

# Python : Introduction

A. Malek TOUMI UV1.1

# Introduction : Langage évolué, compilation, interprétation

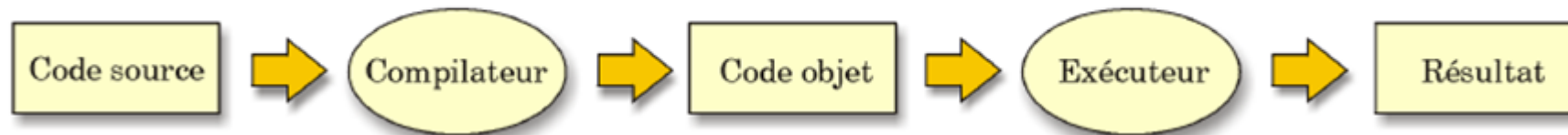
- un langage évolué permet de représenter
  - les nombres par leur notation usuelle
  - les instructions par des opérateurs et des mots-clés expressifs et faciles à lire
- avant exécution, instructions et données doivent être converties en binaire
- interprétation (« code source » est le programme que vous avez écrit) :



*L'interpréteur lit  
le code source ...*

*... et le résultat  
apparaît sur l'écran.*

- compilation :



*Le compilateur lit  
le code source ...*

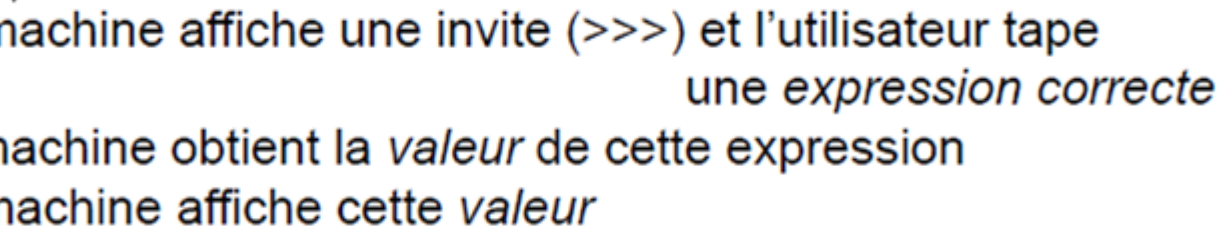
*... et produit un  
code objet (binaire).*

*On exécute  
le code objet ...*

*... et le résultat  
apparaît à l'écran.*

# Introduction : Mode immédiat

- la boucle supérieure (*top-level loop*) :

- 
- The diagram shows a box containing three steps of the top-level loop. An arrow points from the first step to the second, and another arrow points from the third step back to the first, forming a cycle.
- *read* : la machine affiche une invite (>>>) et l'utilisateur tape une *expression correcte*
  - *eval* : la machine obtient la *valeur* de cette expression
  - *print* : la machine affiche cette *valeur*

- exemple :

```
>>> 2**8           # 2 à la puissance 8
256
>>>
```

- exception : si la valeur est **None** (convention), rien n'est affichée

```
>>> nbr = 8
>>>
```

- mode immédiat : bon moyen d'approcher le langage et tester ses idées

*un commentaire*

Simplement avec la  
commande  
«*python* » sur une  
console



# Introduction : Données, opérateurs, expressions

- *données* : données initiales, résultats, valeurs intermédiaires

- types primitifs :

- entiers (sur 32 bits)

0                      123                      -50

- entiers longs

340282366920938463463374607431768211456                      1L

- nombres non entiers (on dit « flottants »)

1.5                      -0.33333333333333331                      0.166054018e-23

- chaînes de caractères

"J'aime Python"

- quelques valeurs conventionnelles

True (*vrai*), False (*faux*), None (*absence de valeur*)

# Introduction : Données, opérateurs, expressions

- plusieurs notations

```
>>> "Bonjour"
'Bonjour'
>>> "Bonjour" == 'Bonjour'
True
>>> "J'ai dit bonjour"
"J'ai dit bonjour"
>>> "J'ai dit \"bonjour\""
'J\'ai dit "bonjour"'
>>> print "J'ai dit \"bonjour\""
J'ai dit "bonjour"
>>> """Ceci est une
... longue chaine"""
'Ceci est une\nlongue chaine'
>>> print """Ceci est une
... longue chaine"""
Ceci est une
longue chaine
>>>
```

## Chaîne de caractères

```
st = "langage python"
st = 'langage python'      # idem
st = 'un guillement "'   # chaîne contenant un guillement
st = "un guillement \""  # chaîne contenant un guillement, il faut ajouter
                           #   pour ne pas confondre avec l'autre guillemet
st = st.upper ()          # mise en lettres majuscules
i = st.find ("PYTHON")     # on cherche "PYTHON" dans st
print (i)                  # affiche 8 Version 3.x, écrire print (i),
                           #   pour la version 2.x, écrire print i
print (st.count ("PYTHON")) # affiche 1 Version 3.x : idem print (...)
print (st.count ("PYTHON", 9)) # affiche 0 Version 3.x : idem print (...)
```

