

# Programmation sous Python



Première partie

# La programmation

## Démarche

- Penser, réfléchir, ...
- Résoudre un problème à la manière d'un analyste.
- Utiliser un langage formel pour décrire notre raisonnement

*La programmation consiste à « **expliquer** » en détails à une machine ce qu'elle doit faire, en sachant qu'elle **ne peut pas véritablement** « **comprendre** » un langage humain, mais seulement effectuer un **traitement automatique** sur des séquences de caractères.*

*Un programme n'est rien d'autre qu'une **suite d'instructions**, codées en respectant de manière stricte un ensemble de conventions fixées à l'avance que l'on appelle un **langage informatique**. La machine est ainsi pourvue d'un mécanisme qui décode ces instructions en **associant à chaque** « **mot** » du langage une **action précise**.*

*Apprendre à programmer :*

- 1. activité intéressante qui contribue à développer votre « intelligence ».*
- 2. procure une grande satisfaction : pouvoir **réaliser des programmes concrets**.*

# Machine, Langage et Programmation

- Dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires.

**...00110011001100110011000011111100001111....**

*Alors que nous raisonnons en base 10 (grâce à nos 10 doigts)*

*Imaginez qu'un ordinateur ne sache compter qu'avec une paire de doigts*



- Toute information d'un autre type doit être convertie, ou codée, en format binaire. Cela est vrai pour :

- les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.),
- les programmes : c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

# Machine, Langage et Programmation

**Le seul «langage» que l'ordinateur puisse véritablement «comprendre» est donc très éloigné de ce que nous utilisons nous-mêmes.**

- C'est une longue suite de 1 et de 0 (les "bits") souvent traités par groupes de 8 (les « octets »), 16, 32, ou même 64.

**00011111, 1E , AB6D, ...etc**

- Ce « langage machine » est évidemment « presque » incompréhensible pour nous.
- Pour « parler » à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous.



# Machine, Langage et Programmation

Le système de traduction proprement dit s'appellera **interpréteur** ou bien **compilateur**, suivant la méthode utilisée pour effectuer la traduction.

*On appellera langage de programmation un ensemble de « **mots-clés** » (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des « **phrases** » que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire)*

- Langage de haut niveau ?
- Langage de bas niveau ?

