Programação em Python

Ciclos e Execução condicional

2023

Departamento de Ciência de Computadores

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Conteúdo

Ciclos for

Ciclos *for*

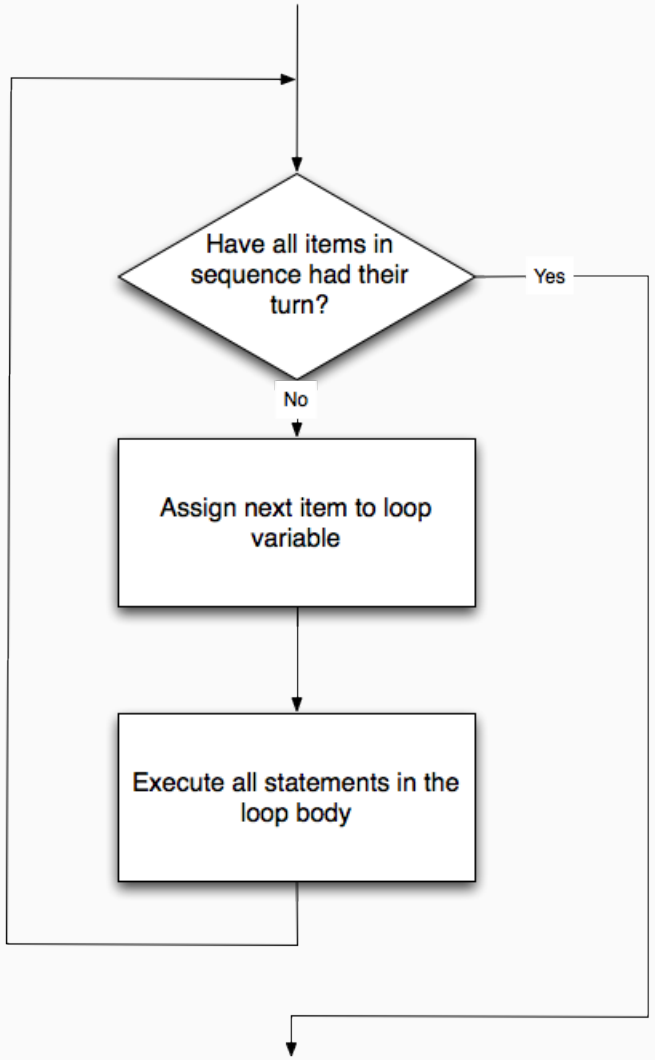
```
for variável in lista de valores:
    instrução 1
    instrução 2
    ⋮
    instrução n
resto do programa
```

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

Iteração

- Repetir instruções um número variável de vezes
- Para exprimir computação, tabelar uma função, etc.

Fluxo de execução de um ciclo *for*



Exemplos

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

Resultado

```
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))
```

Resultado

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

Ciclos for (resumo)

```
for variável in sequência:
    ⋮
    instruções a repetir
    ⋮
```

- Repete instruções (itera) com a variável a tomar sucessivos valores da sequência
- O número de iterações está limitado pelo comprimento da sequência

Função `range` (progressões aritméticas)

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` - Progressões aritméticas

A função `range(a,b,k)` gera sequências em progressão aritmética

$$a,\ a+k,\ a+2k,\ \ldots$$

cujos valores são menores que b.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
```

```
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)
```

`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, $i+d$, $i+2d$, ... 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))
```

