INFO-H-600 - Computing foundations of data sciences

Session 2
Introduction to Python
Functions and control flows

Université libre de Bruxelles
École polytechnique de Bruxelles

2019-2020

Control Flow and Boolean Expressions

Boolean Expressions

A Boolean expression is an expression where the value is either true (`True`) or false (`False`). These expressions are of type `bool`.

```
>>> 5 == 5
True
>>> 5 != 5
False
```

Boolean expressions can be composed of the following relational operators : `<  <=  >  >=  !=  ==`

Do not mix = (assignation) and == (equality comparator).

Simples tests

The instruction `if` allows to evaluate a condition and execute some code only if the condition is *True*.

```python
x = 2
if x >= 0:
  print('x is positive')
```

The executed code is composed of the idented bloc following the `if` instruction.

An idented bloc consists of some code shifted to the right from the same amount.

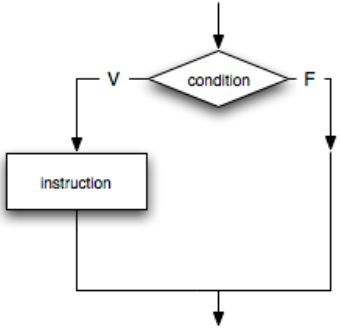
The instruction bloc after the `else` instruction are executed only if the condition is not satisfied.

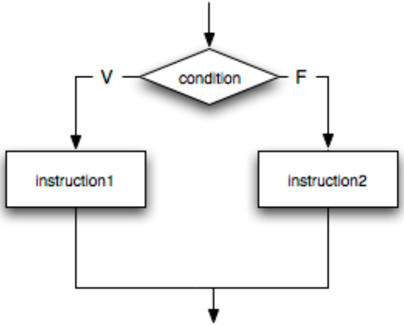
```python
x = 2
if x % 2 == 0:
  print('x is even')
else:
  print('x is odd')
```

Simples tests

```
if condition:
  instruction
```

```
if condition:
  instruction1
else:
  instruction2
```





Chained and nested tests

Tests can be chained using the `elif` (else if) instruction :

```python
if x > y:
        print('x is greater than y')
elif x < y:
        print('x is smaller than y')
else:
        print('x is equal to y')
```

Tests can also be nested thanks to proper indentation :

```python
if x == y:
    print('x is equal to y')
else:
    if x < y:
        print('y is greater than x')
    else:
        print('x is greater than y')
```

Composition of different tests

There are three logical operators : `and`, `or` and `not`.

We built boolean expressions by using these logical operators.

```
>>> x = 5
>>> 0 < x and x < 10
True
>>> x % 2 == 0 or x % 3 == 0
False
>>> not x > 10
True
```

In Python, it is possible to do multiple comparisons at once :

```
>>> x = 5
>>> 0 < x < 10
True
```

Boolean Algebra

- `a and b` is true if and only if `a` is true and `b` is true
- `a or b` is false if and only if `a` is false and `b` is false

a	**b**	**a and b**	**a or b**
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

De Morgan's Law :

- `not(a and b)` is equivalent to `(not a) or (not b)`
- `not(a or b)` is equivalent to `(not a) and (not b)`

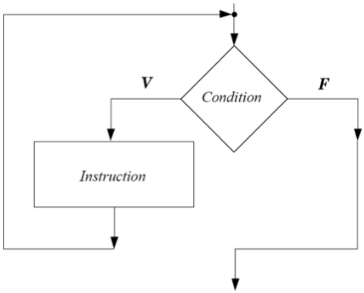
Example of equivalent expressions :

```
not((0 >= x) or (x >= 1000))
0 < x and x < 1000          # more readable
```

`while` loops

The `while` instructions let us have a bloc of instruction which is repeated as long as a condition is verified.

```
while condition:
    instructions
```

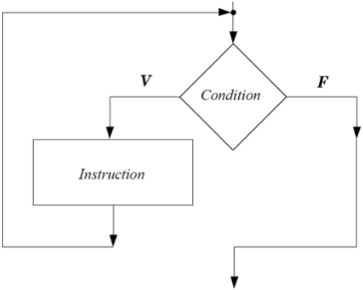


1. The condition is checked
2. If the condition is verified, the instructions are executed and we go back to point 1.

   If the condition is not verified, we “exit” the loop.

while loop : example

```
x = 1

while x < 8:
    print(x)
    x = x + 2

print(x)
```



Functions

Functions

A function is a sequence of instruction that has a name. It can receive as input some arguments and can return some output values.

```
length = len('SPAM')
```

'SPAM' → len → 4

A function can be seen as a blackbox doing some work.