# Langage machine, langage de programmation

*Le langage que vous allez découvrir en premier est **Python***

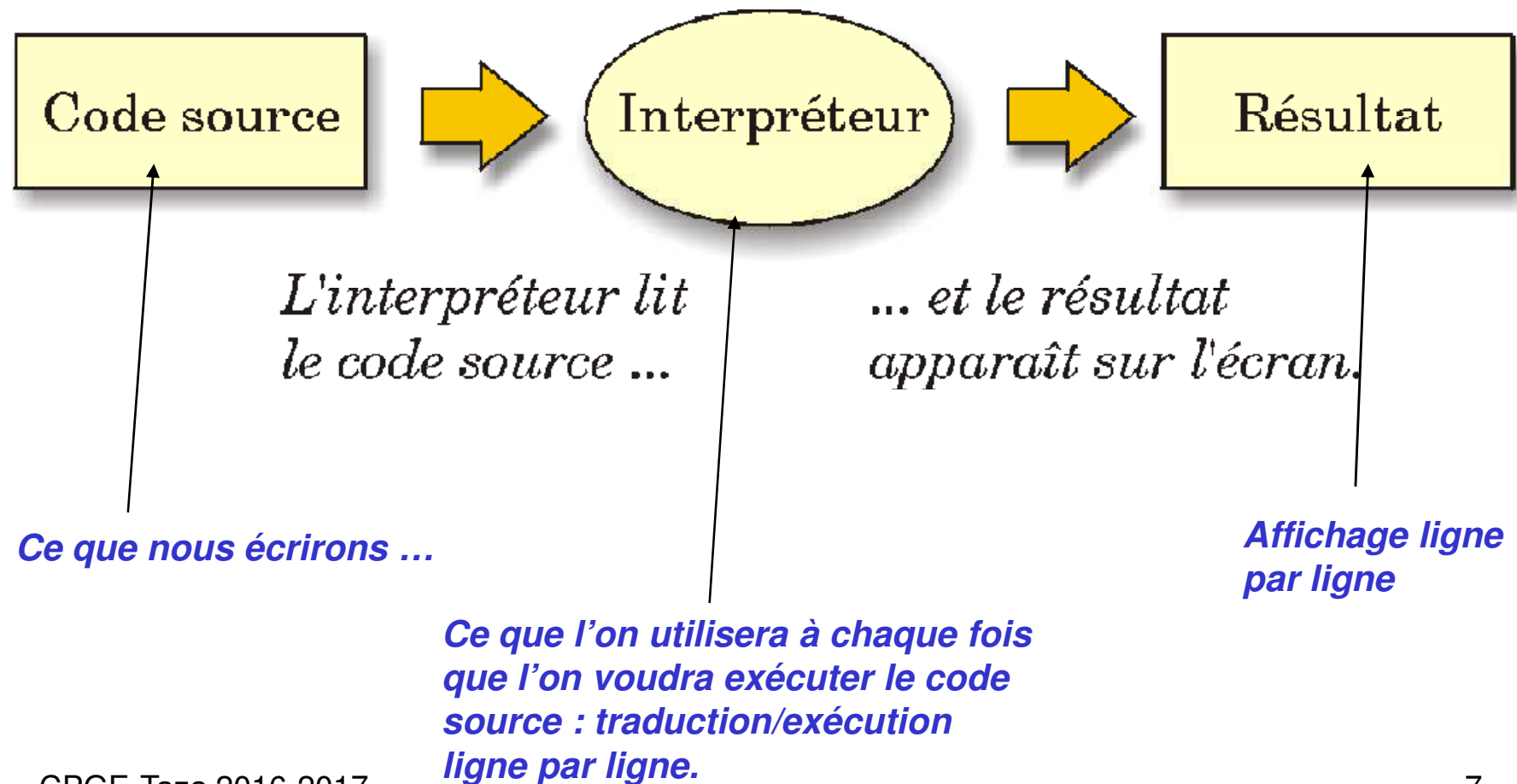
**Pourquoi**



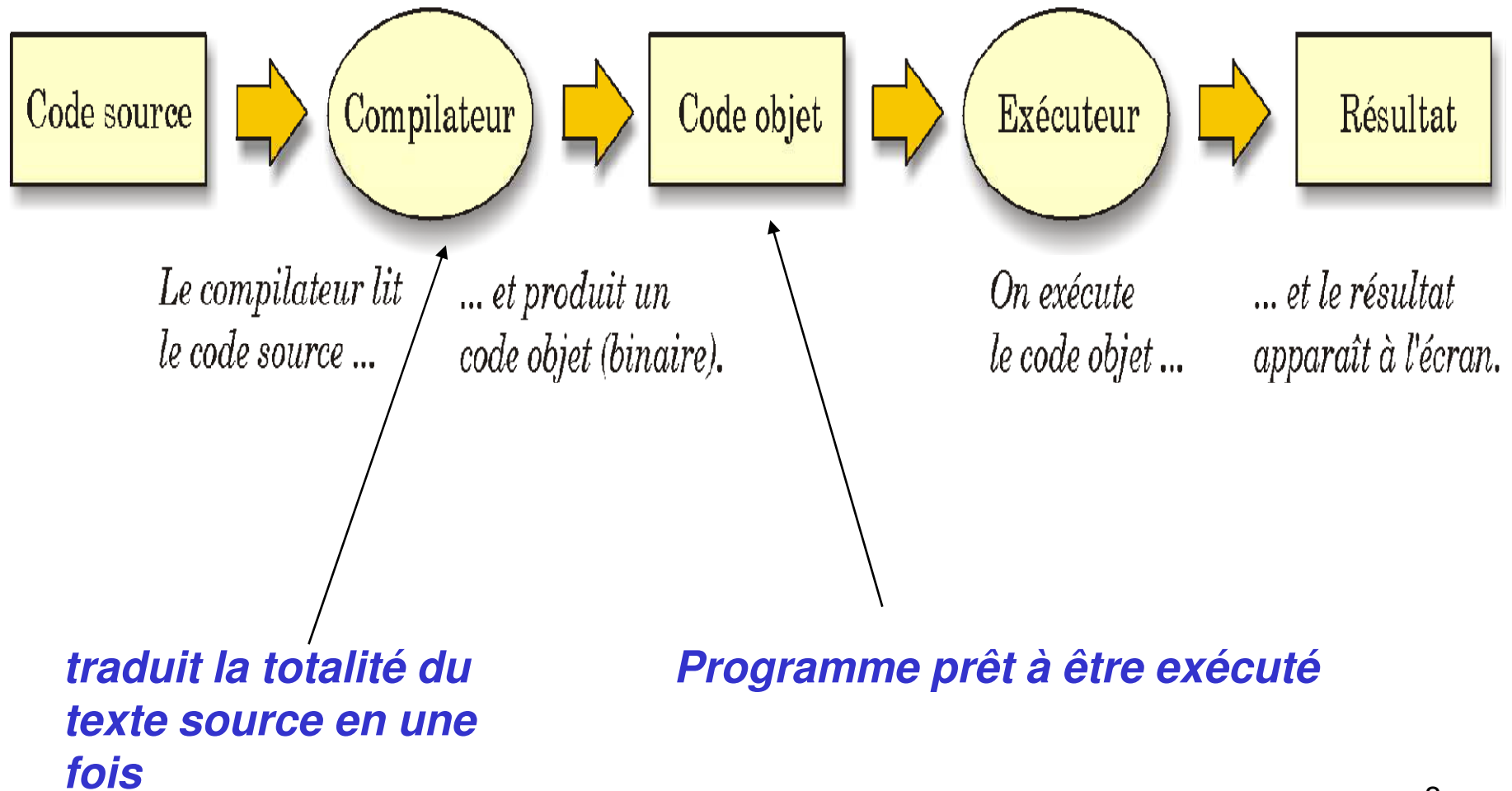
- Il s'agit d'un langage de haut niveau
- Il est beaucoup plus facile d'écrire un programme dans un langage de haut niveau : l'écriture du programme prend donc beaucoup moins de temps
- le programme sera souvent portable : on peut le faire fonctionner sans aucune modification, sur des machines ou des systèmes d'exploitation différents .