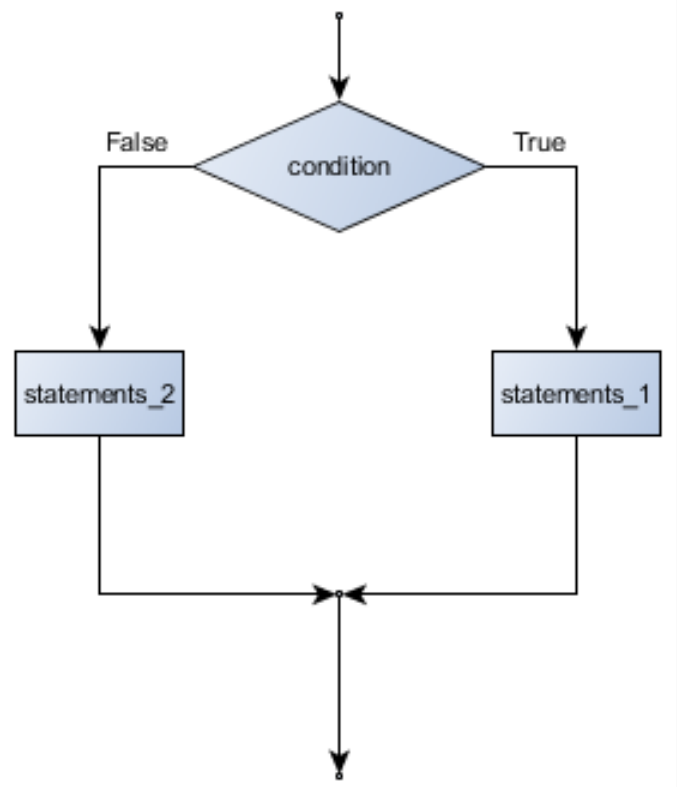
Resultado

```
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

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

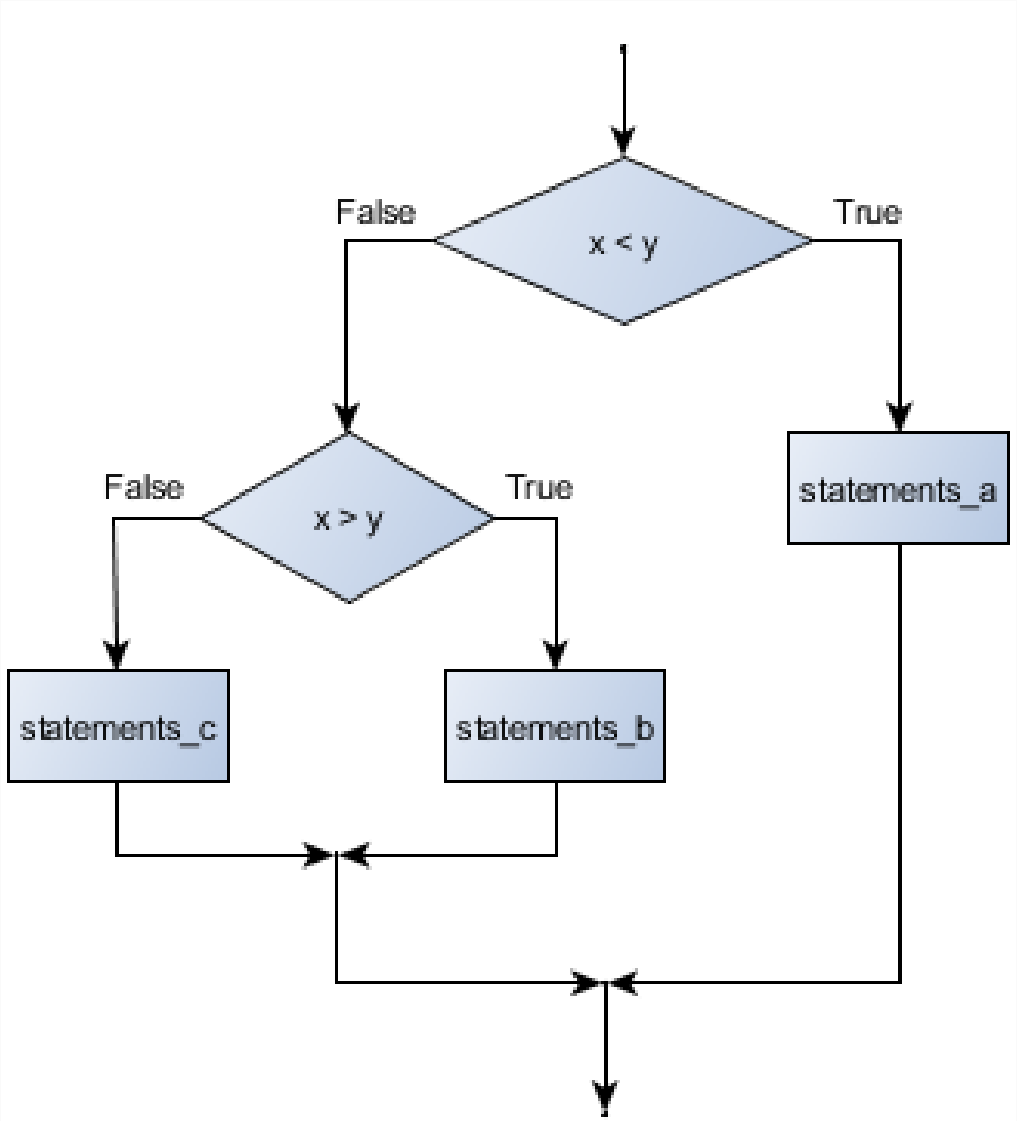