# Introduction : Données, opérateurs, expressions

Variables,  
affectation

- affecter plusieurs variables par la même valeur

```
>>> a = b = c = 0
>>>
```

- affecter plusieurs variables en même temps

```
>>> a, b = 0, 1
>>>
```

- plus fort :

```
>>> a, b = b, a + b
>>>
```

*couple de variables*

*couple de valeurs*

---

# Introduction : Données, opérateurs, expressions

## Opérateurs

- arithmétiques :           +           -           \*           \*\*           /           %  

```
>>> 2 ** 3           # puissance  
8  
>>> 17 % 5           # modulo (reste du quotient)  
2  
>>>
```
- comparaison :           ==           !=           <           <=           >           >=  

```
>>> 2 ** 3 == 8  
True  
>>>
```
- logiques :           and           or           not  

```
>>> x = 14  
>>> x >= 10 and x <= 20  
True  
>>>
```
- quelques bizarreries : + (entre chaînes), % (avec une chaîne), etc.

Opérateurs :  
priorité,  
associativité,  
parenthèses



# Modules et bibliothèques

```
>>> sqrt(4)
[...]  
NameError: name 'sqrt' is not defined
```

*sqrt appartient au module math*

```
>>> math.sqrt(4)
[...]  
NameError: name 'math' is not defined
```

*le module math n'est pas connu*

```
>>> import math  
>>> math.sqrt(4)  
2.0
```

```
>>> sqrt(4)
[...]  
NameError: name 'sqrt' is not defined
```

*importer ne dispense pas  
de préfixer, sauf si...*

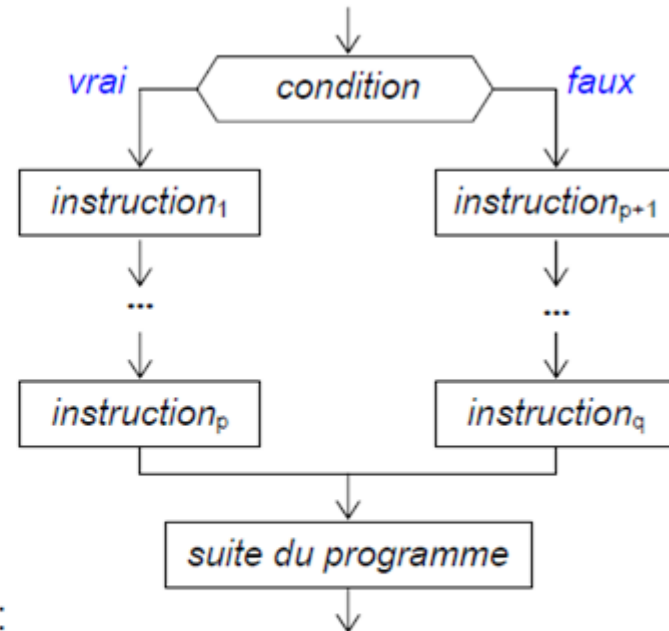
```
>>> from math import *  
>>> sqrt(4)  
2.0  
>>>
```



# Instructions

# Instruction conditionnelle : if, else, elif

```
if condition :  
    instruction1  
    ...  
    instructionp  
else :  
    instructionp+1  
    ...  
    instructionq  
suite du programme
```



- condition est une expression booléenne :  
sa valeur est **True** ou **False**
- l'indexation (marge à gauche) joue un rôle syntaxique
  - elle est obligatoire
  - elle suit des règles strictes

# Instruction conditionnelle : Exemple

- exemple :

```
racine.py
import math
x = float(raw_input("x? "))
if x >= 0:
    y = math.sqrt(x)
    print "La racine de", x, "est", y
else:
    print "On ne peut prendre la racine d'un nombre négatif!"
print "Au revoir"
```