# Compilation et Interprétation

Il existe deux techniques de traduction



# Compilation et interprétation





# Compilation et interprétation

## Chacune de ces deux techniques a ses avantages et ses inconvénients

- L'interprétation est idéale lorsque l'on est en phase d'apprentissage du langage. Avec cette technique, on peut tester immédiatement toute modification apportée au programme source, sans passer par une phase de compilation qui demande toujours plus de temps.
- Par contre, lorsqu'un projet comporte des fonctionnalités complexes qui doivent s'exécuter rapidement, la compilation est préférable : il est clair qu'un programme compilé fonctionnera toujours nettement plus vite que son homologue interprété, puisque dans cette technique l'ordinateur n'a plus à (re)traduire chaque instruction en code binaire avant qu'elle puisse être exécutée.

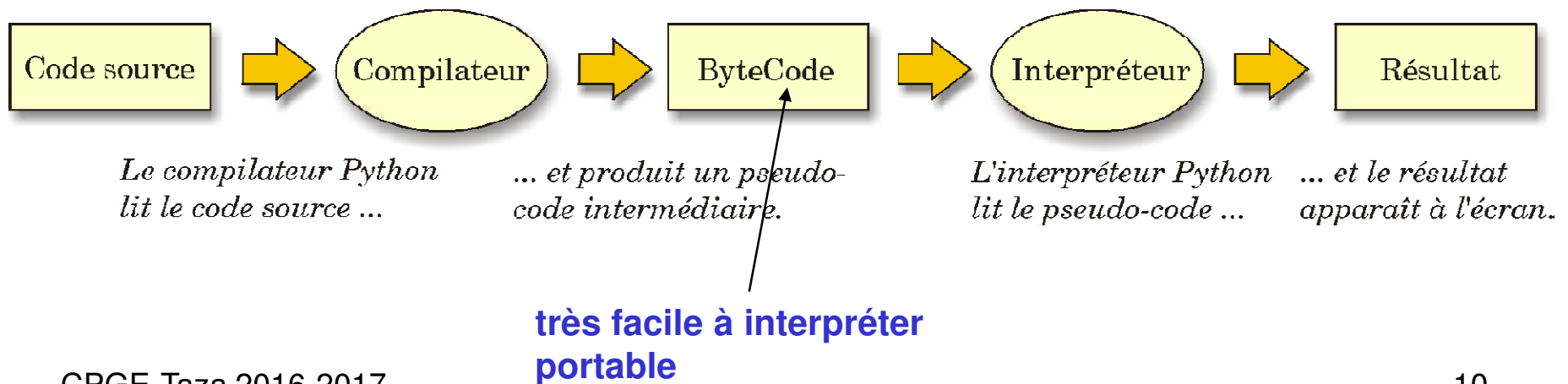
# Compilation et interprétation

Certains langages modernes tentent de **combiner les deux techniques** afin de garder le meilleur de chacune.

- Python



- JAVA



# Compilation et interprétation

## Remarque

Tout ceci peut paraître un peu compliqué, mais la bonne nouvelle est que **tout ceci est pris en charge automatiquement par l'environnement** de développement de Python.

Il vous suffira d'entrer vos commandes au clavier, de frapper <Enter>, et Python se chargera de les compiler et de les interpréter pour vous.

## *Mise au point d'un programme. Recherche des erreurs «debug»*

### Trois types d'erreurs :

1. **syntaxe** : se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.
2. **sémantique** : c'est une erreur de logique, c'est à dire, le programme est sans erreurs mais les résultats sont inattendus.
3. **d'exécution ou « Run-time error »** : lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de lire un fichier qui n'existe plus, une division par zéro).

# Langages naturels et langages formels

1. Les **langages naturels** sont ceux que les êtres humains utilisent pour communiquer. Ces langages n'ont pas été mis au point délibérément (encore que certaines instances tâchent d'y mettre un peu d'ordre) : ils évoluent naturellement.
2. Les **langages formels** sont des langages développés par nous même, en vue d'applications spécifiques.
  - Par exemple, le système de notation utilisé par les mathématiciens est un langage formel particulièrement efficace pour représenter les relations entre nombres et grandeurs diverses.
  - Les chimistes utilisent un langage formel pour représenter la structure des molécules
  - , ...etc.

**Les langages de programmation sont des langages formels qui ont été développés pour décrire des algorithmes et des structures de données.**

# Langages naturels et langages formels

Par comparaison au système de raisonnement humain ...

Lorsque vous lisez une phrase quelconque, vous devez arriver à vous représenter la structure logique de la phrase (même si vous le faites inconsciemment la plupart du temps).



