



Programmation en python

Algorithmie

- Solution « informatique » relative à un problème
- Suite d'actions (instructions) appliquées sur des données
- 3 étapes principales :
 1. saisie (réception) des données
 2. Traitements
 3. restitution (application) des résultats

Programme

- Transcription d'un algorithme avec une syntaxe prédéfinie
- **Python**
- Même principes fondamentaux que les autres langages objets (Delphi, Java, C#, etc.)
- Python s'enrichit de bibliothèques de calcul spécialisées (mathématique, bio informatique, etc.)

Langage interprété : + portabilité application ; - lenteur (R, VBA, **Python**...)

Langage compilé : + rapidité ; - pas portable

(solution possible : write once, compile anywhere ; ex. Lazarus)

Langage pseudo-compilé : + portabilité plate-forme ; - lenteur (?)

(principe : write once, run anywhere ; ex. Java et le principe JIT)



Python est interprété, il est irrémédiablement lent, mais...
on peut lui associer des bibliothèques intégrant des fonctions
compilées qui, elles, sont très rapides.

(1) Python est un [langage de programmation interprété](#). Il est associé à un interpréteur de commandes disponible pour différents OS (Windows, Linux, Mac OS X, etc.)

C'est un « [vrai](#) » langage c.-à-d. types de données, branchements conditionnels, boucles, organisation du code en procédures et fonctions, objets et classes, découpage en modules.

Très bien structuré, facile à appréhender, c'est un langage privilégié pour l'enseignement [1](#), [2](#).

Mode d'exécution : transmettre à l'interpréteur Python le fichier script « **.py** »

(2) Python est associé à de très nombreuses librairies très performantes, notamment des librairies de calcul scientifique (Numpy, SciPy, Pandas, etc.).

De fait, il est de plus en plus populaire, y compris auprès des [data scientists](#).

Il est plus généraliste que R qui est vraiment tourné vers les [statistiques](#).

Les outils de la programmation structurée : pouvoir regrouper du code dans des **procédures** et des **fonctions**. Cela permet de ***mieux organiser*** les applications.

Organisation du code en **modules**. Fichiers « **.py** » que l'on peut appeler dans d'autres programmes avec la commande **import**

Possibilité de distribution des modules : soit directement les fichiers « **.py** », soit sous forme d'**extensions** prêtes à l'emploi.

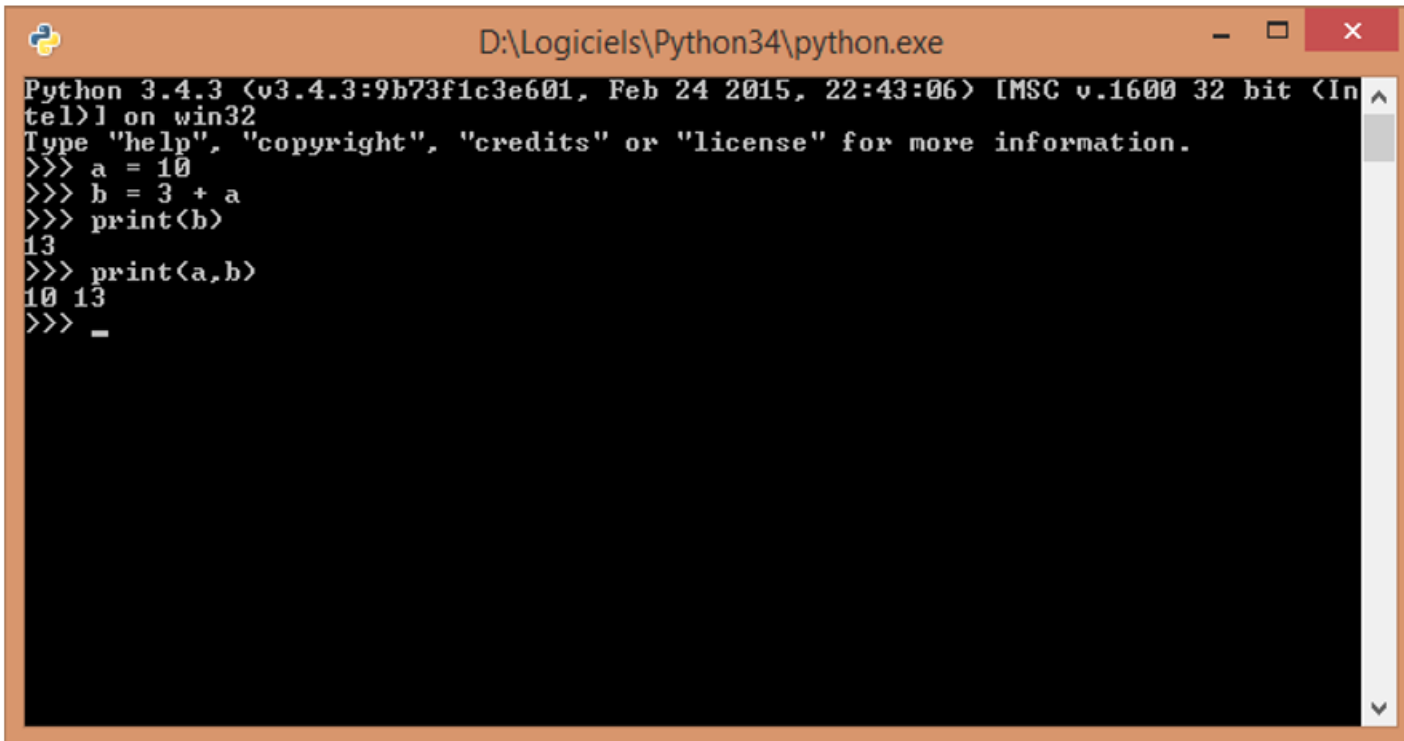
Python est « case sensitive », il différencie les termes écrits en minuscule et majuscule. Des conventions de nommage existent¹. Mais le plus important est d'être raccord avec l'environnement de travail dans lequel vous opérez.

Données typées. Python propose les types usuels de la programmation : entier, réels, booléens, chaîne de caractères.

Structures avancées de données. Gestion des collections de valeurs (énumérations, listes) et des objets structurés (dictionnaires, classes)

Séquences d'instructions, c'est la base même de la programmation, pouvoir écrire et exécuter une série de commandes sans avoir à intervenir entre les instructions.

Structures algorithmiques : les branchements conditionnels et les boucles.

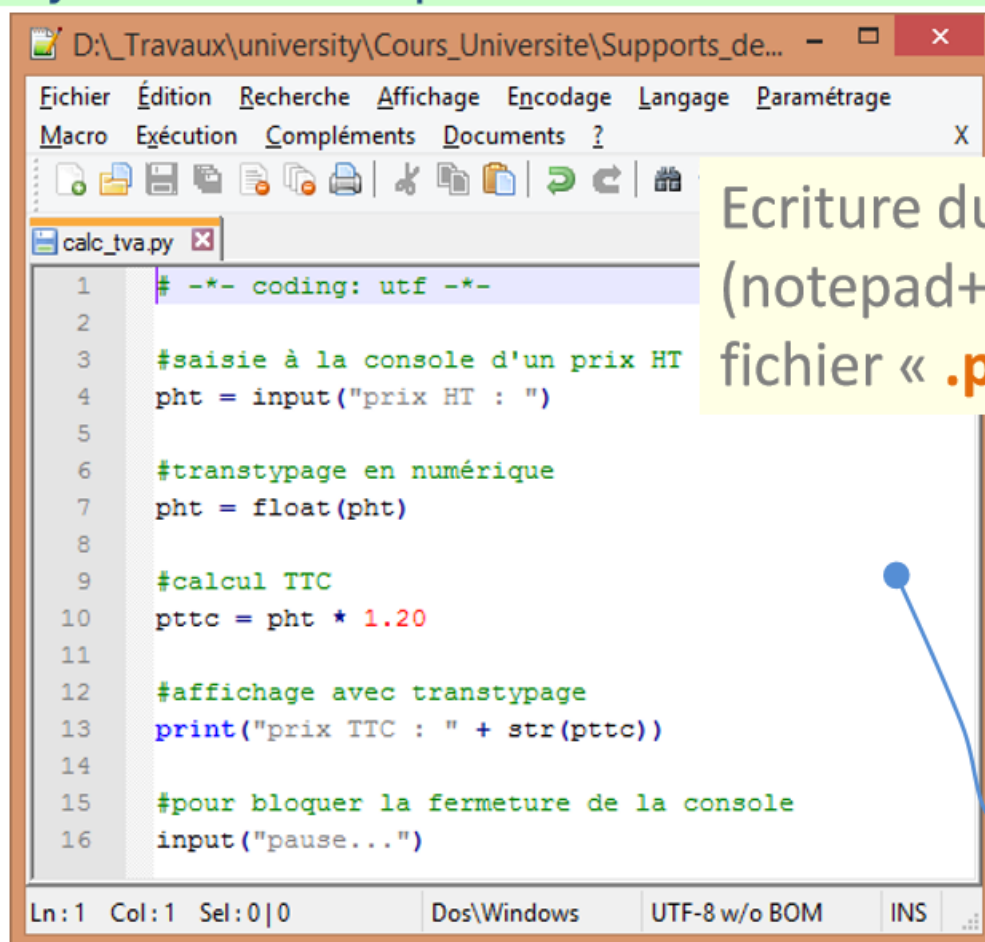


```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a = 10
>>> b = 3 + a
>>> print(b)
13
>>> print(a,b)
10 13
>>> -
```

Lancer la console Python et introduire les commandes de manière interactive.

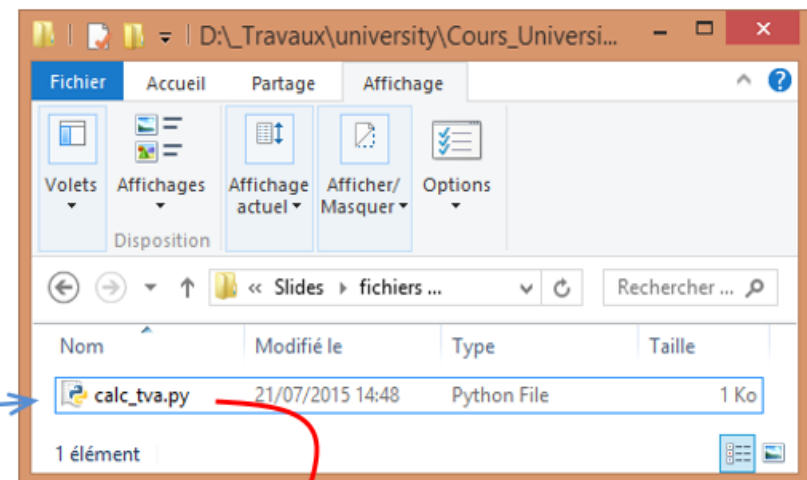
➔ Ce n'est pas adapté pour nous (programmation = enchaînement automatique d'instructions)