- Its arguments are the information that it needs to perform its work.
- Its returned value is the result of its work.

Composition : the argument of a function can be any compatible expression :

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
x = math.exp(math.log(x+1))
```

Working with functions from existing modules

A module is a library containing some variables, functions and classes.

There are different methods to import a library

```
import math           # imports the whole library
print(math.pi))
```

```
import math as m      # imports the whole library
                      # but renames it
print(m.sqrt(16))
```

```
from math import *    # imports the whole library but
                      # within the same namespace !
print(cos(0.4))
```

```
from math import sin # only sin()
                      # within the same namespace !
print(sin(0.5))
```

Creating your own functions

A function definition specifies the name, the parameters (if any) as well as the sequence of instructions that the function is performing

Each line of the sequence is indented (code bloc)

```
>>> def times(x, y):
      return x * y

>>> def pretty_print(a_string):
      print('*' * (len(a_string) + 4))
      print('* ' + a_string + ' *')
      print('*' * (len(a_string) + 4))

>>> y = times(2, 3)
>>> print(y)
6
>>> pretty_print('Python')
**********
* Python *
**********
```

Return and parameters

The `return` keyword interrupt the function and defines its result :

```
>>> def get_ratio(x, y):
      return float(x) / y
      print('done.')

>>> get_ratio(3,4)
0.75
```

Here, the `print(’done.’)` instruction is never executed.

The order of the arguments is important, not their name.

```
>>> get_ratio(4,3)
1.3333333333333333
```

```
>>> x = 4
>>> y = 3
>>> get_ratio(y, x):
0.75
```

Local variables

The parameters as well as the variables declared inside the function are local variables which means they only exist in concerned function !

```
>>> def pretty_print(a_string):
      size = len(a_string) + 4
      print('*' * size)
      print('* ' + a_string + ' *')
      print('*' * size)

>>> pretty_print('Python')
**********
* Python *
**********
>>> a_string
NameError: name 'a_string' is not defined
>>> size
NameError: name 'size' is not defined
```

We talk about the scope of a variable : the region where it is visible.

Document your functions !

A doctring is a commentary (delimited by `"""`) placed at the beginning of your functions.

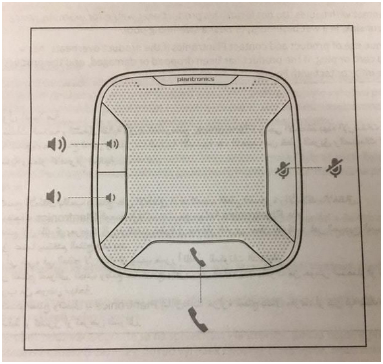
```
>>> def get_sum(x, y):
      """Returns the sum of x and y."""
      return x + y

>>> help(get_sum)
'Help on function get_sum in module __main__:
get_sum(x, y)
  Returns the sum of x and y.
```

Good Habit : clearly document your functions !

(Say what the function does and not how it does it !)

Document your functions!

- However, do not exagerate



```
>>> def get_sum(x, y):
       """Returns the sum of x and y."""
       return x + y
```

List short introduction for today’s exercises

List short introduction for todays exercises

Lists are datastructures that can contain different values.

Initialising lists :

```
>>> a = []          # empty list
>>> print(a)
[]
>>> b = [5, 7, a, 'hello']
>>> print(b)
[5, 7, [], 'hello']
```

List short introduction for todays exercises

Elements can be added to a list thanks to the `append()` function

```
>>> b.append(19)
>>> print(b)
[5, 7, [], 'hello', 19]
```

We can acces elements of a list (starting with 0) :

```
>>> print(b[1])
7
>>> print(b[3])
hello
```

More about lists next time !

`for` loops (simple)

```
>>> for letter in 'Python':
...     print('iterated letter:', letter)

iterated letter: P
iterated letter: y
iterated letter: t
iterated letter: h
iterated letter: o
iterated letter: n
```

```
>>> for i in range(3):
...     print(i)

0
1
2
```

`range(a, b, step=1)` : iterator starting from *a* (included) to *b* (not included) with a defined optional step (default value is one).

`while` and `for` loops

```
for i in range(len(liste)):
        print(liste[i])
```

```
i = 0
while i < len(liste):
        print(liste[i])
        i += 1
```

`for` iterate on the whole sequence, while the `while` loops allows to stop the iteration when a condition is no more satisfied.