Par exemple, lorsque vous lisez la phrase « la pièce est tombée », vous comprenez que « la pièce » en est le sujet et « est tombée » le verbe. L'analyse vous permet de comprendre la signification et la logique de la phrase (sa sémantique).

D'une manière analogue, l'interpréteur Python devra analyser la structure de votre programme source pour en extraire la signification.



Les langages naturels et formels ont donc beaucoup de caractéristiques communes (symboles, syntaxe, sémantique), mais ils présentent aussi des différences très importantes : ambiguïté et redondance chez l'un, littéralité chez l'autre.

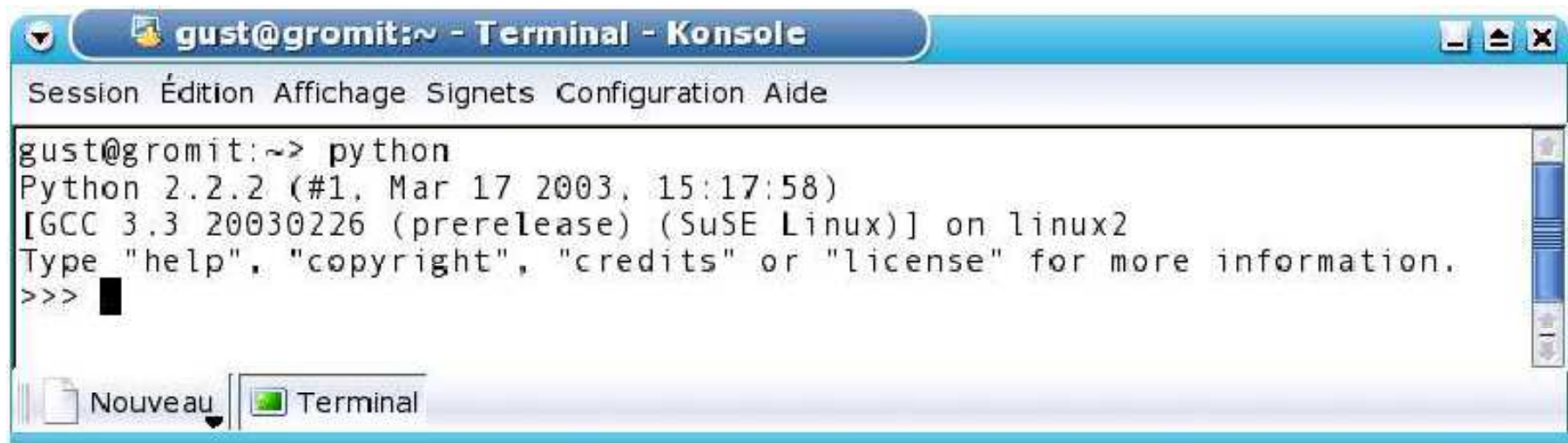
## Premiers PAS !

1. Nous allons demander à l'ordinateur de travailler à notre place, en lui donnant, par exemple, l'ordre d'effectuer une addition et d'afficher le résultat.
2. Pour cela, nous allons devoir lui transmettre des « instructions », et également lui indiquer les « données » auxquelles nous voulons appliquer ces instructions.
3. Nous allons d'abord calculer avec Python en mode interactif, c'est à dire en dialoguant avec lui directement depuis le clavier pour découvrir rapidement un grand nombre de fonctionnalités.

# Premiers PAS !

« Calculer avec Python en mode interactif »

*L'interpréteur peut être lancé directement depuis la ligne de commande (dans un « shell » Linux, ou bien dans une fenêtre DOS sous Windows) : taper la commande **"python"***



```
gust@gromit:~> python
Python 2.2.2 (#1, Mar 17 2003, 15:17:58)
[GCC 3.3 20030226 (prerelease) (SuSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Les trois caractères >>> constituent le **signal d'invite**, ou **prompt principal**, lequel vous indique que Python est prêt à exécuter une commande.



# Calculer avec Python en mode interactif

1. utiliser l'interpréteur comme une simple calculatrice de bureau.
2. tester des commandes, comme :

```
>>> 5+3
```

```
>>> 2 - 9 # les espaces sont optionnels
```

```
>>> 7 + 3 * 4 # la hiérarchie des opérations mathématiques  
# est-elle respectée ?
```

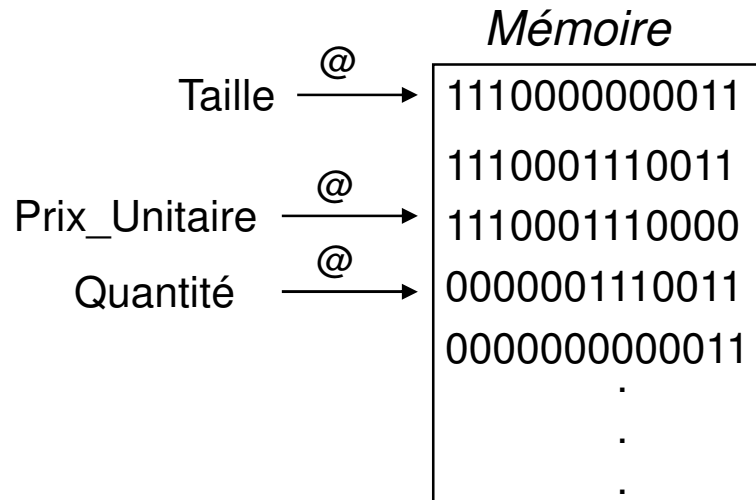
```
>>> (7+3)*4
```