Python – Mode opératoire 2

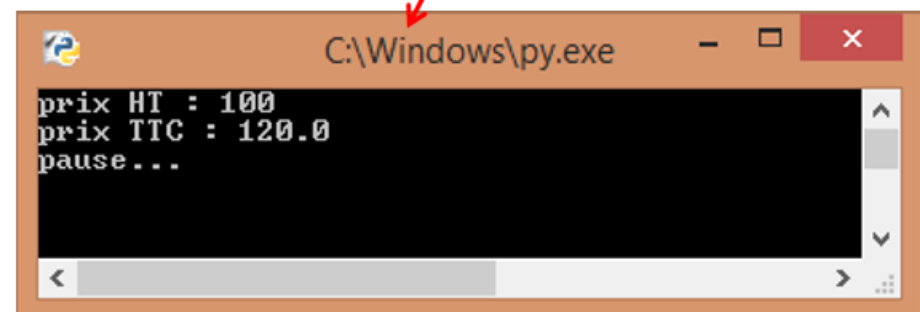


```
1  # -*- coding: utf -*-
2
3  #saisie à la console d'un prix HT
4  pht = input("prix HT : ")
5
6  #transtypage en numérique
7  pht = float(pht)
8
9  #calcul TTC
10 pttc = pht * 1.20
11
12 #affichage avec transtypage
13 print("prix TTC : " + str(pttc))
14
15 #pour bloquer la fermeture de la console
16 input("pause...")
```

Ecriture du code dans un éditeur de code (notepad++) puis l'enregistrer dans un fichier « **.py** »



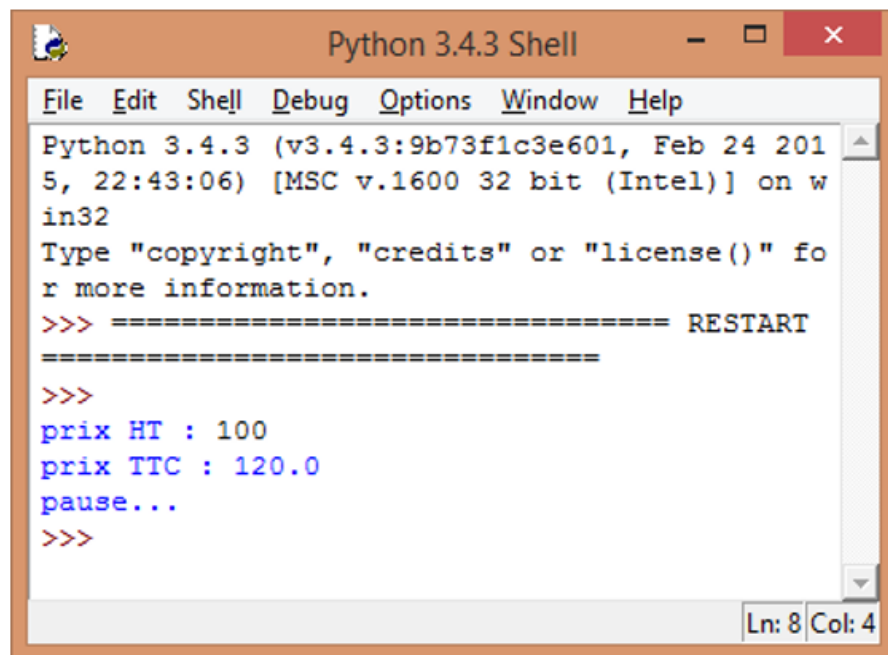
Double cliquer le fichier « **.py** » pour lancer automatiquement le programme dans la console.



```
C:\Windows\py.exe
prix HT : 100
prix TTC : 120.0
pause...
```


Python – Mode opératoire 3 – Utiliser IDLE (environnement de dev. de Python)

Shell : fenêtre d'exécution du programme

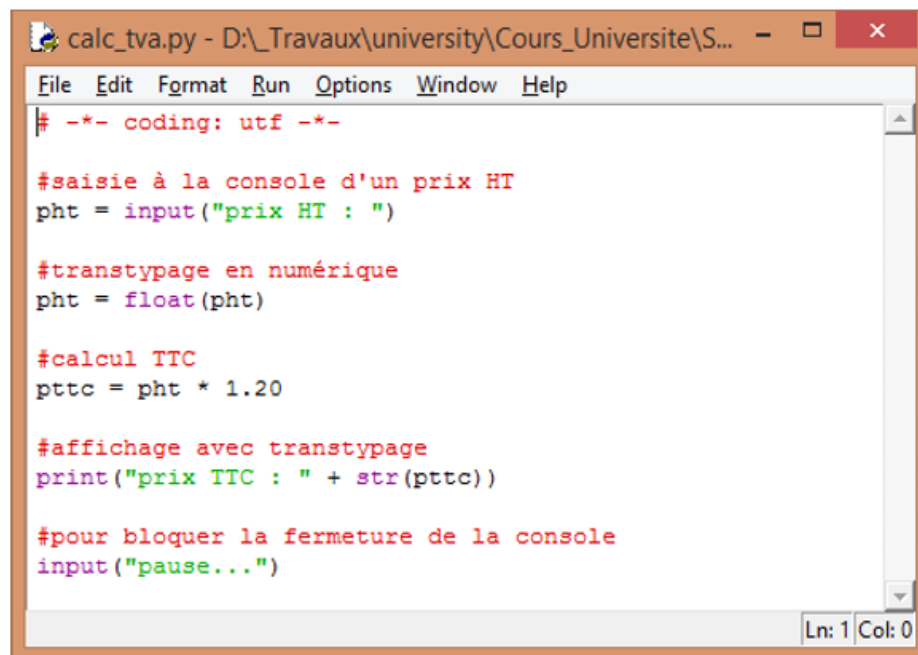


The screenshot shows the 'Python 3.4.3 Shell' window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The text area displays the following content:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART
>>>
prix HT : 100
prix TTC : 120.0
pause...
>>>
```

The status bar at the bottom right indicates 'Ln: 8 Col: 4'.

Editeur de code



The screenshot shows the 'calc_tva.py' code editor window. The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code is as follows:

```
# -*- coding: utf -*-

#saisie à la console d'un prix HT
pht = input("prix HT : ")

#transtypage en numérique
pht = float(pht)

#calcul TTC
pttc = pht * 1.20

#affichage avec transtypage
print("prix TTC : " + str(pttc))

#pour bloquer la fermeture de la console
input("pause...")
```

The status bar at the bottom right indicates 'Ln: 1 Col: 0'.

Menu : RUN / RUN MODULE
(ou raccourci clavier F5)

Permet de mieux suivre l'exécution du programme. Messages d'erreur accessibles, pas comme pour l'exécution console.

Pour lancer le programme

C'est le mode de fonctionnement
que nous allons privilégier !

The screenshot displays the Spyder Python IDE interface. The main window is titled 'Spyder (Python 3.4)' and features a menu bar with options: Fichier, Édition, Recherche, Source, Exécution, Déboguer, Consoles, Outils, Affichage, and Aide. Below the menu is a toolbar with various icons, including a green play button for running code. A tooltip 'Exécuter le fichier (F5)' is visible over the play button. The central pane is the 'Éditeur' (Code Editor), showing a Python script named 'calc_tva.py' with the following code:

```
1 # -*- coding: utf -*-
2
3 #saisie à la console d'un prix HT
4 pht = input("prix HT : ")
5
6 #transtypage en numérique
7 pht = float(pht)
8
9 #calcul TTC
10 pttc = pht * 1.20
11
12 #affichage avec transtypage
13 print("prix TTC : " + str(pttc))
14
15 #fin du programme
16 #inutile de mettre input("pause") ici
```

To the right of the code editor is the 'Inspecteur d'objets' (Object Inspector) pane, which shows the variable 'print' selected. It displays the definition: `print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)` and its type: 'Function of builtins module'. Below this is the 'Console IPython' pane, which shows the execution of the script:

```
In [4]:
runfile('D:/_Travaux/university/Cours_University/Support
s_de_cours/Informatique/Python/Slides/exemples/A/calc_tv
a.py',
wdir='D:/_Travaux/university/Cours_University/Supports_d
e_cours/Informatique/Python/Slides/exemples/A')

prix HT : 100
prix TTC : 120.0

In [5]:
```

At the bottom of the interface is a status bar with the following information: 'Exécuter le fichier', 'Droits d'accès : RW', 'Fins de ligne : CRLF', 'Encodage : UTF', 'Ligne : 16', 'Colonne : 16', and 'Mémoire : 34 %'.

Editeur de code

Informations, dont l'aide
(CTRL+I sur les mots clés)

Console IPython

Sorties + interaction avec l'utilisateur

- Python offre deux outils essentiels : les instructions et les expressions (fonctions, équations, etc.).

les instructions :

Des commandes adressées à l'interpréteur impliquant l'emploi de mots-clés.

```
>>> print "Ceci est mon premier programme PYTHON"
Ceci est mon premier programme PYTHON
>>> |
```

L'instruction print permet d'afficher la donnée fournie, en l'occurrence une chaîne de caractères.

Les instructions peuvent ou non déboucher sur un résultat affiché.

Les symboles >>> et le curseur indiquent que l'interpréteur attend la prochaine instruction Python.

les expressions :

Elles n'utilisent pas de mots-clés.

Il peut s'agir de simples équations, qu'on utilise avec des opérateurs arithmétiques, ou de fonctions, qui sont appelées avec des parenthèses. Les fonctions peuvent ou non accepter une entrée et retourner ou non une valeur.

```
>>> 3 - 5
```

```
-2
```

```
>>> abs(-7)
```

```
7
```

```
>>> _
```

```
7
```

```
>>>
```

La fonction abs prend en entrée un nombre et retourne sa valeur absolue.

Le caractère souligné correspond à la dernière expression évaluée.

La valeur de chaque expression est affichée à l'écran.

Règles et symboles à connaître concernant les instructions en Python

■ Le signe dièse (#)

Les commentaires débutent toujours par un signe dièse (#).

Un commentaire peut débuter n'importe où sur une ligne.

Tous les caractères qui suivent le # sont ignorés par l'interpréteur, jusqu'à la fin de la ligne.

Les commentaires servent à documenter les programmes et améliorer leur lisibilité. Même si Python est un langage facile à apprendre, cela ne dispense pas le programmeur d'utiliser des commentaires de manière adéquate dans son code.

```
>>> # Place aux commentaires.  
>>> print 1 / 3.      # Division réelle.  
0.333333333333  
>>>
```

■ Le caractère de fin de ligne (\n)

Il s'agit du retour à la ligne suivante
instruction par ligne.



qui signifie normalement une

■ La barre oblique inverse (\) (ou antislash).

Cela annonce que l'instruction n'est pas terminée et
qu'elle se poursuit à la ligne suivante.



```
>>> print 10 + 5 \  
      * 4 \  
      - 7  
23  
>>>
```

Il existe 2 cas particuliers où une instruction peut s'étaler sur plusieurs lignes
sans avoir besoin de barres obliques inverses :

- Lorsqu'elle utilise des opérateurs comme les parenthèses, les crochets ou les accolades. [Voir plus tard pour cet usage.](#)
- Lorsque le caractère de retour à la ligne est inséré dans une chaîne entourée de guillemets triples.

```
>>> print '''Place  
aux guillemets triples.'  
Place  
aux guillemets triples.  
>>>
```

```
>>> print """Place  
aux guillemets triples."""  
Place  
aux guillemets triples.  
>>>
```

- Le point-virgule (;) permet de regrouper 2 instructions sur la même ligne.

