Programação e Base de Dados /Computação para Deteção Remota

Introdução à linguagem Python

Sérgio Crisóstomo,
(adaptado dos slides de Rita Ribeiro,
Pedro Vasconcelos e João Pedro Pedroso)

2022/2023

Departamento de Ciência de Computadores

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Conteúdo

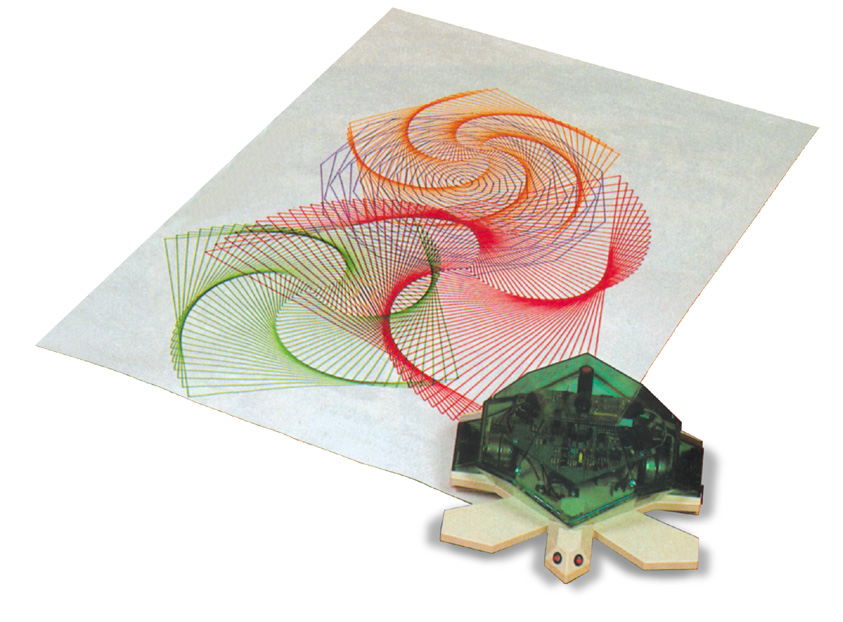
1. Módulo turtle
2. Ciclos for
3. Execução condicional
4. Ciclos while

Módulo turtle

Módulo turtle

Vamos fazer programas que desenham usando o módulo *turtle*:

- o programa controla um *robot* virtual (tartaruga);
- desloca-se para frente, para trás e roda sobre si próprio;
- usa uma caneta para deixar um rasto;
- muito simples, mas permite fazer desenhos impressionantes.



Primeiros passos

Devemos começar por importar o módulo:

```
>>> import turtle
```

Os comandos têm a forma `turtle.`*`comando(...)`*:

```
>>> turtle.clear()          # limpar a janela
>>> turtle.forward(100)     # avançar 100 pixels
>>> turtle.left(90)         # rodar 90 graus a esquerda
>>> turtle.forward(200)     # avançar 200 pixels
```

Primeiros passos (cont.)

Em alternativa, podemos usar

```
>>> from turtle import *
```

e omitir o nome do módulo:

```
>>> clear()
>>> forward(100)
>>> left(90)
>>> forward(200)
```

Comando principais

`forward(`*n*`)`	avançar *n pixels*
`backward(`*n*`)`	retroceder *n pixels*
`left(`α`)`	rodar α graus à esquerda
`right(`α`)`	rodar α graus à direita
`color(`*c*`)`	mudar a cor do traço
`pensize(`*n*`)`	mudar a largura do traço
`penup()`	levantar a caneta
`pendown()`	baixar a caneta
`speed(`*n*`)`	mudar a velocidade da tartaruga
`clear()`	limpar a janela
`reset()`	limpar a janela e re-inicializar a tartaruga

Desenhar um quadrado

Vamos definir uma função sem argumentos para desenhar um quadrado com 100 pixels de lado.

- Desenhar quatro lados, rodando 90° para a esquerda após cada lado
- Alternativa: também poderiamos rodar para a direita

Desenhar um quadrado (cont.)

```
def quadrado():
    forward(100)     # primeiro lado
    left(90)
    forward(100)     # segundo lado
    left(90)
    forward(100)     # terceiro lado
    left(90)
    forward(100)     # quarto lado
    left(90)         # terminar na orientação original
```

Evitando repetições

- Repetimos quatro vezes as instruções:

```
forward(100)
left(90)
```

- Podemos evitar a repetição usando um ciclo *for*

Evitando repetições (cont.)

```
def quadrado():
    for lado in [1,2,3,4]:    # repetir 4 vezes
        forward(100)
        left(90)
```

- A variável *lado* não é usada dentro do ciclo
- Poderiamos usar qualquer outra lista de quatro valores

Alternativas

```
def quadrado():
    for i in [1,2,3,4]:
        forward(100)
        left(90)

def quadrado():
    for i in range(4): # 0, 1, 2, 3
        forward(100)
        left(90)

def quadrado():
    for c in ['red','green','blue','black']:
        color(c)
        forward(100)
        left(90)
```

Generalizando

Vamos agora generalizar a função para desenhar um quadrado de qualquer lado.