```
0 é par
1 é impar
2 é par
3 é impar
4 é par
```

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.)

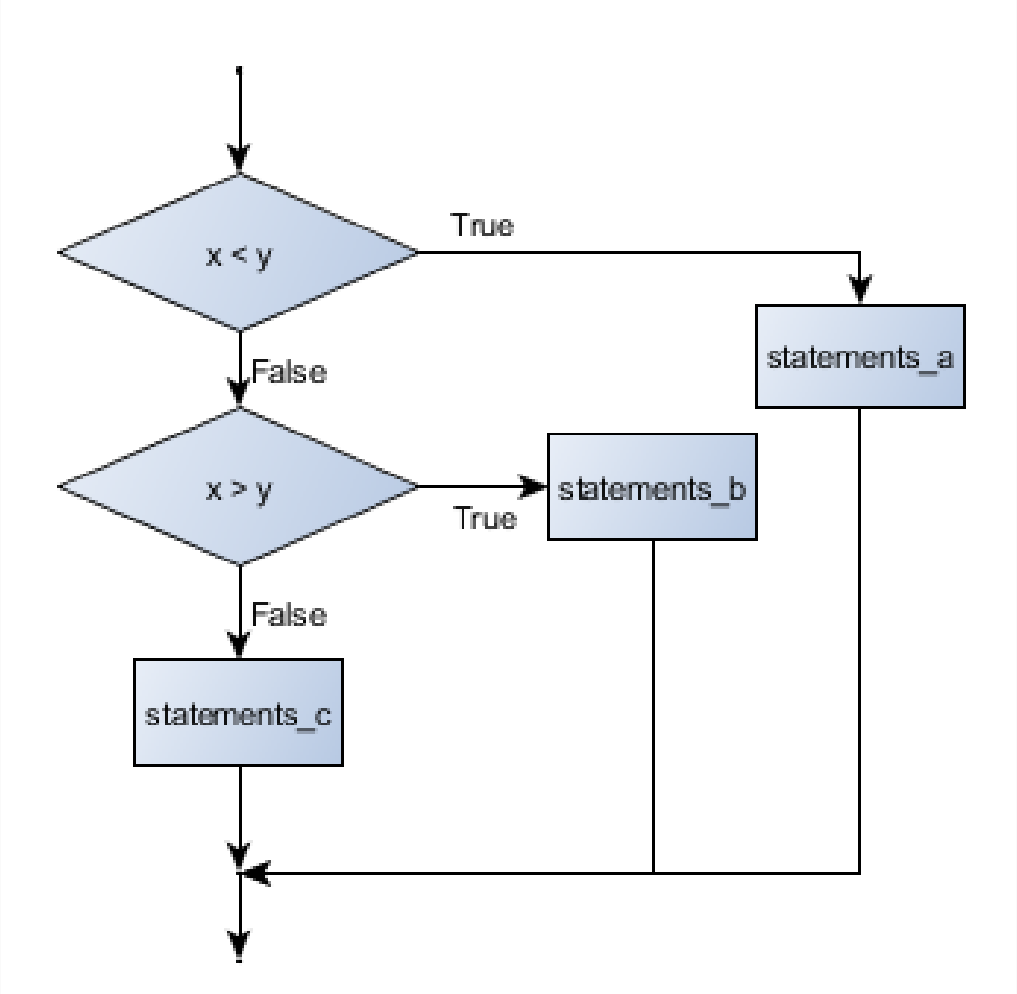


If-else encadeados