```
>>> -2 + 5;    print 5 - 2  
3  
3  
>>>
```

Python ne tient pas compte de la présentation : espaces et sauts de lignes.

Un programme pourrait s'écrire en quelques lignes même si ce n'est pas conseillé (attention à la mise en page : présenter un programme de façon lisible).

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

```
-2
```

```
>>> 8 + 4 * 6
```

```
32
```

```
>>> 22 / 3
```



Division entière

```
7
```

```
>>> 2 * (32 - 26)
```

```
12
```

```
>>> 22. / 3
```



Division réelle (l'entier 3 est converti en réel)
à cause de la présence du point décimal.

```
7.333333333333333
```

```
>>> |
```

Python dispose de 2 opérateurs de division :

/ Si les opérandes sont tous deux des entiers, la partie entière du résultat de la division sera retenue.

Autrement, il s'agira d'une véritable division réelle avec comme résultat une valeur réelle.

// La partie entière du résultat de la division, c'est-à-dire le plus grand entier plus petit ou égal au résultat indépendamment du type des opérandes.

```
>>> 11 // 3
3
>>> 23 // 8
2
>>> -6 // 5
-2
>>> 12.8 // 1.1
11.0
>>>
```

Puisque les opérandes sont réelles, le résultat est réel.

% La partie fractionnaire du résultat de la division, c'est-à-dire le résultat de la division moins sa partie entière.

```
>>> 13 % 3
1
>>> 12.8 % 1.1
0.69999999999999973
>>> -6 % 5
4
```

Si les opérandes sont entières, le résultat l'est.

Si les opérandes sont réelles, le résultat l'est.

****** l'opérateur d'exponentiation.

```
>>> 3.1 ** 2
```

```
9.61000000000000012
```

```
>>> 3.1 ** 2.0
```

```
9.61000000000000012
```

```
>>> 3 ** -1
```

```
0.33333333333333331
```

```
>>> (-3)** 2
```

```
9
```

```
>>> 2 ** 0.5
```

```
1.4142135623730951
```

```
>>> --5
```

```
5
```

Erreurs de précision

Erreurs de précision

Opérateurs unaires

```
>>> 3.1 * 3.1
```

```
9.61000000000000012
```

L'opérateur d'exponentiation a une règle de priorité particulière lorsqu'on le combine avec d'autres opérateurs : il est exécuté avant les opérateurs unaires placés à sa gauche, mais après les opérateurs unaires placés à sa droite.

```
>>> 8 ** 2
```

```
64
```

```
>>> -8 ** 2
```

```
-64
```

```
>>> 8 ** -2
```

```
0.015625
```

```
>>> (-8)**2
```

```
64
```

Priorité des opérateurs :

+ et - *, /, //, % + et – unaires **
Bas de la hiérarchie → Haut de la hiérarchie

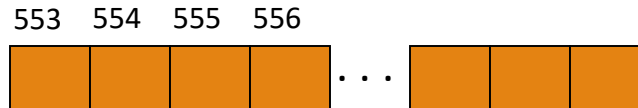
```
>>> -3 + 5**2 -7 / 2  
19
```

```
>>> -3 + 5**2 + -7 / 2  
18
```

On peut utiliser les parenthèses pour clarifier la signification d'une expression ou pour outrepasser l'ordre de priorité des opérateurs : $(5 - 3) * 2 + 4$.

Qu'est-ce qu'une variable ?

- Les programmes doivent mémoriser les données qu'ils utilisent.
- Pour cela, les variables nous fournissent plusieurs représentations et méthodes de stockage des informations.
- Une variable est un emplacement en mémoire principale destiné à recevoir une donnée. Cette zone reçoit une valeur qui peut ensuite être réutilisée.
- La mémoire de votre ordinateur est comparable à des cases alignées une à une. Ces emplacements sont numérotés séquentiellement; il s'agit d'adresses en mémoire.



- Une variable peut occuper une ou plusieurs cases. **Ex. :** la case d'adresse 556.

Ex.: - une donnée numérique avec une précision plus ou moins grande,
- une chaîne de caractères plus ou moins longue.

Taille des variables

- Chaque emplacement en mémoire a la taille d'un octet, i.e. 8 chiffres binaires 0 ou 1 (8 bits ou *Binary digiTS*). La taille d'une variable dépend de son type.
- L'intérêt de la base 2 est qu'elle représente exactement ce que l'ordinateur reconnaît car les ordinateurs ne connaissent pas les lettres, les chiffres, les instructions ou les programmes.

Note : Conversion d'un nombre binaire en base 10.

1010011 en base 2 (i.e. 1010011_2) équivaut en base 10 au nombre suivant :

$$1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 1 \times 2^6$$

ou encore

$$1 + 2 + 0 + 0 + 16 + 0 + 64$$

ce qui donne 83 en base 10 (i.e. 83_{10}).

- Par conséquent, un octet peut prendre les valeurs comprises entre 0 et 255 car $11111111_2 \equiv 255_{10} = 2^8 - 1$.

Conversion d'un nombre décimal en base 2

Exemple : Convertir 99 en base 2.

Puissance de 2 :	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Valeur :	128	64	32	16	8	4	2	1

Calculer la plus grande puissance de 2 plus petit ou égal à 99 i.e. 2^6 .

$99 \geq 2^6 = 64 \Rightarrow$ écrire 1 et soustraire 64 de 99 ce qui donne 35.

$35 \geq 2^5 = 32 \Rightarrow$ écrire 1 et soustraire 32 de 35 ce qui donne 3.

$3 < 2^4 = 16 \Rightarrow$ écrire 0.

$3 < 2^3 = 8 \Rightarrow$ écrire 0.

$3 < 2^2 = 4 \Rightarrow$ écrire 0.

$3 \geq 2^1 = 2 \Rightarrow$ écrire 1 et soustraire 2 de 3 ce qui donne 1.

$1 \geq 2^0 = 1 \Rightarrow$ écrire 1 et soustraire 1 de 1 ce qui donne 0.

Résultat : 1100011_2 .

Identificateur de variable

- Pour identifier une variable, on utilise un identificateur pour désigner le nom de cette variable.
- L'identificateur doit indiquer le rôle joué par cette variable; il faut éviter d'utiliser des noms qui n'évoquent rien. **Ex. :** Rayon_du_cercle au lieu de x.
- Cela vous épargnera de devoir connaître l'adresse réelle en mémoire de celle-ci.

Règles à respecter pour les noms de variables

- Une séquence de lettres ($a \rightarrow z, A \rightarrow Z$) et de chiffres (0 à 9) qui doit toujours commencer par une lettre. Le symbole `_` est considéré comme une lettre.
- Aucune lettre accentuée, cédille, espace, caractère spécial à l'exception du caractère souligné `_`.
- Les minuscules et les majuscules sont des lettres différentes.

Exemple :

Variable_entiere, entier1, mot_en_francais sont valides mais *1er_entier, nom.2, nom de variable, deuxième_entier* et *a-b* ne le sont pas.

Éviter d'utiliser le symbole `_` comme 1^{er} caractère car il peut être utilisé pour définir des entités spéciales pour Python.

- Les 28 mots réservés ci-dessous ne peuvent être utilisés comme nom de variable :

and	continue	else	for	import	not	raise
assert	def	except	from	in	or	return
break	del	exec	global	is	pass	try
class	elif	finally	if	lambda	print	while

Attention :

- Vous devez saisir les majuscules et les minuscules exactement telles qu'elles apparaissent.

Main, main et MAIN sont distincts l'un de l'autre.

Python distingue les majuscules et les minuscules :

Nom_de_variable est différent de nom_de_variable.

- Bien que la longueur des identificateurs ne soit plus un problème dans les langages de programmation d'aujourd'hui, utilisez des noms de taille raisonnable ayant une signification.

Il peut exister des limites qui peuvent changer d'un interpréteur à l'autre.

Opérateur d'affectation =

Syntaxe : identificateur_de_variable = expression

Exemple :

```
>>> entier1 = 8
>>> entier2 = entier1 - 5
>>> entier1 = entier1 + 1
>>> print entier1, entier2
9 3
>>>
```

But : stocker la valeur d'une expression dans une variable.

Note : Les affectations ne sont pas des expressions; elles n'ont pas de valeurs inhérentes mais, il est permis d'enchaîner plusieurs affectations.

```
>>> x = 1
>>> y = x = x + 1
>>> print x, y
2 2
>>>
```



On ne peut pas écrire : $y = (x = x + 1)$.

Affectation

```
>>> i = 2
>>> message = "Ceci est un message"
>>> valeur_de_pi = 3.14159
>>> |
```

Dans le langage Python, ces instructions d'affectation réalisent les opérations suivantes :

- créer et mémoriser un nom de variable,
- lui attribuer implicitement un type bien déterminé
(entier, réel, chaîne de caractères, ...)
- lui associer une valeur particulière,
- Établir un lien entre le nom de la variable et l'emplacement mémoire renfermant la valeur associée.

Note : Pour définir le type des variables avant de pouvoir les utiliser, il suffit d'assigner une valeur à un nom de variable pour que celle-ci soit automatiquement créée avec le type qui correspond le mieux à la valeur fournie. Python possède donc un typage dynamique et non un typage statique (C++, JAVA).

Affichage de la valeur d'une variable

```
>>> s = "Luc"
>>> entier = 3
>>> reel = 10.2
>>> s
'Luc'
>>> entier, reel
(3, 10.199999999999999)
>>> s = 1
>>> s
1
>>> |
```

→ Affichage de la chaîne de caractères s.

→ Affichage des variables entier et reel.

→ Affectation d'une valeur à une variable s. Cela signifie que l'ancienne variable s n'est plus accessible.

Ce mode d'affichage élémentaire est utilisé en mode interactif. Autrement, on opte pour l'instruction `print`.

```
>>> chaine = "Oh! la! la!"
>>> indice = 5
>>> print chaine, indice
Oh! la! la! 5
>>> chaine, indice
('Oh! la! la!', 5)
>>> |
```

→ Les variables sont séparées par des virgules.

→ Notez les différences entre les modes d'affichage.

Affectations multiples et parallèles

```
>>> centre_x, centre_y, rayon = 1.0, 0.5, 12
```

→ Affectation parallèle

```
>>> print centre_x, centre_y, rayon
```

```
1.0 0.5 12
```

```
>>> centre_x = centre_y = 0
```

→ Affectation multiple

```
>>> print centre_x, centre_y, rayon
```

```
0 0 12
```

```
>>> |
```

Une autre façon de réaliser l'affectation de plusieurs variables à la fois est de placer la liste des variables à gauche de l'opérateur d'affectation et la liste des expressions à droite de l'opérateur d'affectation.