```
>>> 20 / 3
```

**Les parenthèses sont fonctionnelles.**

## Données et variables

1. Un programme d'ordinateur consiste à manipuler des données.
2. Ces données peuvent être très diverses mais dans la mémoire de l'ordinateur elles se ramènent toujours et en définitive à une suite finie de nombres binaires.



*Une variable apparaît dans un langage de programmation sous un nom de variable, mais pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.*

3. Pour pouvoir accéder aux données, le programme d'ordinateur (quel que soit le langage dans lequel il est écrit) fait abondamment usage d'un grand nombre de **variables** de différents types.

# Noms de variables et mots réservés

**Les noms de variables sont des noms (identificateurs) qu'on choisit assez librement**

**De préférence assez courts, mais aussi explicites que possible, pour exprimer clairement ce que la variable est censée contenir :**

***altitude*, *altit* ou *alt* (au lieu de *x*)** pour exprimer une altitude  
***prix\_unit*** pour exprimer un prix unitaire, etc.

## Quelques règles pour les noms de variables sous Python :

1. Un nom de variable (identificateur) est une séquence de lettres (a à z , A à Z) et de chiffres (0 à 9), qui doit toujours commencer par une lettre.
2. Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que \$, #, @, etc. sont interdits, à l'exception du caractère \_ (souligné).
3. La casse est significative (les caractères majuscules et minuscules sont distingués).  
**Attention :** *Somme, somme, SOMME sont donc des variables différentes. Soyez attentifs !*
4. Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans *tableDesMatières*.

# ***Noms de variables et mots réservés***

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme noms de variables les 33 « mots réservés » au langage ci-dessous (version 3.3.0):

<b>and</b>	<b>del</b>	<b>from</b>	<b>None</b>	<b>True</b>	<b>as</b>
<b>elif</b>	<b>global</b>	<b>nonlocal</b>	<b>try</b>	<b>assert</b>	<b>else</b>
<b>if</b>	<b>not</b>	<b>while</b>	<b>break</b>	<b>except</b>	<b>import</b>
<b>or</b>	<b>with</b>	<b>class</b>	<b>False</b>	<b>in</b>	<b>pass</b>
<b>yield</b>	<b>continue</b>	<b>finally</b>	<b>is</b>	<b>raise</b>	<b>def</b>
<b>for</b>	<b>lambda</b>	<b>return</b>			

## Affectation (ou assignation)

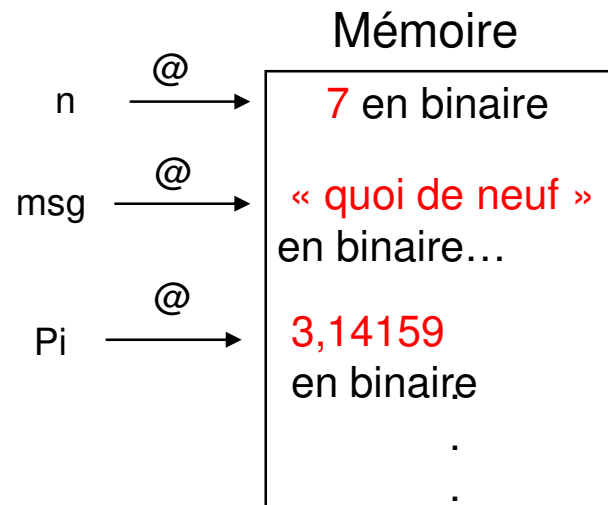
Nous savons désormais comment choisir judicieusement un nom de variable. Voyons à présent comment nous pouvons en définir et *affecter* une valeur.

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe *égal* :

```
>>> n = 7 # donner à n la valeur 7
```

```
>>> msg = "Quoi de neuf ?" # affecter la valeur "Quoi de neuf ?" à msg
```

```
>>> pi = 3.14159 # assigner sa valeur à la variable pi
```



## Afficher la valeur d'une variable

Pour afficher la valeur à l'écran, il existe deux possibilités. :

1. La première consiste à entrer au clavier le nom de la variable, puis <Enter>.