```
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")
```

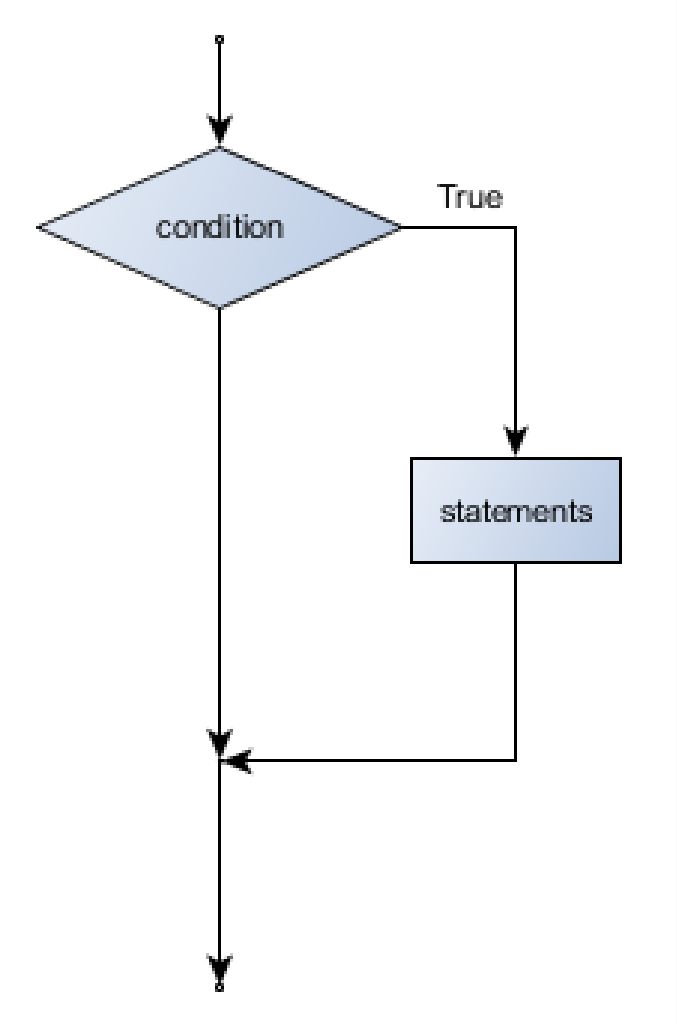
- O `elif` substitui o `else...if`
- Apenas um nível de indentação

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 negada