```
>>> x = 5.0
```

```
>>> y = 10.0
```

```
>>> x, y, z = x + y, x - y, 25
```

```
>>> print x, y, z
```

```
15.0 -5.0 25
```

```
>>> u = v = 1.
```

```
>>> u, v, w = u + w, v + w, 3.4
```

```
Traceback (most recent call last):
```

```
File "<pyshell#6>", line 1, in <module>
```

```
u, v, w = u + w, v + w, 3.4
```

```
NameError: name 'w' is not defined
```

```
>>> x, y = 10, 20
```

```
>>> print x, y
```

```
10 20
```

```
>>> x, y = y, x
```

```
>>> print x, y
```

```
20 10
```

```
>>>
```

→ Permutation de variables sans utiliser de variable temporaire.

Opérateurs et expressions

```
>>> x, y = 3, 13
>>> x, y = x ** 2, y % x
>>> print x, y
9 1
>>> a = 10
>>> a = a + 1
>>> print a
11
>>> |
```

→ l'opérateur `**` d'exponentiation,
l'opérateur modulo `%`.

→ Dans une affectation parallèle avec des expressions,
l'évaluation de celles-ci se fait avec la valeur des
variables avant l'exécution de l'instruction.

Priorité des opérateurs

- Ordre de priorité : les parenthèses, `**`, `*` et `/`, `+` et `-`.
- Si 2 opérateurs ont même priorité, l'évaluation est effectuée de gauche à droite.

```
>>> x = 5
>>> y = 3
>>> print x - 1 + y ** 2 * 3 / 6
8
>>>
```

Opérateurs d'affectation +=, -=, *=, /=, %=, **=, //=

Syntaxe : identificateur_de_variable op expression

But : L'évaluation d'une expression et une affectation sont combinées.

Exemple :

```
>>> x = 5
>>> x **= 2    # Équivaut à x = x ** 2.
>>> x
25
>>> x %= 3
>>> x
1
>>> x //= 2
>>> x
0
>>>
```

Note : Contrairement à C++, Python ne renferme pas les opérateurs ++ et --.

Saisie de données au clavier

La fonction input

- Elle provoque une interruption dans le programme courant où l'utilisateur est invité à entrer des données au clavier et à terminer avec <Enter>. L'exécution du programme se poursuit alors et la fonction fournit en retour les valeurs entrées par l'utilisateur.
- Ces valeurs peuvent alors être stockées dans des variables dont le type correspond à celui des données entrées.

```
>>> print "Entrez un entier positif :"  
>>> n = input()  
>>> print "Deux puissance ", n, " donne comme résultat : ", 2**n  
Deux puissance 5 donne comme résultat : 32
```

- La fonction input est soit, sans paramètre ou soit, avec un seul paramètre, une chaîne de caractères, lequel est un message explicatif destiné à l'utilisateur.

```
>>> nom = input("Entrez votre nom (entre guillemets) :")  
>>> print nom
```



La chaîne de caractères est entrée entre des apostrophes ou des guillemets.

- On peut saisir plusieurs données simultanément.

```
>>> t, u, v, w = input(  
"oui", 34, "non", 59  
>>> print t, u, v, w  
oui 34 non 59
```

- On doit fournir exactement le nombre de données voulues à la saisie.

Exemple :

```
>>> #      Ce programme saisit deux valeurs entières au clavier,  
>>> #      calcule le quotient et  
>>> #      affiche le résultat.  
>>>  
>>> m, n = input("Entrez 2 valeurs entières au clavier :")  
Entrez 2 valeurs entières au clavier :34, 6  
>>> resultat = m / n  
>>> print "Le quotient de ", m, " par ", n, " est : ", resultat  
Le quotient de 34 par 6 est : 5  
>>>
```


Erreur d'exécution ou de logique

En exécutant ce programme, si vous entrez au clavier comme 2^{ième} valeur entière la valeur nulle, le programme terminera anormalement.

Un message sera affiché indiquant que l'on a tenté d'effectuer une division par zéro.

C'est une erreur d'exécution ou de logique.

Exemple :

```
>>> m, n = input("Entrez 2 valeurs entières au clavier :")
Entrez 2 valeurs entières au clavier :12, 0
>>> resultat = m / n
```

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

resultat = m / n

ZeroDivisionError: integer division or modulo by zero

```
>>>
```

Exemple :

```
>>> Valeur = 3.14159 * rayon ** 2
```

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

```
Valeur = 3.14159 * rayon ** 2
```

NameError: name 'rayon' is not defined

```
>>>
```

Erreurs à l'interprétation

- Lorsque vous écrivez une commande ou une expression en Python avec une erreur syntaxique, l'interpréteur affiche un message d'erreur et rien n'est exécuté. Il faut recommencer.

Exemple :

```
>>> Montant_en_$ = 35.56
```

SyntaxError: invalid syntax

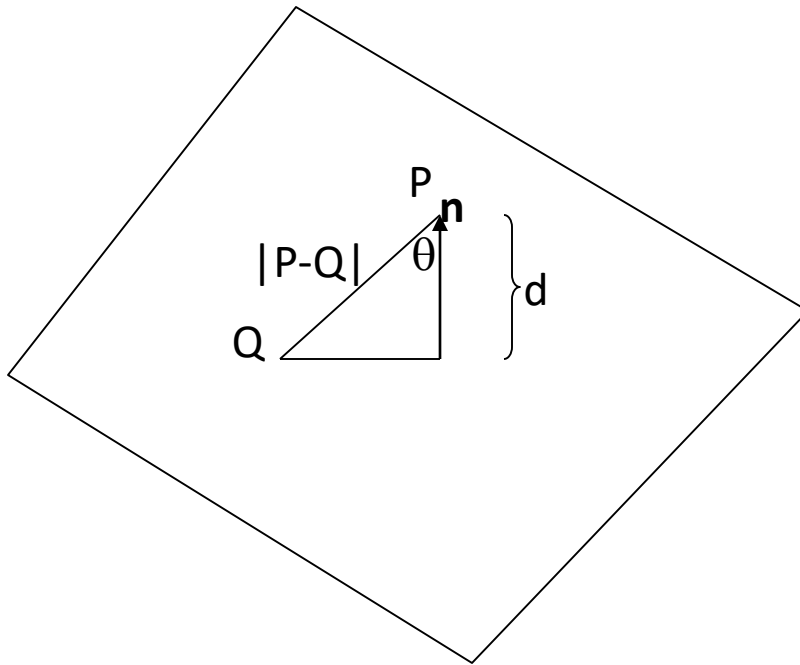
```
>>>
```

Circonférence et aire d'un cercle

```
>>> #    Ce programme saisit au clavier le rayon d'un cercle,  
>>> #    calcule la circonférence et l'aire du cercle et  
>>> #    affiche ces résultats.  
>>>  
>>> rayon = input("Entrez le rayon du cercle :")  
Entrez le rayon du cercle :24.5  
>>> print "Circonférence du cercle : ", 2.0 * 3.14159 * rayon  
Circonférence du cercle : 153.93791  
>>> print "Aire du cercle : ", 3.14159 * rayon **2  
Aire du cercle : 1885.7393975  
>>>
```

Distance entre un point et un plan dans l'espace à trois dimensions

Calculer la distance d d'un point $P = (p_1, p_2, p_3)$ à un plan défini par un point $Q = (q_1, q_2, q_3)$ et un vecteur normale unitaire $\mathbf{n} = (n_1, n_2, n_3)$.



\mathbf{n} est unitaire



$$\Rightarrow d = \frac{||\mathbf{P} - \mathbf{Q}|| \cos \theta}{||\mathbf{n}||} = \frac{(\mathbf{P} - \mathbf{Q}) \cdot \mathbf{n}}{||\mathbf{n}||} = (\mathbf{P} - \mathbf{Q}) \cdot \mathbf{n}$$

Qu'arrive-t-il si P fait partie du plan ?

Distance entre un point et un plan dans l'espace à trois dimensions

```
>>> # Ce programme saisit au clavier les données suivantes :
>>> #     - les coordonnées d'un point P = (Px, Py, Pz) quelconque,
>>> #     - les coordonnées d'un point Q = (Qx, Qy, Qz) d'un plan,
>>> #     - les coordonnées de la normale N = (Nx, Ny, Nz) du plan,
>>> # calcule la distance entre le point P et le plan et
>>> # affiche le résultat.
>>>
>>> Px, Py, Pz = input("Saisie des coordonnées de P : ")
Saisie des coordonnées de P : 2, 3, 4
>>>
>>> Qx, Qy, Qz = input("Saisie des coordonnées de Q : ")
Saisie des coordonnées de Q : 3, 0, 4
>>> Nx, Ny, Nz = input("Saisie des coordonnées de N : ")
Saisie des coordonnées de N : 0, 2, 4
>>>
>>> distance = (Px - Qx) * Nx + (Py - Qy) * Ny + (Pz - Qz) * Nz
>>>
>>> print "Affichage de la distance entre P et le plan : ", distance
Affichage de la distance entre P et le plan : 6
>>>
```

Calcul du produit de deux quaternions

Un quaternion est défini comme un quadruplet de nombres réels, le premier élément étant un « scalaire », et les trois éléments restants formant un « vecteur ».

Soient deux quaternions

$$Q_1 = (s_1, v_1), \quad \text{où } s_1 \text{ est un nombre réel, } v_1 = (v_{1x}, v_{1y}, v_{1z}),$$
$$Q_2 = (s_2, v_2), \quad \text{où } s_2 \text{ est un nombre réel, } v_2 = (v_{2x}, v_{2y}, v_{2z}),$$

alors

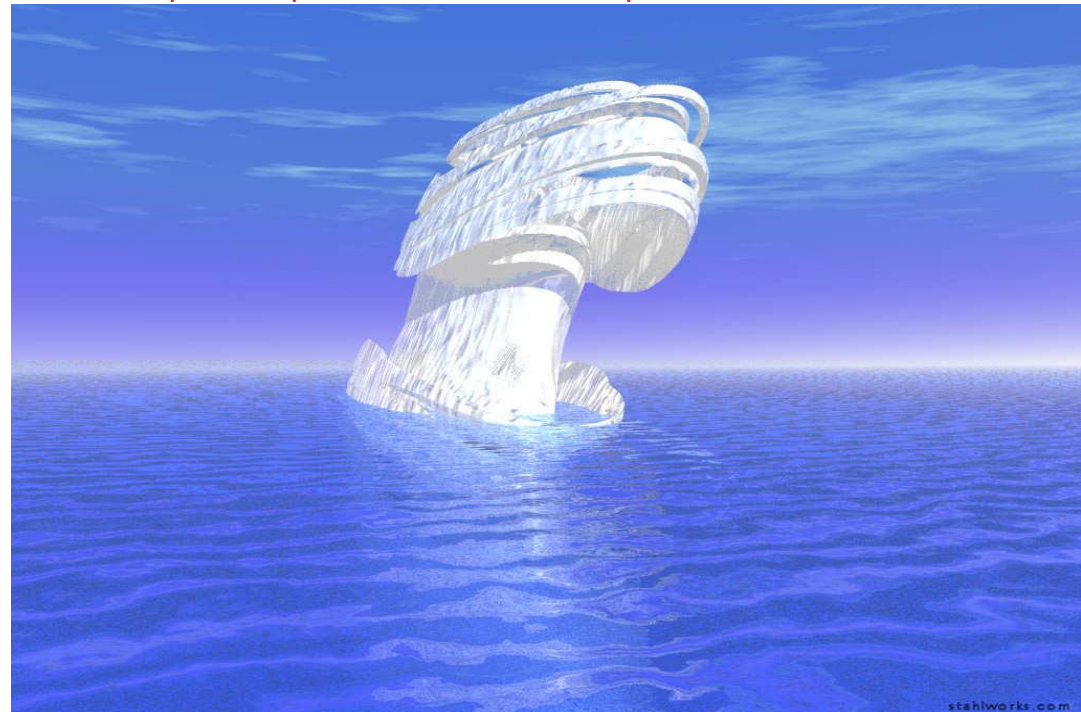
$$Q_1 \cdot Q_2 = (s_1 s_2 - v_1 \bullet v_2, \\ s_1 v_2 + s_2 v_1 + v_1 \times v_2)$$

où \bullet désigne le produit scalaire,
 \times désigne le produit vectoriel.

Note :

$$v_1 \times v_2 = (v_{1y} v_{2z} - v_{1z} v_{2y}, \\ v_{1z} v_{2x} - v_{2z} v_{1x}, \\ v_{1x} v_{2y} - v_{1y} v_{2x})$$

Sculpture représentée à l'aide de quaternions.



Calcul du produit de deux quaternions