Basta tomar a medida do lado como um parâmetro da função.

Generalizando (cont.)

```
def quadrado(lado):
    'Desenhar um quadrado dado o comprimento do lado.'
    for i in range(4):
        forward(lado)
        left(90)
```

Exemplo final

espiral.py

```
from turtle import *

def quadrado(lado):
    for c in ['red','blue','green','black']:
        color(c)
        forward(lado)
        left(90)

reset()
speed(10)
for i in range(36):
    quadrado(50+i*5)
    left(10)
```

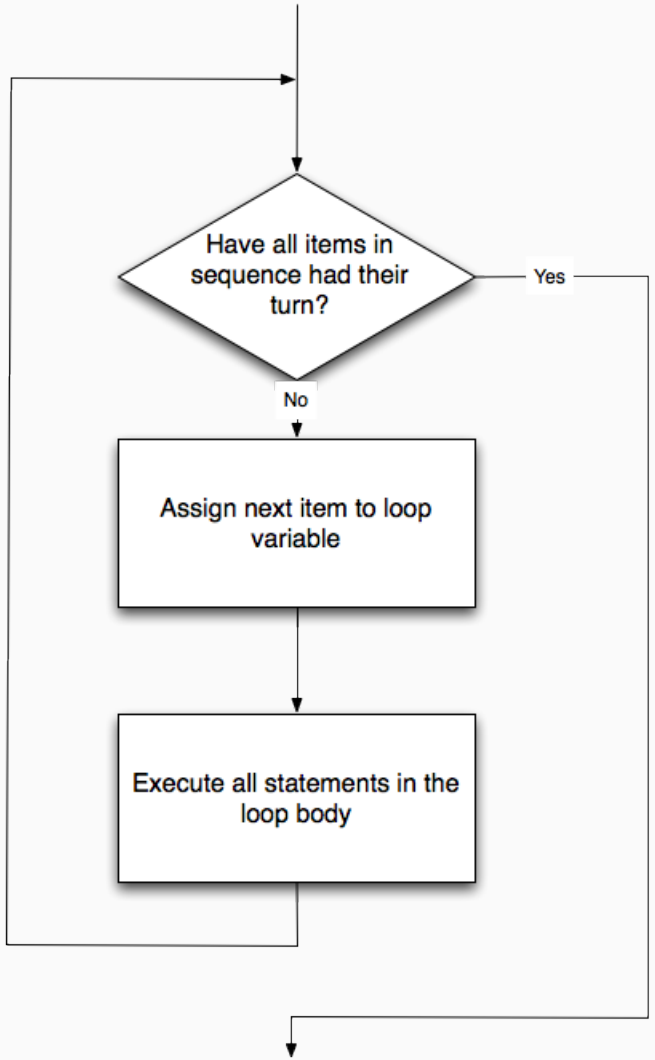
Ciclos for

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:
   - 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)
```

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
```