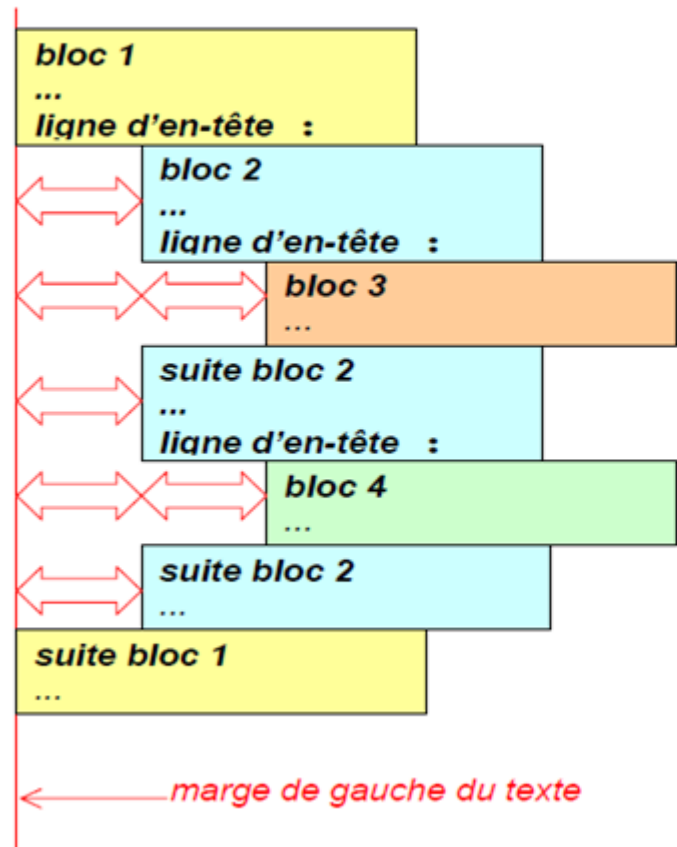
- exécution :

```
$ python racine.py
x? 25
La racine de 25.0 est 5.0
Au revoir
$ python racine.py
x? -5
On ne peut prendre la racine d'un nombre négatif!
Au revoir
$
```



# Indentation

## L'imbrication des blocs et l'indentation



```
if condition :  
    instruction1  
    ...  
    instructionp  
else :  
    instructionp+1  
    ...  
    instructionq  
suite du programme
```

# Instruction répétitive : While

## Ex. : Parcourir une liste

- données séquentielles logées dans la mémoire : listes

```
liste = [ 20, 0, -12, 5.0, 51 ]
```

- problème : effectuer un traitement sur chaque élément d'une liste.  
Exemple : calculer la somme des termes d'une liste *de nombres*

```
somme = 0
```

```
n = len(liste)
```

```
i = 0
```

```
print i, somme
```

```
while i < n :
```

```
    somme = somme + liste[i]
```

```
    i = i + 1
```

```
    print i, somme
```

```
print "somme:", somme
```

*si on est curieux...*

*len = 5*

20	0	-12	5.0	51
0	1	2	3	4

# Instruction répétitive : for

## Ex. : recherche dans une liste, autre manière

- déterminer la présence et le rang d'une valeur dans une liste

```
liste = [ 20, 0, 12, 5.0, 51, 17, 24, 9, 13 ]  
valeur = 12
```

```
rang = -1  
for i in range(len(liste)):  
    if liste[i] == valeur:  
        rang = i  
        break
```