```
>>> #    Ce programme saisit au clavier les coordonnées de 2 quaternions,
>>> #    calcule le produit de ces 2 quaternions,
>>> #    et affiche les coordonnées du quaternion obtenu.
>>>
>>> s1, v1x, v1y, v1z = input("Entrez les coordonnées du premier quaternion : ")
Entrez les coordonnées du premier quaternion : 2.3, 0.0, -1.3, 6.1
>>> s2, v2x, v2y, v2z = input("Entrez les coordonnées du deuxième quaternion : ")
Entrez les coordonnées du deuxième quaternion : 7.9, -0.7, 4.1, 5.5
>>>
>>> #    Calcul du produit des quaternions (s3, v3).
>>>
>>> s3 = s1 * s2 - v1x * v2x - v1y * v2y - v1z * v2z
>>> v3x = v1y * v2z - v1z * v2y
>>> v3y = v1z * v2x - v2z * v1x
>>> v3z = v1x * v2y - v1y * v2x
>>>
>>> #    Affichage du produit des 2 quaternions.
>>>
>>> print ("(", s3, ", (", v3x, ", ", v3y, ", ", v3z, ")")"
( -10.05 , ( -32.16 , -4.27 , -0.91 ))
>>>
```

Types de données élémentaires

Type « int »

- Les entiers ordinaires de Python correspondent aux entiers standards.
- La plupart des machines (32 bits) sur lesquelles s'exécute Python permettent de coder tous les entiers entre -2^{31} et $2^{31} - 1$, i.e. -2 147 483 648 et 2 147 483 647.
- Normalement, les entiers sont représentés en format décimal en base 10, mais on peut aussi les spécifier à l'aide de leur représentation en base 8 ou 16. Les valeurs octales utilisent le préfixe "0", tandis que les valeurs hexadécimales sont introduites par un préfixe "0x" ou "0X" .

```
>>> x, y, z = 123, -123, 2147483650
>>> print x, y, z
123 -123 2147483650
>>> print x, y, type(z), z
123 -123 <type 'long'> 2147483650
>>> w = 0x709A
>>> x = -0XEF2
>>> print w, x
28826 -3826
```

→ Aucun débordement de capacité.

→ La fonction permet de vérifier à chaque itération le type de la variable z.

Types de données élémentaires

Type « long »

Python est capable de traiter des nombres entiers aussi grands que l'on veut. Toutefois, lorsque ceux-ci deviennent très grands, les variables définies implicitement comme étant de type `int` (32 bits) sont maintenant de type `long`.

Ex. :

```
u, v, w = 1, 1, 1
while (w <= 50) :
    if (w >= 40) :
        print w, " : ", v, type(v)
    u, v, w = v, u + v, w + 1
```

```
40 : 165580141 <type 'int'>
41 : 267914296 <type 'int'>
42 : 433494437 <type 'int'>
43 : 701408733 <type 'int'>
44 : 1134903170 <type 'int'>
45 : 1836311903 <type 'int'>
46 : 2971215073 <type 'long'>
47 : 4807526976 <type 'long'>
48 : 7778742049 <type 'long'>
49 : 12586269025 <type 'long'>
50 : 20365011074 <type 'long'>
```

⇒ Il n'y a pas de débordement de capacité.

Types de données élémentaires

Ex. :

```
>>> r = 10L
>>> s = 12345678901234567890
>>> t = -0XABCDEF0123456789ABCDEF
>>> print r, s, t
10 12345678901234567890 -207698809136909011942886895
>>>
```

Les entiers longs sont identifiés par la lettre « L » ou « l », ajoutée à la fin de la valeur numérique. Pour éviter toute confusion, il est préférable d'utiliser L.

```
>>> A= 0XABCDEF01234567890AL
>>> print A
3169232317152542296330
>>> A
3169232317152542296330L
>>>
```

```
>>> x = 12345678901234567890
>>> x = x + 1
>>> print x
12345678901234567891
>>>
```

Types de données élémentaires

Type « bool »

Les valeurs booléennes `True` ou `False` constituent un cas particulier des entiers. Dans un contexte numérique tel qu'une addition avec d'autres nombres, `True` est traité comme un entier valant 1, et `False` a la valeur 0.

Type « complex »

Cela représente des nombres complexes de la forme $a + bj$ où a et b sont des réels en virgule flottante, a représente la partie réelle, b la partie imaginaire et $j^2 = -1$.

Ex. :
6.25 + 2.3j
-4.37 + 13j
0 + 1j
1 + 0j

```
>>> x = 1.2 + 4.5j
>>> print x
(1.2+4.5j)
>>> y = 1.2 - 4.5j
>>> print x * y
(21.69+0j)
>>> print x - y
9j
```

```
>>> Nombre_complexe = 1.57 - 7.9j
>>> Nombre_complexe.real # partie réelle
1.5700000000000001
>>> Nombre_complexe.imag # partie imaginaire
-7.9000000000000004
>>> Nombre_complexe.conjugate() # conjugué
(1.5700000000000001+7.9000000000000004j)
>>> Nombre_complexe * Nombre_complexe.conjugate()
(64.874899999999997+0j)
>>>
```

Type « float »

Les données ayant un point décimal ou un exposant de 10.

Ex. : 3.14159 -12.13 2e13 0.3e-11

Cela permet de manipuler des nombres positifs ou négatifs compris entre 10^{-308} et 10^{308} avec une précision de 12 chiffres significatifs.

```
u, v = 1., 1
while (v <= 40) :
    print v, " : ", u ** (u * u)
    u, v = u + 1, v + 1
```

En principe, 52 bits sont alloués à la mantisse, 11 à l'exposant et un bit pour le signe. En pratique, cela peut dépendre de la machine utilisée et de l'environnement de programmation.

```
1 : 1.0
2 : 16.0
3 : 19683.0
4 : 4294967296.0
5 : 2.98023223877e+017
6 : 1.03144247985e+028
7 : 2.56923577521e+041
8 : 6.27710173539e+057
9 : 1.96627050476e+077
10 : 1e+100
11 : 1.019799757e+126
12 : 2.52405858453e+155
13 : 1.80478943437e+188
14 : 4.37617814536e+224
15 : 4.17381588439e+264
16 :
```

```
Traceback (most recent call last):
  File "E:\essai.py", line 3, in <module>
    print v, " : ", u ** (u * u)
OverflowError: (34, 'Result too large')
```

```
>>>
```

DR.FELMENDJI

Conversion de types numériques

- Jusqu'à maintenant, nous avons appliqué les opérateurs précédents à des opérandes de même type. Qu'arrive-t-il lorsque les opérandes sont de types différents ?

```
>>> 3 + 5.4  
8.4000000000000004  
>>>
```

- Il s'agit de convertir l'un des opérandes au type de l'autre opérande avant d'effectuer l'opération.
- Toutes les conversions ne sont pas possibles comme celle d'un réel en entier, ou celle d'un nombre complexe en n'importe quel autre type non complexe.

Règles de conversion :

- Si l'un des arguments est un nombre complexe, l'autre est converti en complexe.
- Sinon, si l'un des arguments est un nombre réel, l'autre est converti en réel.
- Sinon, si l'un des arguments est un long, l'autre est converti en long.
- Sinon, tous deux doivent être des entiers ordinaires et aucune conversion n'est nécessaire.

cmp()

Prend en entrée deux expressions a et b de valeurs numériques et retourne

- 1 si $a < b$,
- 0 si a est égale à b,
- +1 si $a > b$.

```
>>> u = 3.14
>>> v = 2.6
>>> cmp(u**2 - 2, v*3 + 4)
-1
>>> cmp(3, 3)
0
>>> cmp(1, (1. / 3) * 3.)
0
>>> cmp(u, 3)
1
>>>
```

type() Retourne le type de l'argument.

```
>>> u = 1 + 3.4j
>>> type(u)
<type 'complex'>
>>> type(3 + 4.4)
<type 'float'>
>>>
```

`bool()`

Retourne True si l'argument est différent de 0. False autrement.

```
>>> bool(3.14)
True
>>> bool(0)
False
>>> bool(-3.14)
True
```

`int()`

Prend en entrée comme argument une expression de valeur numérique ou une chaîne de caractères représentant un entier et retourne le résultat de l'expression où la partie fractionnaire a été omise ou la chaîne de caractères convertie en entier.

`int()` supprime le point décimal et toutes les décimales qui suivent (le nombre est tronqué).

```
>>> int(3.14)
3
>>> int(-3.14)
-3
>>> int("3")
3
```

long()

Prend en entrée comme argument une expression de valeur numérique ou une chaîne de caractères représentant un entier et retourne le résultat de l'expression sous forme d'entier long (partie fractionnaire omise) ou la chaîne de caractères convertie en entier long.

```
>>> long(3.14)
3L
>>> long(0xabc)
2748L
>>> long("3L")
3L
>>> long("3")
3L
```

float()

Prend en entrée comme argument une expression de valeur numérique ou une chaîne de caractères représentant un nombre et retourne le résultat de l'expression sous forme de réel ou la chaîne de caractères convertie en réel.

```
>>> float(3)
3.0
>>> float("3")
3.0
>>> float("3.4")
3.3999999999999999
```

complex()

```
>>> complex(3.2, 7)
(3.2000000000000002+7j)
>>> complex("3.2+7j")
(3.2000000000000002+7j)
>>> complex(3.4)
(3.3999999999999999+0j)
```


`abs()`

Retourne la valeur absolue de l'argument.

```
>>> abs(-1 + 0.25)
0.75
>>> abs(3 - 2j)
3.6055512754639896
>>> (3 - 2j)*(3 + 2j)
13.0
>>> abs(3 - 2j) ** 2
13.0000000000000002
```

`pow(m, n)` ou `pow(m, n, p)`

Dans les 2 cas, m^n est d'abord calculé; puis, si le troisième argument est fourni, alors $(m ** n) \% p$ est retourné; sinon, $m ** n$ est retourné.

```
>>> pow(4, 2)
16
>>> pow(3, 2, 5)
4
```

`round()`

Arrondit un nombre réel à l'entier le plus proche et retourne le résultat comme une valeur réelle. Un 2^{ème} paramètre présent arrondit l'argument au nombre de décimales indiqué.