```
>>> n
7
>>> msg
"Quoi de neuf ? "          #affiche les guillemets, donc le type.
>>> pi
3.14159
```

2. A l'intérieur d'un programme, vous utiliserez toujours l'instruction **print** :

```
>>> print (msg)
Quoi de neuf ?          #pas de guillemets.
```

**Pas de Typage** explicite des variables sous python

# Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément.

Exemple :

```
>>> x = y = 7
>>> x
7
>>> y
7
```

On peut aussi effectuer des *affectations parallèles* à l'aide d'un seul opérateur :

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs 4 et 8,33.

## Exercices d'application

2.1. Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :

```
>>> largeur = 20
```

```
>>> hauteur = 5 * 9.3
```

```
>>> largeur * hauteur
```

```
930
```

2.2. Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c.

Effectuez l'opération  $a - b/c$ .



# Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent, en les combinant avec des **opérateurs** pour former des **expressions**.

Exemple :

**a, b = 7.3, 12**

**y = 3\*a + b/5**

## Exercice

Décrivez ce qui se passe à l'exécution des lignes suivantes :

**r , pi = 12, 3.14159**

**s = pi \* r\*\*2**

**print(s)**

**print(type(r), type(pi), type(s))**

Quelle est, à votre avis, l'utilité de la *fonction* **type()** ?

# Priorité des opérations

**PEMDAS** pour le mémoriser !

1. **P** pour **parenthèses**. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez. Ainsi  $2*(3-1) = 4$  , et  $(1+1)**(5-2) = 8$ .
2. **E** pour **exposants**. Les exposants sont évalués avant les autres opérations. Ainsi  $2**1+1 = 3$  (et non 4), et  $3*1**10 = 3$  (et non 59049 !).
3. **M** et **D** pour **multiplication** et **division**, qui ont la même priorité. Elles sont évaluées avant l'**addition A** et la **soustraction S**, lesquelles sont donc effectuées en dernier lieu. Ainsi  $2*3-1 = 5$  (plutôt que 4), et  $2//3-1 = -1$
4. Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite. Ainsi dans l'expression  $59*100//60$ , la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer  $5900//60$ , ce qui donne **98**. Si la division euclidienne était effectuée en premier, le résultat serait **59** .

# Composition

L'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de **construire des instructions complexes par assemblage de fragments divers**. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
print(type(17 + 3))
```

```
<class 'int'>
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
h, m, s = 15, 27, 34
```

```
print("nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s)
```

Ce que vous placez à la gauche du signe *égale* dans une expression **doit toujours être une variable**, et non une expression : le signe égale n'a pas la même signification qu'en mathématiques, il s'agit d'un symbole d'affectation

**$m + 1 = b$**  est incorrect.

**Par contre,  $a = a + 1$**  est inacceptable en math,  
mais correct en programmation

# Structures de contrôle

**Les structures de contrôle** sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées dans le programme.

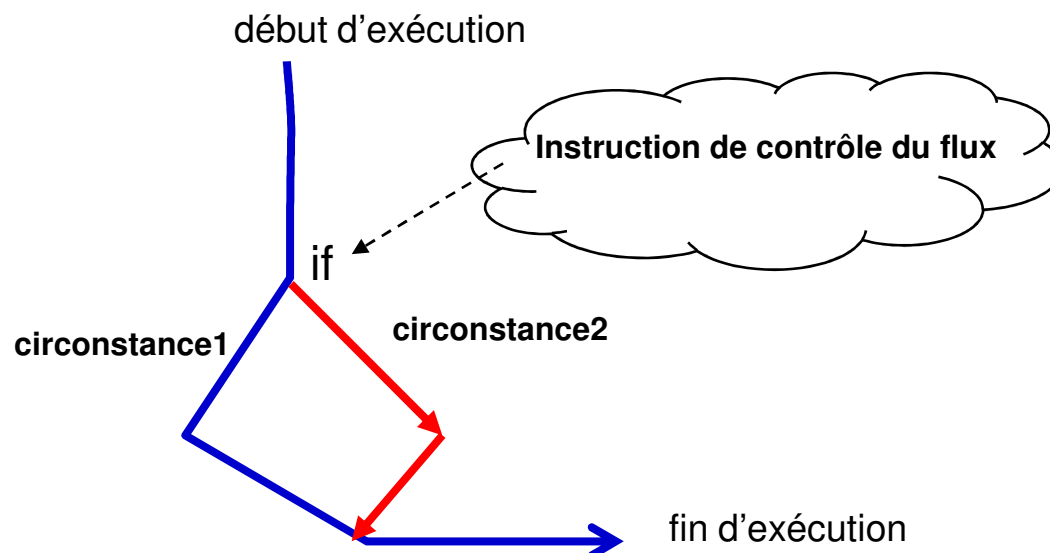
En programmation moderne, il en existe seulement trois : la **séquence**, la **sélection** que nous allons décrire dans ce qui suit, et la **répétition**.

# Séquence d'instructions

**Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du programme.**

Le « chemin » d'exécution est appelé un **flux d'instructions**, et les constructions qui le modifient sont appelées des instructions de contrôle de flux.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une instruction conditionnelle comme l'instruction «if». Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.



# Sélection ou exécution conditionnelle

- C'est une technique permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées.
- Disposer d'instructions capables de tester une certaine condition et de modifier le comportement du programme en conséquence.

## L'instruction **if**.

```
a = 150
if (a > 100):
    print("a dépasse la centaine" )
```



Indentation obligatoire

### En interactif : (mode commande)

Frappez encore une fois <Enter>. Le programme s'exécute, et vous obtenez :  
a dépasse la centaine.

