Un programme effectue des opérations sur des nombres ou des caractères afin de générer un résultat. Ces nombres ou ces caractères sont stockés dans des variables. Les opérations sur ces nombres sont effectuées par l'intermédiaire d'opérateurs. Ils caractérisent le type d'opération que nous désirons exécuter sur ces données.

Affectation et initialisation

Cet opérateur est l'opérateur fondamental de tout langage de programmation, en effet il permet d'affecter (d'assigner, d'imposer) le contenu d'une variable (qui peut être une valeur précise, le contenu d'une autre variable, ou même tout un calcul préliminaire avant de récupérer la valeur du traitement associé). Dans ce registre, il est possible de proposer des affectations multiples (enchaînées). En Python, nous utilisons le symbole **« = »**.

Cet opérateur est également utilisé pour la déclaration d'une variable, puisque je le rappelle, pour qu'une variable soit réellement déclarée, elle doit être obligatoirement initialisée. Contrairement à d'autres langages, Python ne fait pas la différence entre la phase d'initialisation et l'affectation classique puisqu'une variable peut changer de type à tout moment (comportement à éviter).

*Attention : « **=** » ne correspond pas à une égalité, mais bien à une affectation (la variable prend une nouvelle valeur).*

Exemples

```
>>> a = 5 # déclaration d'une variable entière (initialisation)
>>> b = 3.5 # déclaration d'une variable réelle (initialisation)
>>> c, d = 4, 5.6 # combinaisons de deux déclarations
>>> e = f = b # affectations multiples (initialisations multiples)
>>> a, c = c, a # permutation des valeurs entre les deux variables
>>> b, d = b+2, d+2*b # traitements multiples avant affectations
>>> print(a, b, c, d, e, f)
4 5.5 5 12.6 3.5 3.5
```

*L'opérateur virgule **« , »** est un opérateur comme les autres, il permet de réaliser une séquence de traitements successifs.*

Opérateurs arithmétiques

Ils permettent d'effectuer les opérations dites mathématiques. Ces opérations ne s'appliquent qu'a des valeurs ou des variables de type numériques (entières ou réelles).

Opérateurs arithmétiques	Opération effectuée
+	Addition
-	Soustraction
*	Multiplication
/	Division réelle
//	Division entière
%	Modulo (reste d'une division entière)
**	Exponentiation (puissance)

Exemples

```
>>> a, b = 5, 3 # déclaration de deux variables entières
>>> a+b # addition
8
>>> a-b # soustraction
2
>>> a*b # multiplication
15
>>> a/b # division réelle
1.6666666666666667
>>> a//b # division entière
1
>>> a%b # reste de la division entière
2
>>> a**b # exponentiation
125
```

Opérateurs d'égalité, relationnels et logiques

Les opérateurs relationnels permettent de comparer des valeurs relatives. Ceux sont des opérateurs dits de test, ils sont donc généralement utilisés pour les structures de tests, ou les structures de boucle.

*Le résultat de ces opérateurs est de type booléen, donc : soit **Vrai**, soit **Faux**.*
***Faux :** représentée par la constante littérale : **False**.*
***Vrai :** représentée par la constante littérale : **True**.*

Opérateurs relationnels	Opération effectuée
>	Supérieur à
>=	Supérieur ou égal à
<	Inférieur à
<=	Inférieur ou égal à
==	Égal à
!=	Différent de, ou non égal à
and	ET logique
or	OU logique
not	NON logique

Exemples

```
>>> a, b, c = 65, 200, -98 # déclaration de trois variables entières
>>> a > b, c >= b, c < a
(False, False, True)
>>> c <= -98, a==0, a==65
(True, False, True)
>>> b!=c, a != b == c, a==65 and b==200
(True, False, True)
>>> a==65 and b==2000, a==65 or b==2000, a==7 or b==0
(False, True, False)
>>> a==65 or not b!=200, a==7 or not b!=200, a==7 or not b!=0
(True, True, False)
>>> a < b < c, c < a < b
(False, True)
```