```
>>> round(3.4999999)
3.0
>>> round(2.8)
3.0
>>> round(253.358901234, 2)
253.36000000000001
>>> round(-3.4), round(-3.5)
(-3.0, -4.0)
```

Fonctions ne s'appliquant qu'à des entiers ordinaires ou longs

Représentation dans une base

- Nous savons que Python gère automatiquement des représentations octales et hexadécimales, en plus de la représentation décimale.
- Python dispose aussi de deux fonctions intégrées, `oct()` et `hex()`, qui retournent des chaînes de caractères contenant respectivement la représentation octale ou hexadécimale d'une expression entière quelle qu'en soit la représentation.

```
>>> print type(hex(255)), hex(255)
<type 'str'> 0xff
>>> hex(12345678901234567890L)
'0xab54a98ceb1f0ad2L'
>>> oct(8**4)
'010000'
>>>
```

Fonctions ne s'appliquant qu'à des entiers ordinaires ou longs

Conversion ASCII

- Chaque caractère est associé à un nombre unique entre 0 et 255, son indice dans la table ASCII (« American Standard Code for Information Interchange »).
- La table ASCII est la même sur tous les ordinateurs ce qui garantit un comportement identique des programmes sur différents environnements.

Code décimal	Code hex	Caractère	Code décimal	Code hex	Caractère	Code décimal	Code hex	Caractère	Code décimal	Code hex	Caractère
0	00		16	10		32	20	Espace	48	30	0
1	01		17	11		33	21	!	49	31	1
2	02		18	12		34	22	"	50	32	2
3	03		19	13		35	23	#	51	33	3
4	04		20	14		36	24	\$	52	34	4
5	05		21	15		37	25	%	53	35	5
6	06		22	16		38	26	&	54	36	6
7	07	\a	23	17		39	27	'	55	37	7
8	08	\b	24	18		40	28	(56	38	8
9	09	\t	25	19		41	29)	57	39	9
10	0A	\n	26	1A		42	2A	*	58	3A	:
11	0B	\v	27	1B		43	2B	+	59	3B	;
12	0C	\f	28	1C		44	2C	,	60	3C	<
13	0D	\r	29	1D		45	2D	-	61	3D	=
14	0E		30	1E		46	2E	.	62	3E	>
15	0F		31	1F		47	2F	/	63	3F	?

Fonctions ne s'appliquant qu'à des entiers ordinaires ou longs

- On retrouve ici les 128 premiers caractères qui renferment notamment les majuscules, les minuscules, les chiffres et les signes de ponctuation. Des codes étendus sont disponibles.

Code décimal	Code hex	Caractère	Code décimal	Code hex	Caractère	Code décimal	Code hex	Caractère	Code décimal	Code hex	Caractère
64	40	@	80	50	P	96	60	'	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	
77	4D	M	93	5D]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

Fonctions ne s'appliquant qu'à des entiers ordinaires ou longs

- `chr()` Prend comme argument une expression entière entre 0 et 255 inclusivement et retourne le caractère ASCII sous la forme d'une chaîne de caractères.
- `ord()` Prend comme argument un caractère ASCII sous la forme d'une chaîne de caractères de longueur 1 et retourne le code ASCII correspondant.

```
>>> print "Le caractère 5 en code ASCII est : ", ord("5")
Le caractère 5 en code ASCII est : 53
>>> print "Le code ASCII 36 désigne le caractère : ", chr(36)
Le code ASCII 36 désigne le caractère : $
>>>
```

Nous verrons plus loin des modules renfermant d'autres fonctions manipulant des expressions numériques.

Type « string »

Une chaîne de caractères délimitée par des apostrophes ou des guillemets.

```
mot1 = "C'est une grosse journée;"  
mot2 = 'vous pouvez me croire, la journée est "pesante".'  
print mot1, mot2
```



C'est une grosse journée; vous pouvez me croire, la journée est "pesante".

L'instruction print insère un espace entre les éléments affichés.

Note : Le caractère spécial « \ » permet d'écrire une commande sur plusieurs lignes.

Il permet d'insérer un certain nombre de caractères spéciaux (saut de ligne, apostrophes, guillemets) à l'intérieur d'une chaîne de caractères.

```
mot = 'C\'est le jour de Pâques.\nBonne \nfin de semaine.'  
print mot
```

C'est le jour de Pâques.
Bonne fin de semaine.

\n saut de ligne
' permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes.

Accès aux caractères d'une chaîne

```
mot = "apprendre" # Le premier caractère est en position 0.  
print mot[3], mot[4], mot[5], mot[6]
```

rend

Concaténation de chaînes à l'aide de l'opérateur +

```
mot = "apprend"  
mot = mot + "re"  
print mot[3] + mot[4] + mot[5] + mot[6]
```

rend

Répétition de chaînes à l'aide de l'opérateur *

```
mot = "cher" * 2  
print mot
```

chercher

Longueur d'une chaîne

```
print len(mot)
```

9

Convertir une chaîne qui représente un nombre en un nombre véritable

```
m = "12.3"  
n = '13'  
print float(m) + int(n)
```

25.3

DR F.

On peut utiliser des apostrophes triples pour protéger des caractères spéciaux.

```
>>> Texte = "Python"  
>>> Texte = Texte + ' est un langage '  
>>> Texte += '"renfermant les guillemets (")."'  
>>> print Texte  
Python est un langage renfermant les guillemets ("  
>>>
```

Convertir un nombre en une chaîne de caractères

 **str()** Convertir un nombre en chaîne de caractères.

```
>>> print "On peut concaténer une chaîne et un nombre converti : " + str(1.14 + 3)  
On peut concaténer une chaîne et un nombre converti : 4.14  
>>>
```


Gestion de la mémoire

- En Python, il n'existe pas de déclaration explicite de variables lesquelles sont implicitement déclarées lors de leur première utilisation.
- Cependant, il n'est pas possible d'accéder à une variable avant qu'elle n'ait été créée et initialisée :

```
>>> a
```

```
Traceback (most recent call last):
```

```
File "<pyshell#0>", line 1, in <module>
```

```
  a
```

```
NameError: name 'a' is not defined
```

```
>>>
```

- En Python, il n'y a pas non plus de spécification explicite de type. Cela se fait implicitement à la première utilisation.

```
X = 3.4
```

→ X est alors une variable de type réel renfermant la valeur 3.4.

- La libération de l'espace mémoire d'une variable est sous la responsabilité de l'interpréteur. Lorsqu'il n'y a plus de références à un espace mémoire, le « ramasse-miettes » se charge de libérer cet espace mémoire.

```
>>> x = 3.4
>>> x = "La variable réelle est perdue."
>>> print x
La variable réelle est perdue.
```

- En temps normal, vous ne « supprimez » pas vraiment un nombre : vous cessez simplement de l'utiliser! Si vous souhaitez supprimer une référence, utilisez l'instruction `del`. Après quoi, on ne peut plus utiliser le nom de variable à moins de l'affecter à une nouvelle valeur.

```
>>> s = 1
>>> print s
1
>>> del s
>>> print s
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print s
NameError: name 's' is not defined
```

```
>>> s = "Ceci est un test."
>>> s
'Ceci est un test.'
>>>
```

Les Listes

Une liste est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets. En voici quelques exemples :

```
1 >>> animaux = ['girafe', 'tigre', 'singé', 'souris']
2 >>> tailles = [5, 2.5, 1.75, 0.15]
3 >>> mixte = ['girafe', 5, 'souris', 0.15]
4 >>> animaux
5 ['girafe', 'tigre', 'singé', 'souris']
6 >>> tailles
7 [5, 2.5, 1.75, 0.15]
8 >>> mixte
9 ['girafe', 5, 'souris', 0.15]
```

Comme les indices des chaînes, les indices des listes commencent à 0, et les listes peuvent être découpées, concaténées... (rapidité!!!!) Les listes sont **modifiables**, on peut donc changer les éléments individuellement: remplacement, destruction,...

Les Listes

Tableau des operations

Opération	Interprétation
L1=[]	liste vide
L2=[0, 1, 2, 3]	4 élément indicé de 0 à 3
L3=['abc', ['def', 'ghi']]	liste incluses
L2[i], L3[i][j]	indice
L2[i:j]	tranche
len(L2)	longueur
L1+L2	concaténation
L1*3	répétition
for x in L2	parcours
3 in L2	appartenance
L2.append(4)	méthodes : agrandissement
L2.sort()	tri
L2.index()	recherche
L2.reverse()	inversion
del L2[k], L2[i:j]=[]	effacement
L2[i]=1	affectation par indice
L2[i:j]=[4, 5, 6]	affectation par tranche
range(4), xrange(0,4)	création de listes / tuples d'entiers

Les Listes

Un exemple

- `>>> a = ['spam', 'eggs', 100, 1234]`
- `>>> a`
- `['spam', 'eggs', 100, 1234]`
- `>>> a[0]`
- `'spam'`
- `>>> a[3]`
- `1234`
- `>>> a[-2]`
- `100`
- `>>> a[1:-1]`
- `['eggs', 100]`
- `>>> a[:2] + ['bacon', 2*2]`
- `['spam', 'eggs', 'bacon', 4]`
- `>>> 3*a[:3] + ['Boe!']`
- `['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']`

Les Listes

Exemple pour les méthodes

Un exemple qui utilise toutes les méthodes des listes:

- `>>> a = [66.6, 333, 333, 1, 1234.5]`
- `>>> print a.count(333), a.count(66.6), a.count('x')`
- `2 1 0`
- `>>> a.insert(2, -1)`
- `>>> a.append(333)`
- `>>> a`
- `[66.6, 333, -1, 333, 1, 1234.5, 333]`
- `>>> a.index(333)`
- `1`
- `>>> a.remove(333)`
- `>>> a`
- `[66.6, -1, 333, 1, 1234.5, 333]`
- `>>> a.reverse()`
- `>>> a`
- `[333, 1234.5, 1, 333, -1, 66.6]`
- `>>> a.sort()`
- `>>> a`
- `[-1, 1, 66.6, 333, 333, 1234.5]`

Les Tuples

Déf. Comme une liste, un tuple est une collection ordonnée d'objets; mais le tuple n'est **pas modifiable** .

Déclaration: des valeurs (entre parenthèses) séparées par des virgules