```
>>> 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:

$$\texttt{not (A == B)} \iff \texttt{A != B}$$
$$\texttt{not (A < B)} \iff \texttt{A >= B}$$
$$\texttt{not (A <= B)} \iff \texttt{A > B}$$
$$\vdots$$
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

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*.

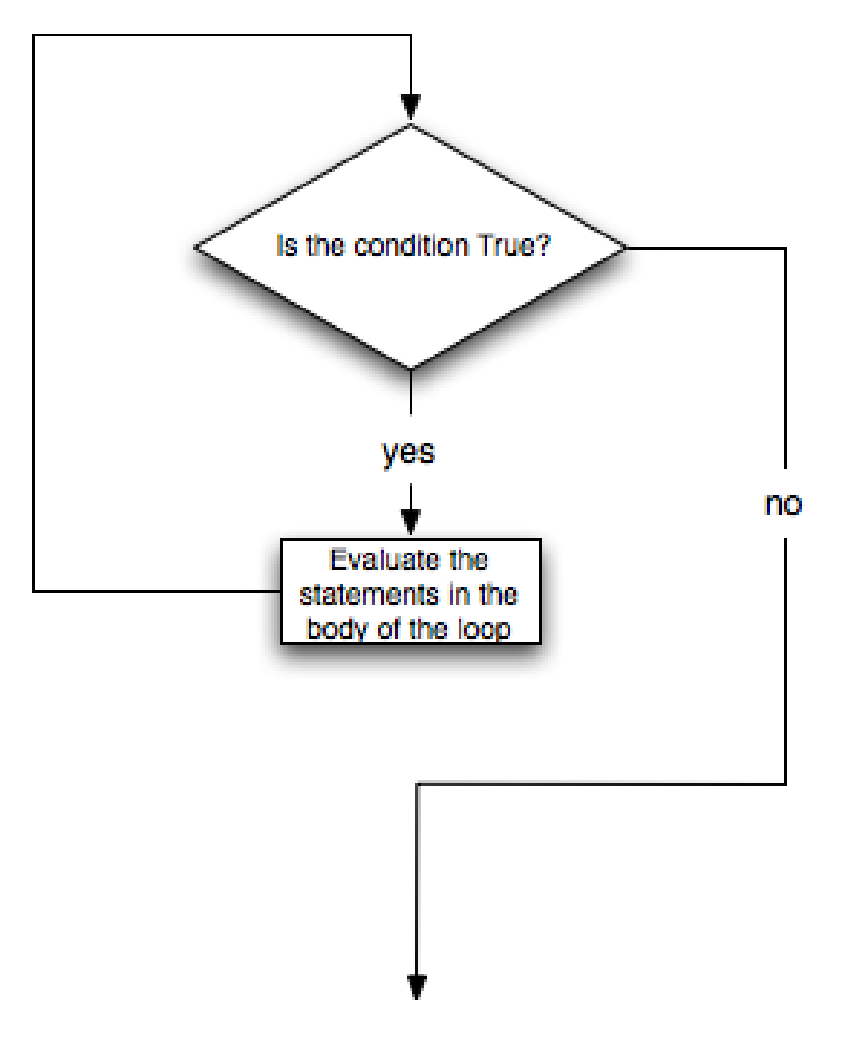
Ciclos *while* (cont.)

```
while condição:
      ⋮
      instruções a repetir
      ⋮
```

- Repete as instruções enquanto a condição for verdadeira
- A condição é re-avaliada após cada iteração
- O corpo do ciclo deve modificar alguma variável da condição

Ciclos while (cont.)

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



Exemplo

```
def crescente(n):
    i = 1
    while i<n:
        # repete estas instruções
        print(i)
        i = i+1
    # após o fim do ciclo
    print('fim')
```

Exemplo (cont.)

```
>>> crescente(10)
1
2
3
4
5
6
7
8
9
fim
```

Outro 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
```

Modificar variáveis

Muitas vezes necessitamos de incrementar ou decrementar variáveis.

```
⋮
i = i+1
⋮
```

Esta atribuição pode ser abreviada:

```
⋮
i += 1
⋮
```

Modificar variáveis (cont.)

Mais geralmente, temos as seguintes atribuições abreviadas:

```
v  +=  k   ⟶   v = v + k
v  -=  k   ⟶   v = v - k
v  *=  k   ⟶   v = v * k
v  /=  k   ⟶   v = v / k
v  //=  k  ⟶   v = v // k
v  **=  k  ⟶   v = v ** k
```

Um ciclo while pode não terminar

```
def repete():
  n = 1
  while True:  # repete indefinidamente
     print('Se',n,'elefantes incomodam muita gente')
     print(n+1,'elefantes incomodam muito mais!')
     n += 1
```

Sequência de Collatz

Nem sempre é fácil concluir que um ciclo while não termina.
Exemplo:

Calcular a Sequência de Collatz para um dado $n > 0$
Enquanto n não for igual a 1, gerar o próximo elemento da sequência usando a função:

$$f(n) = \begin{cases} n/2 & \text{se } n \text{ é par} \\ 3n+1 & \text{se } n \text{ é ímpar} \end{cases}$$

Sequência de Collatz (cont.)