Opérateurs de bit (spécialisés pour le traitement binaire)

Le langage Python fourni cinq opérateurs qui permettent de manipuler les opérations au niveau de chaque bit

Opérateurs de bit	Opération effectuée
&	ET bit à bit
^	OU exclusif bit à bit
|	OU bit à bit
variable << décalage	Décalage à gauche
variable >> décalage	Décalage à droite
~	Complément (NON) bit à bit

Exemples

```
>>> h = 0b1010 # possibilité de créer une variable entière à l'aide d'un littéral binaire (0b)
>>> h
10
>>> a, b, c = 65, 200, -98 # déclaration de trois variables entières
>>> bin(a), bin(b), bin(c) # retrouver le nombre binaire correspondant au nombre entier, fonction bin()
('0b1000001', '0b11001000', '-0b1100010')
>>> a & b, bin(a & b)
(64, '0b1000000')
>>> a | b, bin(a | b)
(201, '0b11001001')
>>> a ^ b, bin(a ^ b)
(137, '0b10001001')
>>> ~a, bin(~a)
(-66, '-0b1000010')
>>> ~c, bin(~c)
(97, '0b1100001')
>>> b << 1, bin(b << 1) # décalage à gauche d'une unité (multiplication par 2)
(400, '0b110010000')
>>> a >> 2, bin(a >> 2) # décalage à droite de deux unités (division entière par 4)
(16, '0b10000')
```

Opérateurs d'affectations composés

Il existe 12 opérateurs d'affectation. L'opérateur **« = »** est le plus simple d'entre eux; les autres sont les opérateurs d'affectation composés. C'est une combinaison entre l'opérateur d'assignation et un autre des opérateurs déjà vus.

Opérateurs d'affectation composés	Opération effectuée
Expression1 = Expression2	Expression1 = Expression 2
Expression1 *= Expression2	Expression1 = Expression1 * Expression2
Expression1 /= Expression2	Expression1 = Expression1 / Expression2
Expression1 //= Expression2	Expression1 = Expression1 // Expression2
Expression1 %= Expression2	Expression1 = Expression1 % Expression2
Expression1 += Expression2	Expression1 = Expression1 + Expression2
Expression1 -= Expression2	Expression1 = Expression1 - Expression2
Expression1 <<= Expression2	Expression1 = Expression1 << Expression2
Expression1 >>= Expression2	Expression1 = Expression1 >> Expression2
Expression1 &= Expression2	Expression1 = Expression1 & Expression2
Expression1 ^= Expression2	Expression1 = Expression1 ^ Expression2
Expression1 |= Expression2	Expression1 = Expression1 | Expression2

Syntaxe classique	Avec l'affectation composée
i = i + 5	i += 5
j = j – 2	j -= 2
k = k * 3	k *= 3
x = y / 5	Rien ne change, les variables sont différentes
i += 1	Incrémentation
i -= 1	Décrémentation

Sélection ou exécution conditionnelle

Si nous désirons développer des applications vraiment utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour cela, nous devons disposer d'instructions capables de tester certaines conditions et de modifier le comportement du programme en conséquence.

*Le langage python propose trois mots réservés associés à tous les comportements conditionnels qui réclament une décision adaptée. Comme pour les principaux langages, nous disposons de **« if »** et de **« else »**, et le langage python propose également le **« elif »** (contraction de **« else if »**).*

Par contre, le langage python ne dispose pas de sélections multiples à l'image du « switch » de la plupart des autres langages de programmation.