```
>>>tuple=(0,1.4,'world')
```

Pourquoi les tuples alors que nous avons les listes??

La non-possibilité de modifier les tuples assure une certaine **intégrité** car nous pouvons être sûr qu'un tuple ne sera **pas modifié** à travers une référence ailleurs dans le programme.

Les Tuples

Tableau des opérations

Opération	Interprétation
()	un tuple vide
n1 = (0,)	un tuple à un élément (et non une expression)
n2 = (0,1,2,3)	un tuple à quatre éléments
n2 = 0,1,2,3	un autre tuple à quatre éléments
n3 = ('abc', ('def', 'ghi'))	tuple avec inclusion
t[i], n3[i][j]	indilage
n1[i:j]	tranche
len(n1)	longueur
n1+n2	concaténation
n2 * 3	répétition
for x in n2	itération
3 in s2	test d'appartenance

Les Tuples

Tableau des opérations

Opération	Interprétation
()	un tuple vide
n1 = (0,)	un tuple à un élément (et non une expression)
n2 = (0,1,2,3)	un tuple à quatre éléments
n2 = 0,1,2,3	un autre tuple à quatre éléments
n3 = ('abc', ('def', 'ghi'))	tuple avec inclusion
t[i], n3[i][j]	indilage
n1[i:j]	tranche
len(n1)	longueur
n1+n2	concaténation
n2 * 3	répétition
for x in n2	itération
3 in s2	test d'appartenance

Les Tuples

Tuple unpacking

L'instruction `t = 12345, 54321, 'salut!'` est un exemple d' *emballage en tuple* (tuple packing).

L'opération inverse est aussi possible, par ex.:

- `>>> x, y, z = t`

Ceci est appelé, fort judicieusement, *déballage de tuple* (tuple unpacking).

Le déballage d'un tuple nécessite que la liste des variables à gauche ait un nombre d'éléments égal à la longueur du tuple

A l'occasion, l'opération correspondante sur les listes est utile:

list unpacking.

Ceci est possible en insérant la liste des variables entre des crochets carrés:

- `>>> a = ['spam', 'oeufs', 100, 1234]`
- `>>> [a1, a2, a3, a4] = a`

Les Dictionnaires

Déf. Un dictionnaire est un ensemble non ordonnés de couples clé:valeur avec comme contrainte que les clés soient uniques (dans un même dictionnaire).

Déclaration: des couples clé:valeur séparés par des virgules et entre accolades >>>
dico = {'japon' : 'japan', 'chine' : 'china'}

Un dictionnaire, à la différence des séquences qui sont indexées par un intervalle numérique (cf. liste, chaîne, tuple), est indexé par une clé qui peut être n'importe quel type non-modifiable (les chaînes, les nbs et les tuples s'ils ne contiennent que des éléments non modifiables).

Les Dictionnaires

Tableau des opérations

Opération	Interprétation
<code>d1 = {}</code>	dictionnaire vide
<code>d2={'one': 1, 'two': 2}</code>	dictionnaire à deux éléments
<code>d3={'count': {'one': 1, 'two': 2}}</code>	inclusion
<code>d2['one'], d3['count']['one']</code>	indigage par clé
<code>d2.has_keys('one')</code>	methodes : test d'appartenance
<code>d2.keys()</code>	liste des clés
<code>d2.values()</code>	liste des valeurs
<code>len(d1)</code>	longueur (nombre d'entrée)
<code>d2[cle] = [nouveau]</code>	ajout / modification
<code>del d2[cle]</code>	destruction

Les Dictionnaires

Un exemple

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

Les fichiers

Tableau des opérations

Opération	Interprétation
<code>sortie = open('/tmp/spam', 'w')</code>	crée un fichier de sortie ('w' => écriture)
<code>entre = open('donnee', 'r')</code>	ouvre un fichier en entrée ('r' => lecture)
<code>s = entre.read()</code>	lit le fichier entier dans une chaîne
<code>s = entre.read(N)</code>	lit N octets (1 ou plus)
<code>s = entre.readline()</code>	lit la ligne suivante
<code>L = entre.readlines()</code>	lit le fichier dans une liste de lignes
<code>sortie.write(s)</code>	écrit s dans le fichier
<code>sortie.writelines(L)</code>	écrit toutes les lignes contenues pas L
<code>sortie.close()</code>	fermeture manuelle

STRUCTURES ALGORITHMIQUES

Branchement conditionnel « if »

Condition est très souvent une opération de comparaison



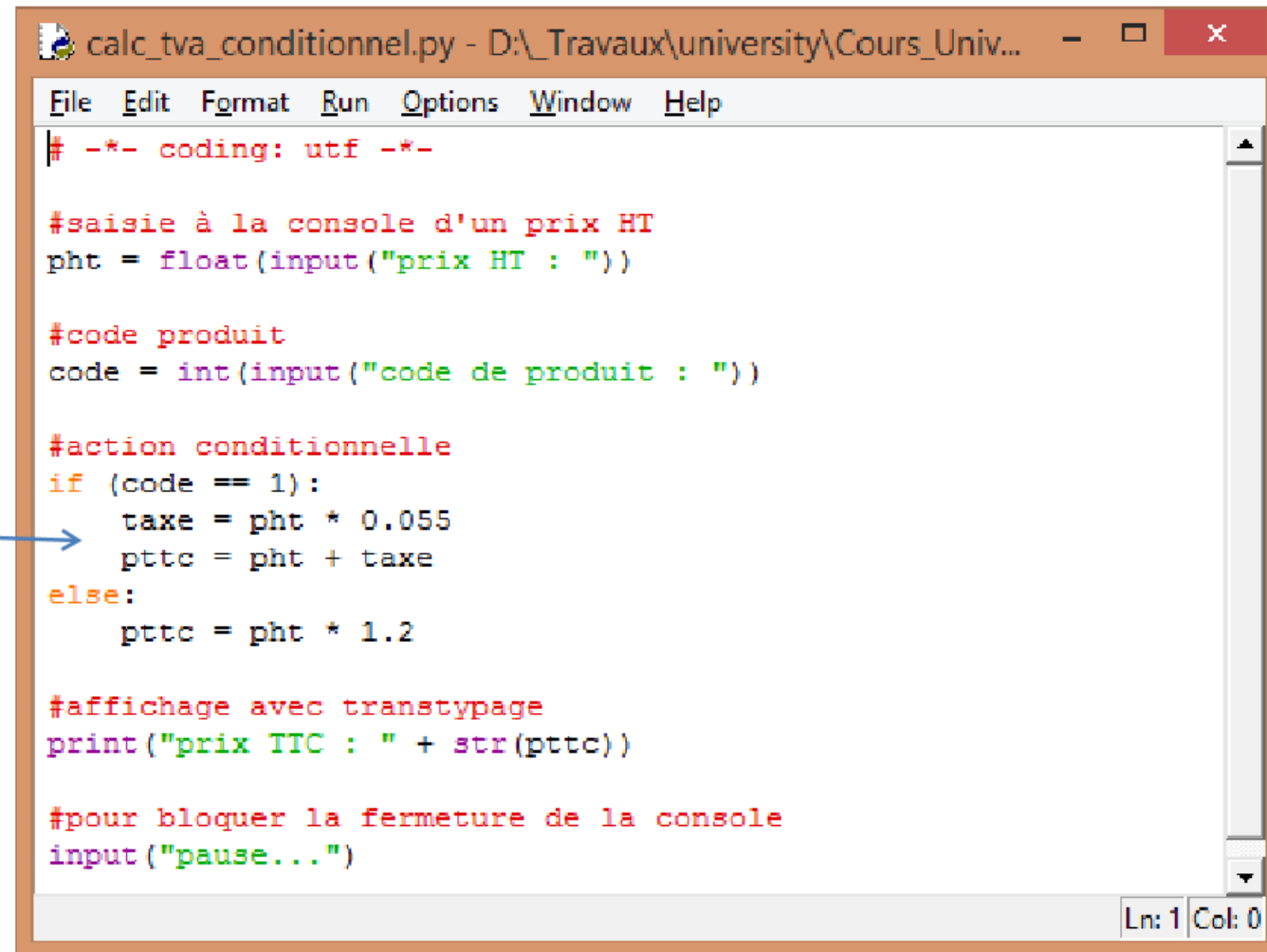
```
if condition:  
    bloc d'instructions  
else:  
    bloc d'instructions
```

- (1) Attention au **:** qui est primordial
- (2) C'est l'**indentation** (le décalage par rapport à la marge gauche) qui délimite le bloc d'instructions
- (3) La partie **else** est facultative

Branchement conditionnel « if » (exemple)

Noter l'imbrication des blocs.

Le code appartenant au même bloc doit être impérativement aligné sinon erreur.



```
calc_tva_conditionnel.py - D:\Travaux\university\Cours_Univ...
File Edit Format Run Options Window Help

# -*- coding: utf -*-

#saisie à la console d'un prix HT
pht = float(input("prix HT : "))

#code produit
code = int(input("code de produit : "))

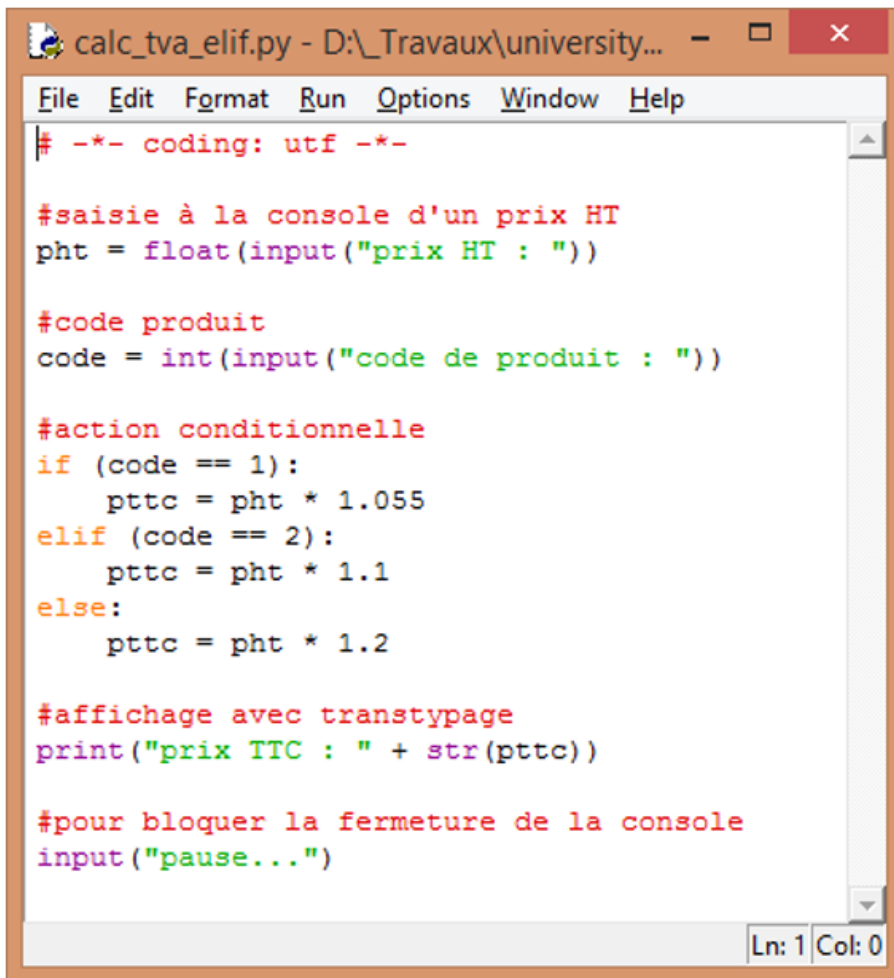
#action conditionnelle
if (code == 1):
    taxe = pht * 0.055
    pttc = pht + taxe
else:
    pttc = pht * 1.2

#affichage avec transtypage
print("prix TTC : " + str(pttc))

#pour bloquer la fermeture de la console
input("pause...")

