Bioinformatics

Python Programming
Introduction to the Python Language

Sérgio Crisóstomo

2026

Departamento de Ciência de Computadores

Contents

1. For loops

2. Conditional execution

3. While loops

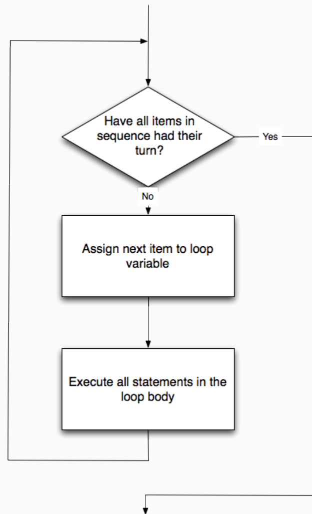
For loops

For loops

```
for variable in list of values:
    instruction 1
    instruction 2
    ⋮
    instruction n
rest of the program
```

1. The loop body is indented.
2. As long as all values are not exhausted:
   - the variable takes the next value in the list;
   - we execute the loop body.
3. After the last value: the execution continues in the rest of the program.

Execution flow of a *for* loop



Examples

```
friends = ["Ana","João","Pedro","Beatriz"]
for name in friends:
    msg = "Hello, " + name + "!"
    print(msg)
```

```
Hello, Ana!
Hello, João!
Hello, Pedro!
Hello, Beatriz!
```

Examples (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
```

Function `range`

Often we want to perform a loop over numeric values in an arithmetic progression (example: $0, 1, 2, 3, \ldots$)

The *range* function allows us to easily generate values this way.

Function `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)
```

Function `range` (cont.)

range(n) integer values from 0 to $n - 1$ inclusive;

range(i,n) integer values from i to $n - 1$ inclusive;

range(i,n,d) integer values $i,\ i + d, i + 2d, \ldots$ less than n.

Note that `range(n)` includes zero but does not include n.

Programmers prefer to count from zero!

Example

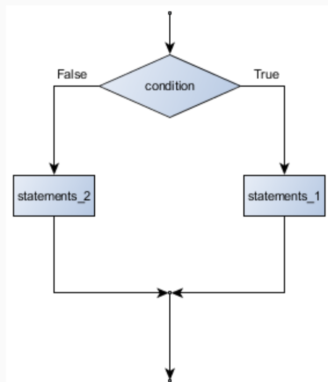
```
import math
for x in range(10,110,10):
    print(x, math.log10(x))
```

```
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
```

Conditional execution

Conditional execution

```
if condition:
    instructions 1
else:
    instructions 2
```



- The expression in the `if` line is the condition
- The block after the `if` is executed if the condition is true
- The block after the `else` is executed if the condition is false

Conditions

`==`	equal
`!=`	not equal
`>`	greater than
`<`	less than
`>=`	greater than or equal to
`<=`	less than or equal to

The result is a logical value (`True` or `False`).

Conditions (cont.)

Examples:

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

Example

```
for x in range(5):
    if x%2 == 0:
       print(x, "is even")
    else:
       print(x, "is odd")
```

Output

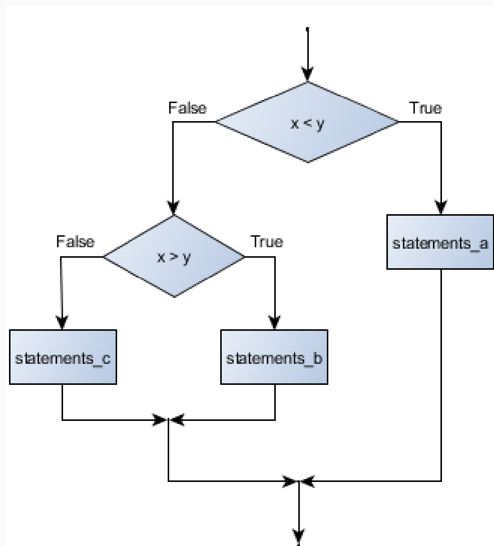
```
0 is even
1 is odd
2 is even
3 is odd
4 is even
```

If-else inside an *if-else*

```
if x < y:
    print(x, "is less than", y)
else:
    if x > y:
        print(x, "is greater than", y)
    else:
        print(x, "and", y, "are equal")
```

- The indentation indicates the structure of the conditions
- It can be hard to read with more than two levels

If-else inside an *if-else* (cont.)