```
def sequencia(n):
    print(n)
    while n != 1:
        if n%2 == 0:   # n é par
            n = n//2
        else:          # n é ímpar
            n = 3*n+1
        print(n)
```

Sequência de Collatz (cont.)

- `sequencia(1)`: 1
- `sequencia(2)`: 2, 1
- `sequencia(3)`: 3, 10, 5, 8, 4, 2, 1
- `sequencia(4)`: 4, 2, 1
- `sequencia(5)`: 5, 8, 4, 2, 1
- `sequencia(6)`: 6, 3, 10, 5, 8, 4, 2, 1
- ...
- `sequencia(27)`: 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, ... $\rightarrow$ são necessárias 111 iterações para chegar a 1
- ...

Não sabemos se esta função termina para todo $n > 0$.

`http://en.wikipedia.org/wiki/Collatz_conjecture`

Fatorial iterativo

Calcular o fatorial de n

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n$$

1. Inicialmente: $p = 1$
2. Repetir para i de 2 até n:
   $$p \leftarrow p \times i$$
3. No fim do ciclo o valor de p é $n!$

Fatorial usando um ciclo for

```
def factorial(n):
    # p acumula o produto 1*2*...*n
    p = 1
    # repetir para i de 2 até n
    for i in range(2, n+1):
        p *= i
    # resultado é o valor de p
    return p
```

Fatorial usando um ciclo while

```
def factorial(n):
    # p acumula o produto 1*2*...*n
    p = 1
    # repetir para i de 2 até n
    i = 2
    while i<=n:
      p *= i
      i += 1
    # resultado é o valor de p
    return p
```

Fatorial usando um ciclo while (cont.)

Mais geralmente:

- Podemos *sempre* re-escrever um ciclo for num ciclo while.
- Nem sempre podemos re-escrever um ciclo while como um ciclo for (exemplo: a função que gera a sequência de Collatz).

Saída e continuação num ciclo

Saída e continuação num ciclo

Duas instruções permitem alterar a execução de um ciclo:

break sair a meio do ciclo

continue passar à próxima iteração

Break com um ciclo for

```
for i in [12, 16, 17, 24, 29]:
    if i % 2 == 1: # se é ímpar
        break      #  ... termina o ciclo
    print(i)
print("fim")
```

Resultado:

```
12
16
fim
```

Continue com um ciclo for

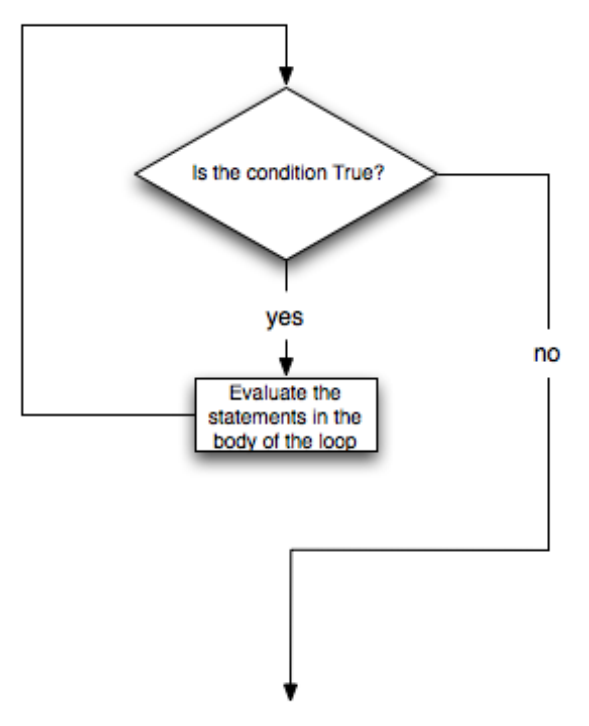
```
for i in [12, 16, 17, 24, 29, 30]:
    if i % 2 == 1:      # se é impar
        continue        # ...passa ao próximo
    print(i)
print("fim")
```

Resultado:

```
12
16
24
30
fim
```

Saída de um ciclo while

O teste do ciclo *while* ocorre antes da execução do corpo.



Podemos usar *break* para colocar um teste no corpo do ciclo.

Exemplo: teste no meio do corpo

```
total = 0
while True:
    resposta = input("Insira um número (ou vazio)")
    if resposta == ' ':
        break           # termina o ciclo
    total += int(resposta)
print("Total = ", total)
```

Sumário

while repetir enquanto a condição é verdadeira

for repetir para uma sequência de valores

Que tipo de ciclo usar?

Ciclo for
1. iterar sobre sequências aritméticas;
2. iterar sobre listas ou tuplos.

Ciclo while nos outros casos.