*Les conditions de traitement s'expriment à l'aide des opérateurs relationnels que nous venons d'étudier, comme par exemple l'opérateur supérieur **« > »**, l'opérateur inférieur **« < »**, l'opérateur d'égalité **« == »**, etc.*

equation.py

```
""" (début d'un bloc de commentaires)
Résolution de l'équation du premier degré ax+b = 0
avec a ≠ 0
""" (fin d'un bloc de commentaires)

# Récupérer les valeurs des paramètres de l'équation (commentaire de ligne)

print("Résolution de l'équation du premier degré : ax+b=0")
a = float(input("Introduisez la valeur de a : "))
b = float(input("Introduisez la valeur de b : "))

# calcul effectué uniquement si a ≠ 0  (commentaire de ligne)
if a!=0 : print("La solution est x =", -b/a)
```

Shell Python : [évaluer equations.py]

```
Résolution de l'équation du premier degré : ax+b=0
Introduisez la valeur de a : 1
Introduisez la valeur de b : -1
La solution est x = 1.0
```

*Ce code est un script correspondant à la résolution d'une équation du premier degré, comme cela est évoqué dans le commentaire de bloc au début du fichier. Vous pouvez documenter tous vos codes sources afin de mieux comprendre leurs constitutions. Deux types de commentaire existent : le commentaire sur plusieurs lignes (délimité par des triples guillemets), le commentaire de ligne qui se termine automatiquement à la fin de la ligne (débute par le caractère **« # »**).*

*La plupart des scripts élaborés nécessitent à un moment ou à un autre une interaction avec l'utilisateur afin de pouvoir saisir les données indispensables pour le traitement du programme, notamment ici pour récupérer les valeurs des paramètres **« a »** et **« b »** pour la résolution de cette équation du premier degré. Nous connaissons déjà la fonction **« print() »** qui permet d'afficher en sortie le résultat d'une évaluation ou d'un simple texte. Il existe aussi la fonction intégrée **« input() »** qui réalise l'opération inverse, en entrée, qui permet de récupérer les valeurs saisies par l'utilisateur.*

*Ainsi, cette fonction **« input() »** provoque une interruption du programme courant et attend les caractères saisies au clavier par l'utilisateur validées par la touche **« Entrée »**. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction retourne systématiquement une chaîne de caractères correspondant à ce que l'utilisateur a saisi (même si l'utilisateur n'a utilisé que le clavier numérique).*

*Si la saisie correspond uniquement à des valeurs numériques, vous devez alors transformer cette chaîne vers le type correspondant, en utilisant des fonctions intégrées adaptées au type de traitement : la fonction **« int() »** pour une valeur entière, la fonction **« float() »** pour une valeur réelle, la fonction **« bool() »** pour une valeur booléenne, etc.*

*L'instruction conditionnelle débute systématiquement par **« if » (si)**, suivie de la condition elle-même **« a!=0 »** (résultat toujours booléen), ponctué impérativement par le symbole **« : »**. À la suite se trouve l'instruction qui sera exécutée uniquement dans le cas où la condition est validée.*

*Si vous devez exécuter plusieurs instructions pour une même condition, cela s'appelle un **« bloc d'instructions »** qui dans beaucoup de langages sont délimités par des accolades. Dans le cas de Python, il suffit d'utiliser ce que nous appelons l'indentation, c'est-à-dire l'utilisation de la tabulation pour toutes les instructions associées à cette condition. Python utilise toujours cette même technique pour tout ce qui est **« bloc d'instructions »**, notamment pour les boucles, pour les définitions de fonctions, pour les déclarations de classes, etc.*

equation.py

```
# Récupérer les valeurs des paramètres de l'équation
print("Résolution de l'équation du premier degré : ax+b=0")
a = float(input("Introduisez la valeur de a : "))
b = float(input("Introduisez la valeur de b : "))

# calcul effectué uniquement si a est différent de 0
if a!=0 :
    x = -b/a # bloc d'instructions
    print("La solution est x =", x)
```

*Il est bien entendu possible de réaliser un traitement dans le cas où la condition n'est pas validée grâce au mot réservé **« else » (sinon)**. Par exemple, nous pouvons spécifier que si a est égal à 0 alors l'équation n'a pas de solution :*

equation.py

```
# Récupérer les valeurs des paramètres de l'équation
print("Résolution de l'équation du premier degré : ax+b=0")
a = float(input("Introduisez la valeur de a : "))
b = float(input("Introduisez la valeur de b : "))

# calcul effectué uniquement si a ≠ 0 sinon afficher « Pas de solution »
if a!=0 : print("La solution est x =", -b/a)
else : print("Pas de solution")
```