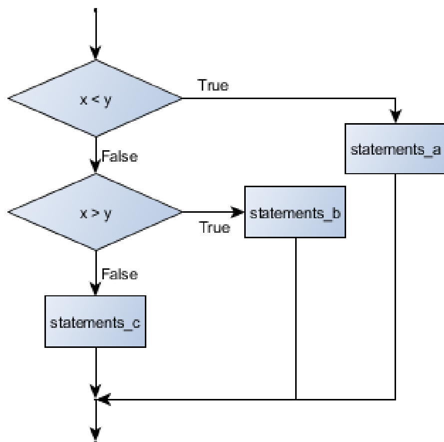


Chained *if-else*

```
if x < y:
    print(x, "is less than", y)
elif x > y:
    print(x, "is greater than", y)
else:
    print(x, "and", y, "are equal")
```

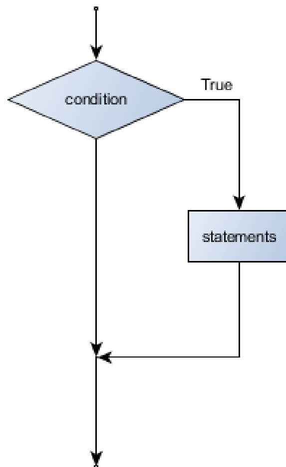
- The `elif` replaces `else...if`
- Only one level of indentation

Chained *if-else* (cont.)



Omitting the `else` block

```
if x < 0:
    print(x, "is negative")
```



Logical connectives

`and`	both conditions are true
`or`	at least one condition is true
`not`	the condition is false

```
>>> import math

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

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

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

Simplifying conditions

Example:

```
if not (age>=18):
    print("Not old enough to drive!")
```

is equivalent to

```
if age<18:
    print("Not old enough to drive!")
```

Simplifying conditions (cont.)

More generally, we can simplify negations using equivalences:

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

$$\vdots$$

etc.

Simplifying conditions (cont.)

We can simplify negations of logical connectives.

```
if not (energy>=0.90 and shield>=100):
    print("The attack has no effect.")
else:
    print("The dragon dies!")
```

is equivalent to

Simplifying conditions (cont.)

```
if energy<0.90 or shield<100:
    print("The attack has no effect.")
else:
    print("The dragon dies!")
```

Simplifying conditions (cont.)

More generally:

$$
\begin{aligned}
\texttt{not (P and Q)} &\iff \texttt{(not P) or (not Q)} \\
\texttt{not (P or Q)} &\iff \texttt{(not P) and (not Q)} \\
\texttt{not (not P)} &\iff \texttt{P}
\end{aligned}
$$

Simplifying conditions (cont.)

We can swap the order of *if-else* blocks.

```
if not (energy>=0.90 and shield>=100):
    print("The attack has no effect.")
else:
    print("The dragon dies!")
```

is equivalent to

Simplifying conditions (cont.)

```
if energy>=0.90 and shield>=100:
    print("The dragon dies!")
else:
    print("The attack has no effect.")
```

While loops

While loops

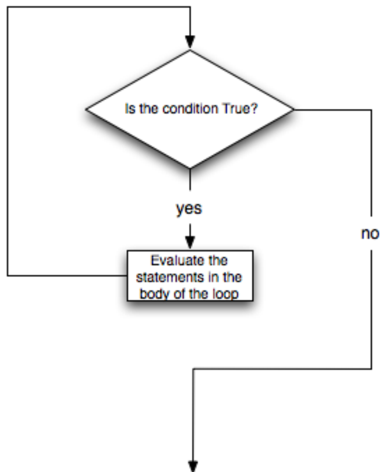
In a *for* loop, we specify the list of values to iterate through.

Sometimes we need to iterate without knowing *a priori* how many times the loop will execute.

For these cases, we can use a *while* loop.

While loops (cont.)

```
while condition:
    instruction 1
    instruction 2
    ⋮
    instruction n
rest of the program
```



Example

Find the first natural number n such that

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

```
n = 0                 # upper bound of the sum
s = 0                 # value of the sum 1+2+...+n
while s <= 1000:      # while the sum does not exceed 1000
    n = n+1           # one more natural number
    s = s+n           # update the sum
print(n)              # print the number found
```