Recommencez le même exercice, mais avec a = 20 en guise de première ligne : cette fois Python n'affiche plus rien du tout.

# Sélection ou exécution conditionnelle

```
a = 20
if (a > 100):
    print("a dépasse la centaine")
else:
    print("a ne dépasse pas cent")
```

En interactif :

Frappez <Enter> encore une fois. Le programme s'exécute, et affiche cette fois :  
**a ne dépasse pas cent.**

Comme vous l'aurez certainement déjà compris, l'instruction **else** (« sinon », en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités

# Sélection ou exécution conditionnelle

On peut faire mieux encore en utilisant aussi l'instruction **elif** (contraction de « else if ») :

```
a = 0
if a > 0 :
    print( " a est positif ")
elif a < 0 :
    print("a est négatif")
else:
    print("a est nul")
```



## Opérateurs de comparaison

La condition évaluée après l'instruction «if» peut contenir les **opérateurs de comparaison** suivants :

**x == y**    # x est égal à y (deux signes « égale » et non d'un seul)  
**x != y**    # x est différent de y  
**x > y**    # x est plus grand que y  
**x < y**    # x est plus petit que y  
**x >= y**   # x est plus grand que, ou égal à y  
**x <= y**   # x est plus petit que, ou égal à y

**Exemple :**

```
a = 7
if (a % 2 == 0):
    print("a est pair")
    print("parce que le reste de sa division par 2 est nul")
else:
    print("a est impair")
```

**Même symbolisme qu'en C++ et en Java**

## Instructions composées – Blocs d'instructions

L'instruction **if** est votre premier exemple d'*instruction composée*

Sous Python, toutes les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point, suivie d'une ou de plusieurs instructions indentées sous cette ligne d'en-tête.

Exemple :

**Ligne d'en-tête:**

**première instruction du bloc**

... ..

... ..

**dernière instruction du bloc**

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, *elles doivent l'être exactement au même niveau* .

Ces instructions indentées constituent ce que nous appellerons désormais un *bloc d'instructions*.

# Instructions imbriquées

Il est parfaitement possible d'imbriquer les unes dans les autres plusieurs instructions composées, de manière à réaliser des structures de décision complexes.

Exemple :

```
if age >= 18:                                     # 1
    if etat_civile == "marié":                     # 2
        if nb_enfants>0:                           # 3
            if sexe == « masculin»:                 # 4
                print("ce monsieur, est majeur et père de famille" )
                # 5
            else:
                print("cette dame est la mère de la famille")
                # 6
        elif etat_civile!= 'veuf' and etat_civile!='veuve': # 7
            print("c'est peut-être un (e) célibataire") # 8
    Else:
        print("c'est un (e ) mineur(e ) ") # 9
```

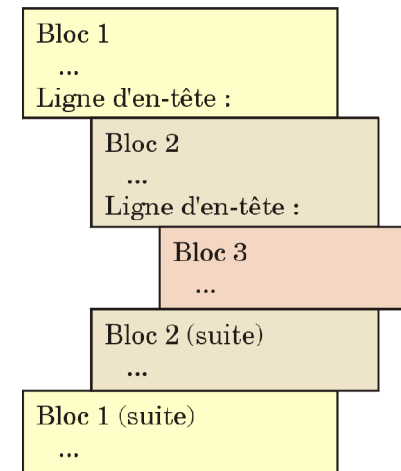
# Quelques règles de syntaxe Python

- **Les limites des instructions et des blocs sont définies par la mise en page**

Avec Python, vous devez utiliser les sauts à la ligne et l'indentation. Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

- **Instruction composée = En-tête , double point , bloc d'instructions indenté**

*Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, ...) se terminant par un double point*



- **Les espaces et les commentaires sont normalement ignorés**

# Instructions répétitives.

## Ré-affectation

*Permet de remplacer l'ancienne valeur d'une variable par une nouvelle*

```
>>> altitude = 320
>>> altitude
320
>>> altitude = 375
>>> altitude
375
```

### Rappels sur l'affectation :

- L'égalité est *commutative*, alors que l'affectation ne l'est pas.
- Les écritures  $a = 7$  et  $7 = a$  sont équivalentes, alors qu'une instruction de programmation telle que  **$375 = \text{altitude}$**  serait illégale.
- l'égalité est permanente, alors que l'affectation ne l'est pas.

```
>>> a = 5
>>> b = a # a et b contiennent des valeurs égales
>>> b = 2 # a et b sont maintenant différentes
```

# Instructions répétitives

## *Exercice (permutation de valeurs)*

Rappelons ici que Python permet d'affecter leurs valeurs à plusieurs variables simultanément :

```
>>> a, b, c, d = 3, 4, 5, 7
```

Supposons par exemple que nous voulions maintenant échanger les valeurs des variables **a** et **c**. (Actuellement, **a** contient la valeur 3, et **c** la valeur 5.

Nous voudrions que ce soit l'inverse). Comment faire ?

4.1. Écrivez les lignes d'instructions nécessaires pour obtenir ce résultat sous python.