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

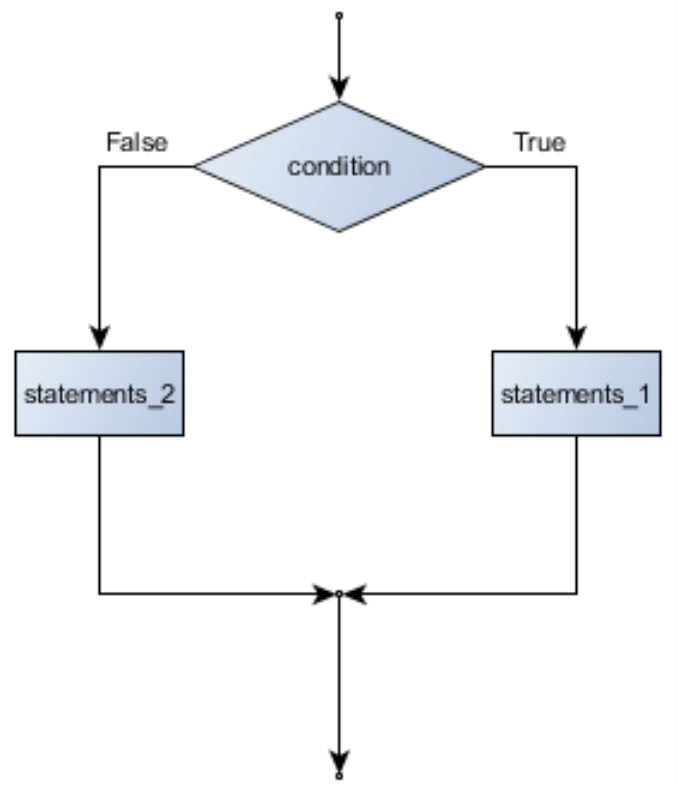
Resultado

```
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

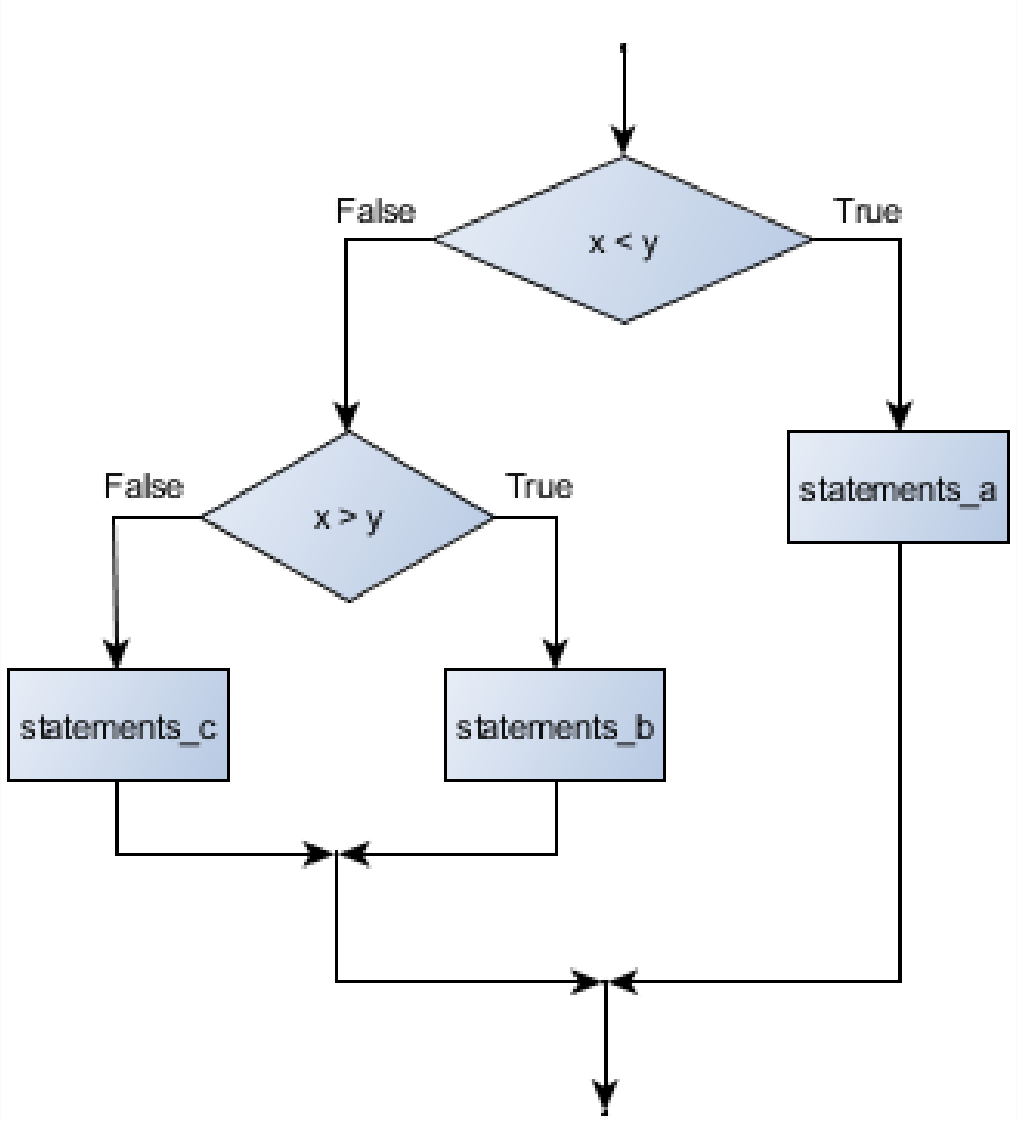