Shell Python : [évaluer equations.py]

```
Résolution de l'équation du premier degré : ax+b=0
Introduisez la valeur de a : 0
Introduisez la valeur de b : -5
Pas de solution
```

*Quelquefois, nous avons besoin de proposer plusieurs alternatives à un problème avec des conditions multiples. Il faut alors un enchaînement de plusieurs **« if »** dans le cas où les premières conditions ne sont pas validées. La suite logique, après le premier « **if** » serait donc un enchaînement de **« else if »**. Python propose une contraction de cette dernière instruction avec le mot réservé **« elif »**.Voici un exemple ci-dessous*

equation.py

```
# Résolution de l'équation du deuxième degré ax^2 + bx + c = 0

# récupérer la fonction racine carrée (square root) de la bibliothèque mathématique
from math import sqrt

# Récupérer les valeurs des paramètres de l'équation
print("Résolution de l'équation du deuxième degré : ax^2 + bx + c =0")
a = float(input("Introduisez la valeur de a : "))
b = float(input("Introduisez la valeur de b : "))
c = float(input("Introduisez la valeur de c : "))


# Recherche des solutions et affichages
if a==0 : print("Ce n'est pas une équation du second degré")
else :
    delta = b**2 - 4*a*c
    if   delta<0  : print("Pas de solution")
    elif delta==0 : print("x =", -b/(2*a))
    else : print("x1 =", (-b+sqrt(delta))/(2*a), ", x2 =", (-b-sqrt(delta))/(2*a))
```

Shell Python : [évaluer equations.py]

```
Résolution de l'équation du deuxième degré : ax^2 + bx + c =0
Introduisez la valeur de a : 0
Introduisez la valeur de b : 1
Introduisez la valeur de c : 1
Ce n'est pas une équation du second degré
```

Shell Python : [évaluer equations.py]

```
Résolution de l'équation du deuxième degré : ax^2 + bx + c =0
Introduisez la valeur de a : 1
Introduisez la valeur de b : 1
Introduisez la valeur de c : 1
Pas de solution
```

Shell Python : [évaluer equations.py]

```
Résolution de l'équation du deuxième degré : ax^2 + bx + c =0
Introduisez la valeur de a : 1
Introduisez la valeur de b : -2
Introduisez la valeur de c : 1
x = 1.0
```

Shell Python : [évaluer equations.py]

```
Résolution de l'équation du deuxième degré : ax^2 + bx + c =0
Introduisez la valeur de a : 1
Introduisez la valeur de b : -3
Introduisez la valeur de c : 2
x1 = 2.0 , x2 = 1.0
```

Python est très riche et possède beaucoup de fonctions préprogrammées. Plutôt que d'encombrer inutilement la mémoire centrale avec l'ensemble des possibilités qu'offre Python, ces fonctions sont regroupées dans des modules (fichiers) apparentés, comme par exemple la bibliothèque mathématique qui possède tout un tas de fonctions classiques bien utiles, telles que les fonctions sinus, cosinus, racine carrée, etc.

*Il est possible de récupérer une de ces fonctions dans un des modules au moyen de la commande **« import »**. Vous devez alors préciser le module concerné avec **« from »** et la fonction qui vous intéresse. Si vous désirez incorporer plusieurs fonctions du même module, vous pouvez alors utiliser la jocker **« * »** (signifiant toutes les fonctions du module concerné).*

```
# récupérer toutes les fonctions de la bibliothèque mathématique
from math import *
```

Itératives - les instructions répétitives - while

Il arrive très souvent que vous ayez besoin de répéter un certain nombre d'instructions plusieurs fois d'affilée avant de poursuivre votre programme principal, souvent avec un ensemble de valeurs bien précises et suivant des conditions prédéterminées. De façon plus prosaïque, le terme utilisé est souvent le terme **« boucle »** (tourner en boucle). Les professionnels de l'informatique utilise plutôt le terme de **« itérative »**.