## ***Notions de conteneur et d'itérable***

De façon générale, nous parlerons de conteneur pour désigner un type de données permettant de stocker un ensemble d'autres données, en ayant ou non, suivant les types, une notion d'ordre entre ces données.

Nous parlerons aussi d'itérable pour désigner un conteneur que l'on peut parcourir élément par élément.

Pour parcourir ces conteneurs, nous nous servons parfois de **range()** qui fournit un moyen commode pour générer une liste de valeurs.

Par exemple :

```
uneListe = list(range(6))    # list génère une structure liste.  
print(uneListe)             # on affiche la liste
```

Ce qui affichera : [0, 1, 2, 3, 4, 5]

Ces notions seront étudiées plus tard dans le chapitre des listes.

## Instructions répétitives.

*L'une des tâches que les machines font le mieux est la répétition (bouclage)  
**sans erreur** de tâches identiques !*

Python propose deux type de répétitions (ou boucles) :

1. Les répétitions avec **for** dite aussi boucle **déterminée**
2. Les répétitions avec **while** dite aussi boucle **indéterminée**

Voyons la syntaxe de chacune d'elles et leurs caractéristiques.



## ***Répétitions en boucle - l'instruction for***

Quelques exemples de boucle for :

```
for lettre in "ciao":  
    print(lettre)  
#affichage de:  c i a o  
  
for x in [2, 'a', 3.14]:  
    print(x)  
#affichage de :    2 a 3.14  
  
for i in range(5):  
    print(i)  
# affichage de :  0 1 2 3 4  
  
nb_voyelles = 0  
for lettre in "Python est un langage tres sympa":  
    if lettre in "aeiouy":  
        nb_voyelles = nb_voyelles + 1  
  
print(nb_voyelles)  
# affichage du contenu de la variable nb_voyelle qui vaut 10
```

## ***Répétitions en boucle - l'instruction for***

### **Commentaires**

1. Dans le premier exemple on extrait chaque caractère de la chaîne et on l'affiche.
2. Le deuxième exemple c'est le parcours d'un autre itérable qui est la liste et affiche chacun de ses éléments.
3. l'utilisation de la fonction `range()` pour générer un itérable qu'on parcourt à l'aide de la boucle `for`.
4. Le dernier exemple utilise une variable pour compter les voyelles, en énumérant les éléments d'une chaîne de texte et en affichant le nombre trouvé.

## Répétitions en boucle - l'instruction while

*L'une des tâches que les machines font le mieux est la répétition **sans erreur** de tâches identiques !*

Une des méthodes pour programmer ces tâches répétitives est construite autour de l'instruction **while**.

1. Entrer les commandes ci-dessous

```
a = 0
while (a < 7): # (n'oubliez pas le double point !)
    a = a + 1 # (n'oubliez pas l'indentation !)
print(a)
```

2. Exécuter le (ctrl+E si vous travaillez avec pyzo)
3. Comment a-t-il fait le calcul ?

## Répétitions en boucle - l'instruction while

```
a = 0
while (a < 7): # (n'oubliez pas le double point !)
    a = a + 1 # (n'oubliez pas l'indentation !)
print(a)
```

### Commentaires

1. Dans notre exemple, si la condition  $a < 7$  est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.
2. La variable évaluée dans la condition **doit exister au préalable** (Il faut qu'on lui ait déjà affecté au moins une valeur)
3. **Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté**
4. **Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment.**

Exemple de boucle sans fin ou infinie (à éviter) :

```
n = 3
while n < 5:
    print("hello !")
```

Ici la valeur de  $n$  ne change pas dans le corps de la boucle:  
Elle est toujours égale à 3 donc la condition est toujours vraie!!

S'assurer toujours que la condition va devenir fausse, dans le corps de la boucle, à un moment donné.

## ***Répétitions en boucle - Interrompre une boucle : break***

L'instruction permet de sortir immédiatement de la boucle for ou while en cours .

Exemple :

```
for x in range(1, 11):  
    if x == 5:  
        break  
    print(x, end=" ")  
    # end permet de garder le curseur sur la même ligne sans retourner à la ligne  
print ("Boucle interrompue pour x =", x)
```

**Le résultat obtenu :**

**1 2 3 4**

**Boucle interrompue pour x = 5**

## ***Répétitions en boucle - Court-circuiter une boucle : continue***

L'instruction permet de passer immédiatement à l'itération suivante de la boucle for ou while en cours; reprend à la ligne de l'en-tête de la boucle.

Exemple :

```
for x in range(1, 11):  
    if x == 5:  
        continue  
    print(x, end=" ")
```

Le résultat est le suivant :

**1 2 3 4 6 7 8 9 10**  
**# la boucle a sauté la valeur 5**

## Exercices

1. Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7
2. Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous :

**1 euro(s) = 1.65 dollar(s)**

**2 euro(s) = 3.30 dollar(s)**

**4 euro(s) = 6.60 dollar(s)**

**8 euro(s) = 13.20 dollar(s)**

etc. (S'arrêter à 16384 euros)

3. Écrivez un programme qui affiche une suite de 12 nombres dont chaque terme soit égal au triple du terme précédent.

Fin première partie