```
0 é par
1 é impar
2 é par
3 é impar
4 é par
```

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.)

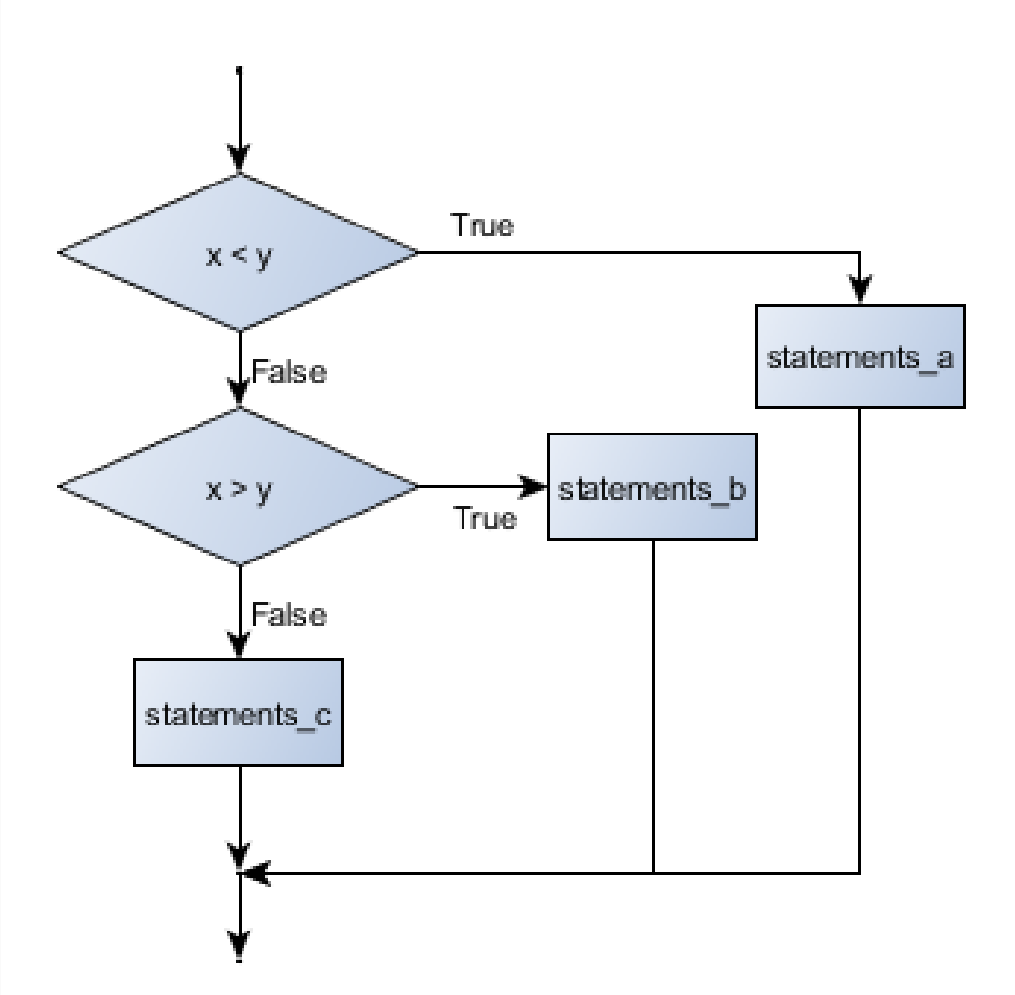


If-else encadeados

