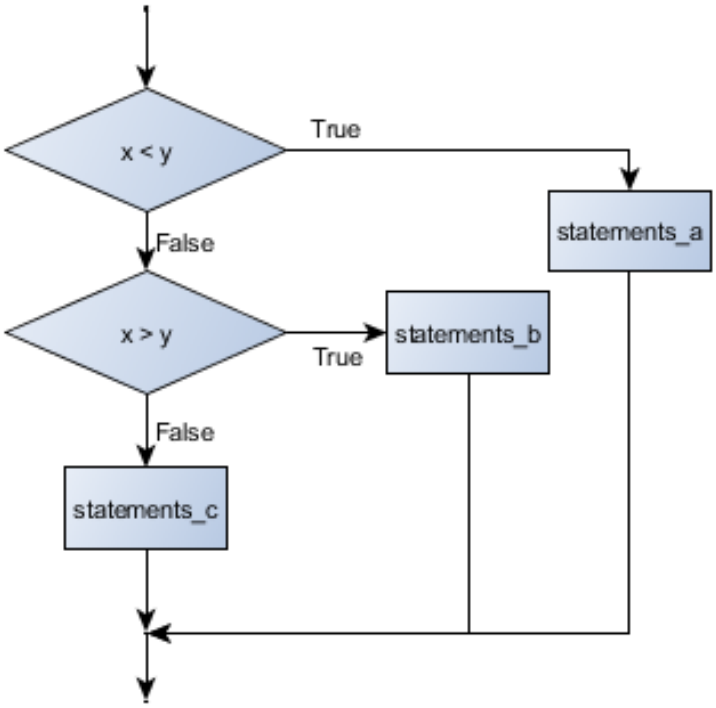


If-else encadeados

```
if x < y:
    print(x, "é menor que", y)
elif x > y:
    print(x, "é maior que", y)
else:
    print(x, "e", y, "são iguais")
```

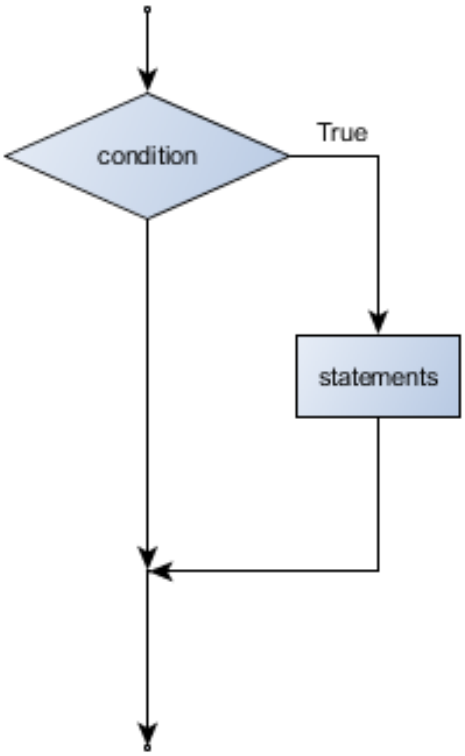
- O `elif` substitui o `else...if`
- Apenas um nível de indentação

If-else encadeados (cont.)



Omitir o bloco `else`

```
if x < 0:
    print(x, "é negativo")
```



Conectivas lógicas

`and`	ambas as condições são verdadeiras
`or`	pelo menos uma das condições é verdadeira
`not`	a condição é falsa

```
>>> import math

>>> math.pi>3 and math.pi<4
True

>>> math.pi<3 or math.pi==3
False

>>> not (math.pi<3)
True
```

Simplificar condições

Exemplo:

```
if not (idade>=18):
    print("Não tem idade para conduzir!")
```

é equivalente a

```
if idade<18:
    print("Não tem idade para conduzir!")
```


Simplificar condições (cont.)

Mais geralmente, podemos simplificar a negações usando equivalências:

```
not (A == B)   ⟺   A != B
 not (A < B)   ⟺   A >= B
not (A <= B)   ⟺   A > B
```

⋮

etc.

Simplificar condições (cont.)

Podemos simplificar a negações de conetivas lógicas.

```
if not (energia>=0.90 and escudo>=100):
    print("O ataque não surte efeito.")
else:
    print("O dragão morre!")
```

é equivalente a

```
if energia<0.90 or escudo<100:
    print("O ataque não surte efeito.")
else:
    print("O dragão morre!")
```


Simplificar condições (cont.)

Mais geralmente:

```
not (P and Q)   ⟺   (not P) or (not Q)
not (P or Q)    ⟺   (not P) and (not Q)
not (not P)     ⟺   P
```

Simplificar condições (cont.)

Podemos trocar a ordem dos blocos *if-else*.

```
if not (energia>=0.90 and escudo>=100):
    print("O ataque não surte efeito.")
else:
    print("O dragão morre!")
```

é equivalente a

```
if energia>=0.90 and escudo>=100:
    print("O dragão morre!")
else:
    print("O ataque não surte efeito.")
```


Ciclos *while*

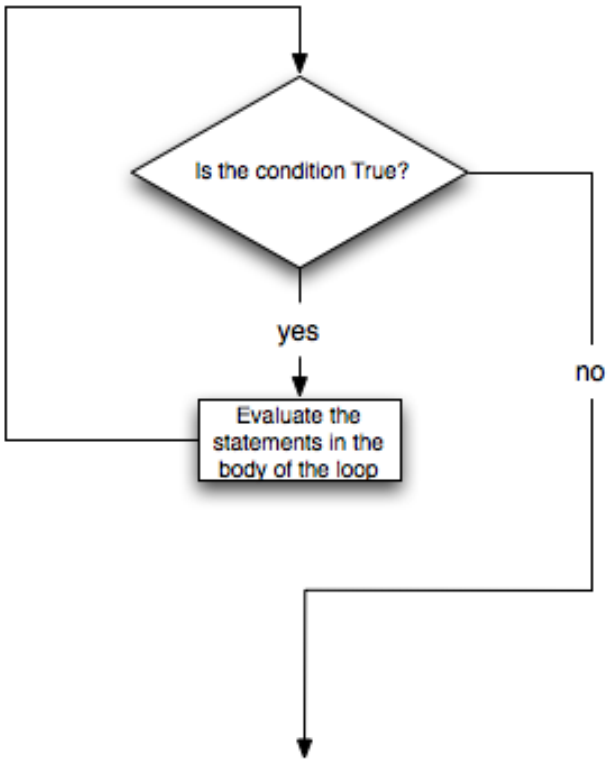
No ciclo *for* especificamos a lista de valores a percorrer.

Por vezes necessitamos de iterar sem saber *a priori* quantas vezes vamos executar o ciclo.

Para esses casos podemos usar um ciclo *while*.

Ciclos *while* (cont.)

```
while condição:
    instrução 1
    instrução 2
    ⋮
    instrução n
resto do programa
```



Exemplo

Encontrar o primeiro natural *n* tal que

$$1 + 2 + \cdots + n > 1000$$

```
n = 0                  # limite superior da soma
s = 0                  # valor da soma 1+2+...+n
while s <= 1000:       # enquando a soma não ultrapassa 1000
    n = n+1            # mais um natural
    s = s+n            # actualiza a soma
print(n)               # imprimir o número que encontrou
```