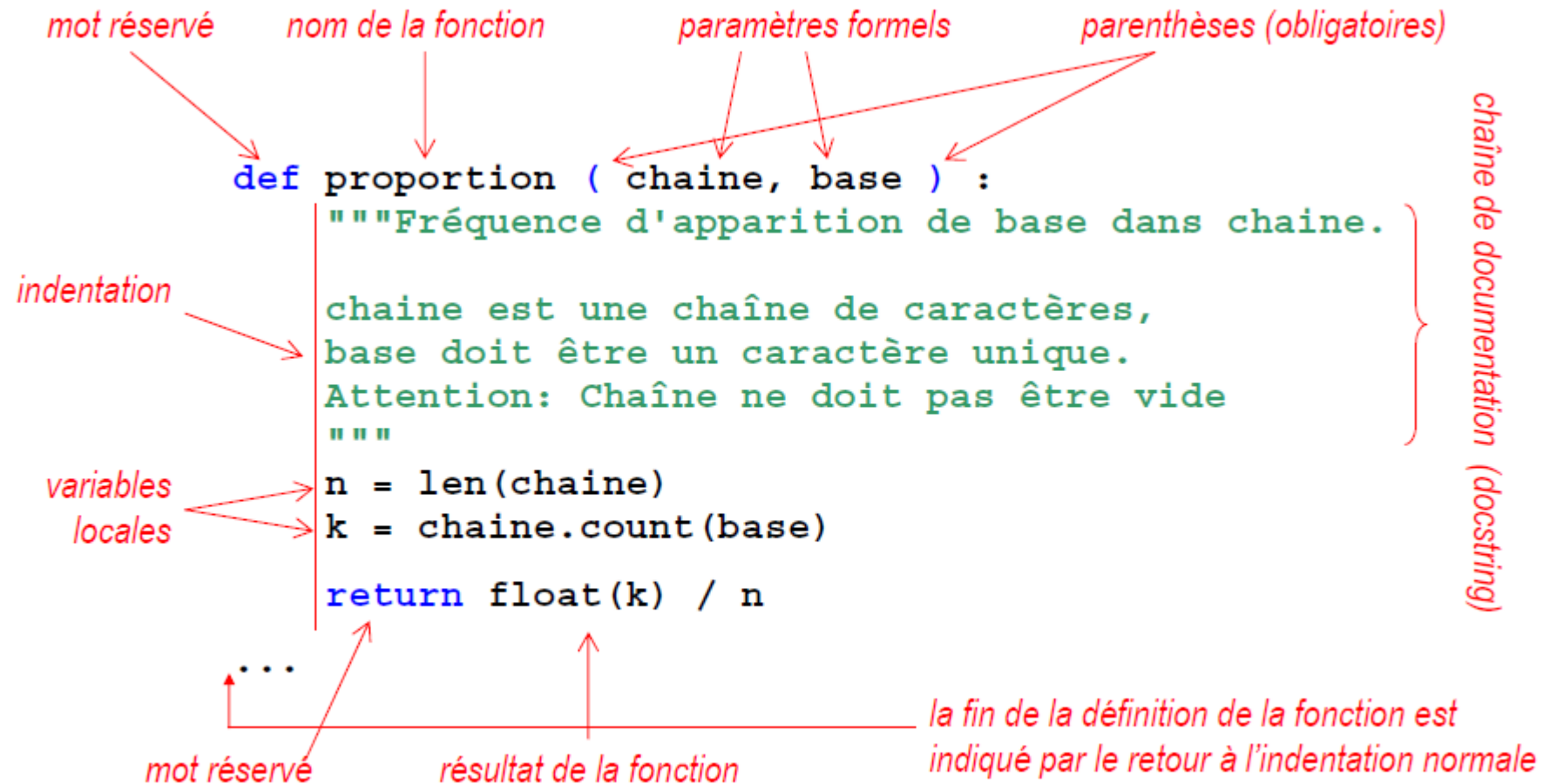
```
if rang >= 0 :  
    print "valeur presente - rang:", rang  
else :  
    print "valeur absente"
```



# Fonction

(ou procédure, méthode)

# Syntaxe



The diagram illustrates the syntax of a Python function definition with the following code and annotations:

```
def proportion ( chaine, base ) :  
    """Fréquence d'apparition de base dans chaine.  
    chaine est une chaîne de caractères,  
    base doit être un caractère unique.  
    Attention: Chaîne ne doit pas être vide  
    """  
    n = len(chaine)  
    k = chaine.count(base)  
    return float(k) / n
```

Annotations and their corresponding parts of the code:

- mot réservé** (reserved word) points to `def`.
- nom de la fonction** (function name) points to `proportion`.
- paramètres formels** (formal parameters) points to `chaine, base`.
- parenthèses (obligatoires)** (mandatory parentheses) points to the parentheses around the parameters.
- chaîne de documentation (docstring)** (documentation string) points to the triple-quoted string.
- indentation** points to the indentation of the function body.
- variables locales** (local variables) points to `n` and `k`.
- mot réservé** (reserved word) points to `return`.
- résultat de la fonction** (function result) points to the expression `float(k) / n`.
- A note at the bottom right states: *la fin de la définition de la fonction est indiquée par le retour à l'indentation normale* (the end of the function definition is indicated by the return to the normal indentation).

# Utilisation

- définition

```
def proportion ( chaine, base ) :  
    n = len(chaine)  
    k = chaine.count(base)  
    return float(k) / n
```

*paramètres ou paramètres formels*

- appel

```
...  
y = proportion ( adn, "t" )  
...
```

*arguments ou paramètres effectifs*

```
n = len(adn)  
k = chaine.count("t")  
return float(k) / n
```

*précisément, comment chaine et base  
sont-ils substitués par adn et "t" ?*



# Fonction sans résultats (procédure)

- fonction « pure » : renvoie un résultat et n'a aucun effet secondaire
- procédure : ne renvoie pas de résultat, le seul intérêt est l'effet secondaire
- procédure en Python :
  - pas d'instruction `return`, ou de la forme « `return` » (sans expression)
  - la fonction renvoie alors la valeur conventionnelle `None`

```
def lister(uneListe) :  
    """Affichage 'vertical' de la liste donnée"""  
    i = 0  
    for x in uneListe :  
        print i, ":", x  
        i = i + 1
```

- le programmeur n'a pas indiqué de résultat  $\Rightarrow$  la fonction renvoie `None`.  
Emploi erroné :

```
n = lister([11, 22, 33, 44, 55])      # étourderie !
```

la plupart des utilisations de la valeur de `n` provoqueront une erreur

Suite ... UV 1.1