```
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")
```

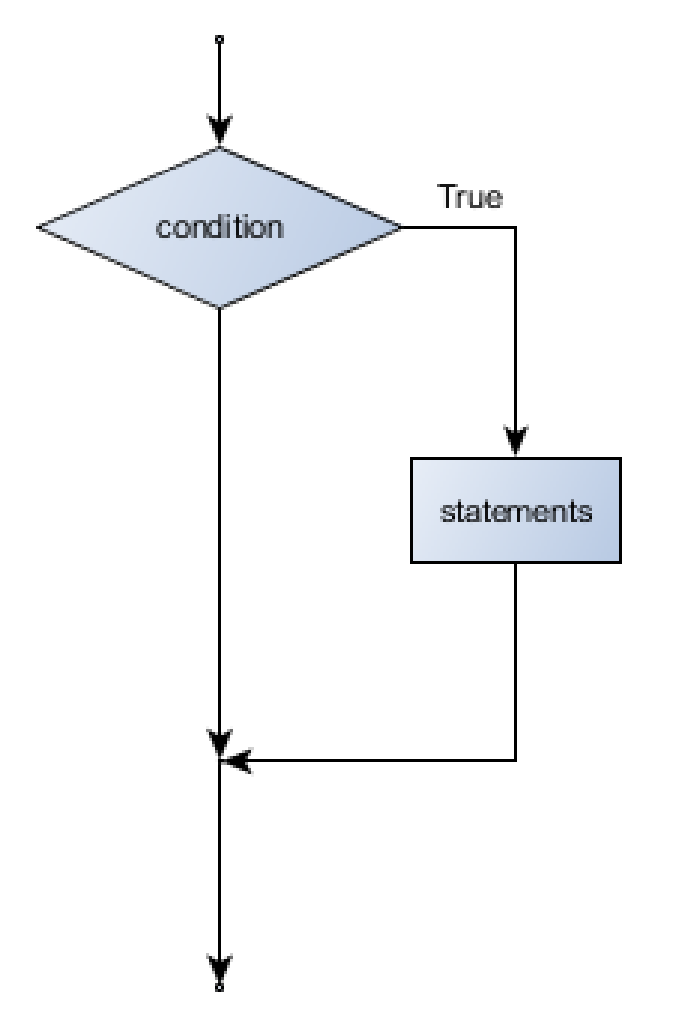
- O `elif` substitui o `else...if`
- Apenas um nível de indentação

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 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:

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

$$\texttt{not (P and Q)} \iff \texttt{(not P) or (not Q)}$$
$$\texttt{not (P or Q)} \iff \texttt{(not P) and (not Q)}$$
$$\texttt{not (not P)} \iff \texttt{P}$$

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*

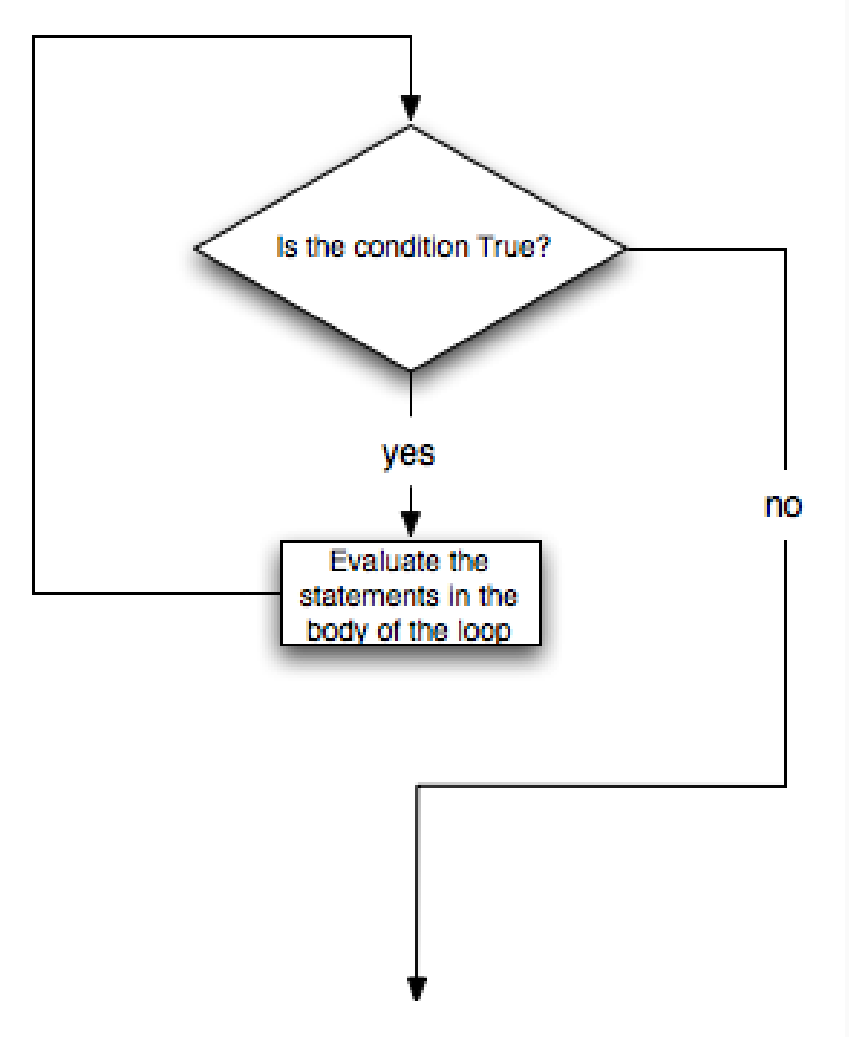
No ciclo *for* especificamos a lista de valores a percorrer.

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:
    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
```