Ln: 1 Col: 0
```


Succession de if avec **elif**



```
calc_tva_elif.py - D:\Travaux\university...  
File Edit Format Run Options Window Help  
# -*- coding: utf -*-  
  
#saisie à la console d'un prix HT  
pht = float(input("prix HT : "))  
  
#code produit  
code = int(input("code de produit : "))  
  
#action conditionnelle  
if (code == 1):  
    pttc = pht * 1.055  
elif (code == 2):  
    pttc = pht * 1.1  
else:  
    pttc = pht * 1.2  
  
#affichage avec transtypage  
print("prix TTC : " + str(pttc))  
  
#pour bloquer la fermeture de la console  
input("pause...")  
Ln: 1 Col: 0
```

- **elif** n'est déclenché que si la (les) condition(s) précédente(s) a (ont) échoué.
- **elif** est situé au même niveau que **if** et **else**
- On peut en mettre autant que l'on veut



Il n'y a pas de **switch()** ou de **case...of** en Python

Principe de la boucle for

Elle ne s'applique que sur une collection de valeurs. Ex. tuples, listes,... à voir plus tard.

Suite arithmétique simple (séquence de valeurs entières)


On peut définir des boucles indicées en générant une collection de valeurs avec `range()`

(1) `range(4)` → 0 1 2 3

(2) `range(1, 4)` → 1 2 3

(3) `range(0, 5, 2)` → 0 2 4

Séquence est une collection de valeurs
Peut être générée avec range()



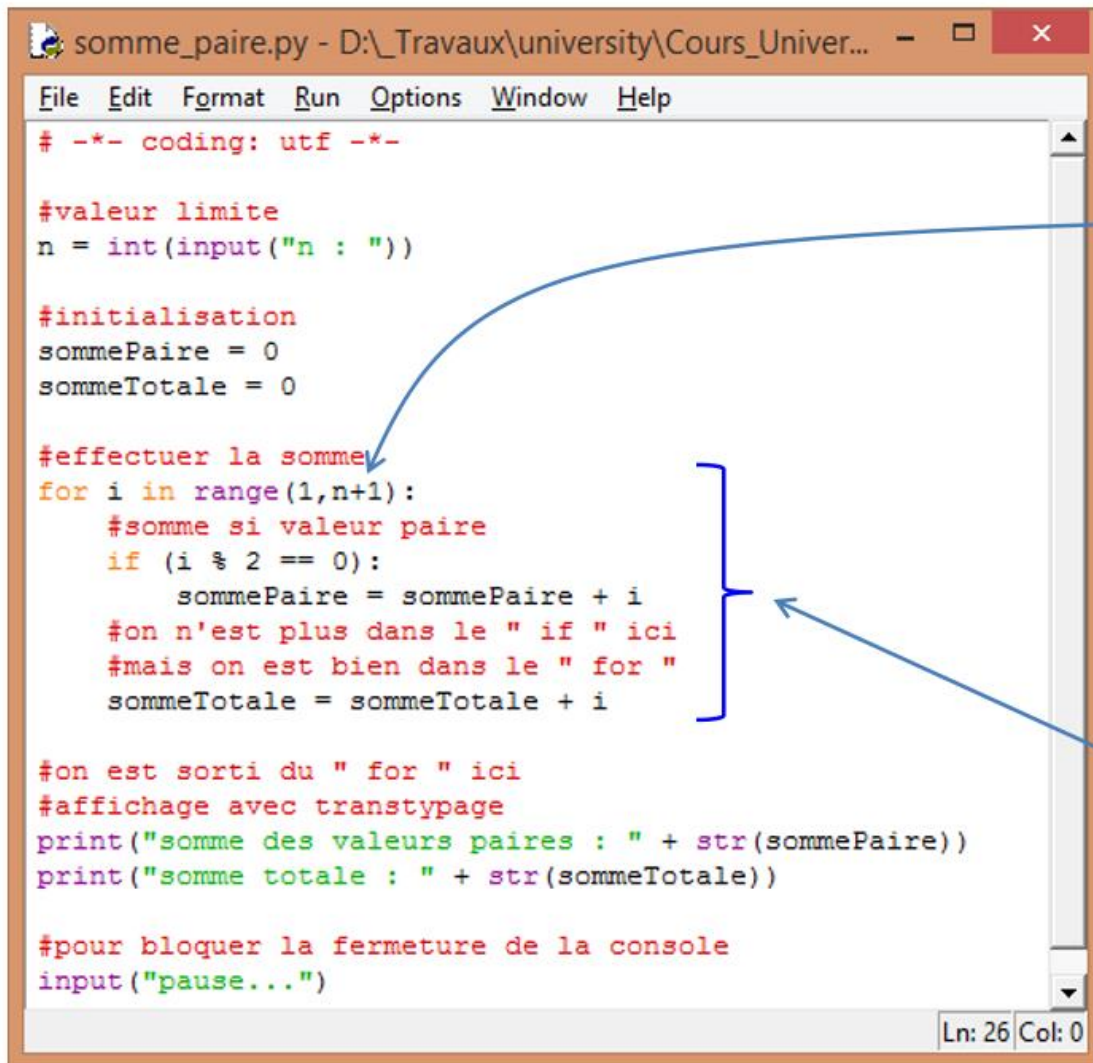
```
for indice in séquence:  
    bloc d'instructions
```

Remarque :

- Attention à l'indentation toujours
- On peut « casser » la boucle avec **break**
- On peut passer directement à l'itération suivante avec **continue**
- Des boucles imbriquées sont possibles
- Le bloc d'instructions peut contenir des conditions

Boucle « for » (exemple)

Somme totale des valeurs comprises entre 1 et **n** (inclus) et somme des valeurs paires dans le même intervalle



```

somme_paire.py - D:\Travaux\university\Cours_Univer...
File Edit Format Run Options Window Help
# -*- coding: utf -*-

#valeur limite
n = int(input("n : "))

#initialisation
sommePaire = 0
sommeTotale = 0

#effectuer la somme
for i in range(1,n+1):
    #somme si valeur paire
    if (i % 2 == 0):
        sommePaire = sommePaire + i
    #on n'est plus dans le " if " ici
    #mais on est bien dans le " for "
    sommeTotale = sommeTotale + i

#on est sorti du " for " ici
#affichage avec transtypage
print("somme des valeurs paires : " + str(sommePaire))
print("somme totale : " + str(sommeTotale))

#pour bloquer la fermeture de la console
input("pause...")
Ln: 26 Col: 0

```

The image shows a Python script editor window titled 'somme_paire.py'. The script calculates the sum of all integers from 1 to n and the sum of even integers in that range. A blue arrow points from the text 'Il faut mettre n+1 dans range()' to the 'n+1' in the 'range(1,n+1)' line. Another blue arrow points from the text 'Observez attentivement les indentations.' to a blue bracket that groups the lines inside the 'for' loop, highlighting the indentation levels.

Il faut mettre **n+1** dans `range()` pour que **n** soit inclus dans la somme

Observez attentivement les indentations.

Opération de comparaison
Attention à la boucle infinie !

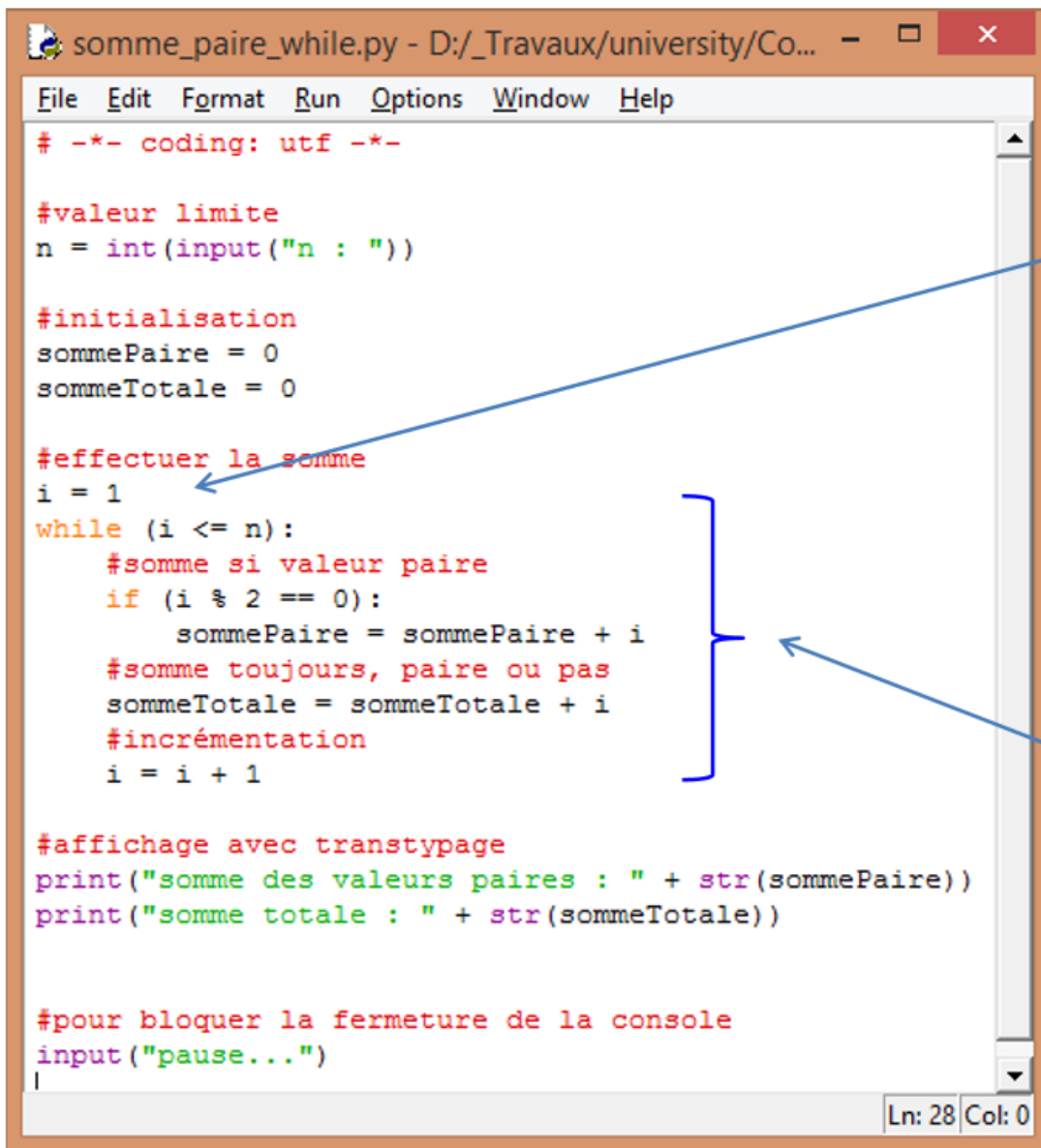


```
while condition:  
    bloc d'instructions
```

Remarque :

- Attention à l'indentation toujours
- On peut « casser » la boucle avec **break**

Boucle « while » (exemple)



```
File Edit Format Run Options Window Help
# -*- coding: utf -*-

#valeur limite
n = int(input("n : "))

#initialisation
sommePaire = 0
sommeTotale = 0

#effectuer la somme
i = 1
while (i <= n):
    #somme si valeur paire
    if (i % 2 == 0):
        sommePaire = sommePaire + i
    #somme toujours, paire ou pas
    sommeTotale = sommeTotale + i
    #incréméntation
    i = i + 1

#affichage avec transtypage
print("somme des valeurs paires : " + str(sommePaire))
print("somme totale : " + str(sommeTotale))

#pour bloquer la fermeture de la console
input("pause...")
|
```

Ln: 28 Col: 0

Ne pas oublier
l'initialisation de **i**

Observez attentivement
les indentations.