*En algorithmique, il existe trois types d'itératives: **« répéter jusqu'à »**, **« tant que »** et **« pour »**. Python fait toujours au plus simple, l'itérative de type **« répéter jusqu'à »** n'existe pas, seule **« tant que »** est implémentée qui est représentée par le mot réservé **« while »**.*

*Nous le verrons dans le chapitre suivant, l'itérative **« pour »** est également implémentée, mais uniquement pour gérer une collection complète d'éléments de mêmes natures (séquence), comme les chaînes de caractères ou les suites. Dans ce cadre là, il s'agit plutôt d'une itérative de type **« foreach »** qu'utilisent certains langages de programmation.*

factorielle.py

```
# Calcul de la factorielle d'un nombre

# Interaction avec l'utilisateur
print("Calcul de la factorielle d'un nombre : n!")
n = int(input("Introduisez la valeur de n : "))

# Calcul de la factorielle
i, calcul = 1, 1
while i <= n : calcul, i = calcul*i, i+1

# Affichage du résultat
print(n, "! = ", calcul, sep='')
```

Shell Python : [évaluer factorielle.py]

```
Calcul de la factorielle d'un nombre : n!
Introduisez la valeur de n : 42
42! = 1405006117752879898543142606244511569936384000000000
```

*L'itérative **« tant que »** est strictement identique à la structure conditionnelle que nous avons abordé au chapitre précédent. Ici, la boucle débute systématiquement par **« while »**, suivie de la condition elle-même **« i<=n »** (résultat toujours booléen), ponctué impérativement par le symbole **« : »**. À la suite se trouve l'instruction (ou les instructions) qui sera exécutée uniquement dans le cas où la condition est validée, ici tant que la variable **« i »** n'atteint pas la valeur de **« n »**.*

*Vous remarquez que l'instruction proposée dans la boucle est extrêmement concise grâce à l'utilisation de l'opérateur **« = »**.*

En consultant le résultat de ce petit programme, vous remarquez également qu'avec Python vous pouvez effectuer des calculs impliquant des valeurs entières comportant un nombre de chiffres significatifs quelconque. Ce nombre n'est limité que par la taille de la mémoire actuellement disponible. Il va de soit cependant que les calculs impliquant de très grands nombres devront être décomposés par l'interpréteur en calculs multiples sur des nombres plus simples, ce qui pourra nécessiter un temps de traitement considérable dans certains cas.

Interrompre une boucle prématurément – break

Même si cela doit être exceptionnel, il est possible d'interrompre le déroulement d'une itérative sous certaines conditions particulières. Pour cela vous disposez du mot réservé **« break »**.

premier.py

```
# Recherche si un nombre est premier
from math import  sqrt

print("Recherche si un nombre entier est premier")
nombre = int(input("Introduisez la valeur du nombre à tester : "))

i, premier = 2, True
while i <= sqrt(nombre) :
    if nombre%i == 0 :
        premier = False
        break # Il existe au moins un diviseur
    i+=1

if premier : print("Ce nombre est premier")
else : print("Ce nombre comporte des diviseurs")
```

Shell Python : [évaluer premier.py]

```
Recherche si un nombre entier est premier
Introduisez la valeur du nombre à tester : 31
Ce nombre est premier
```

Shell Python : [évaluer premier.py]

```
Recherche si un nombre entier est premier
Introduisez la valeur du nombre à tester : 125
Ce nombre comporte des diviseurs
```

*Dans cet exemple, l'instruction **« break »** est intéressante et particulièrement judicieuse. Elle nous évite de faire des calculs qui ne servent plus à rien, lorsque notamment le nombre possède au moins un diviseur.*

Parcours d'un ensemble d'éléments (d'une séquence)

Il arrive très souvent que nous ayons besoin de traiter l'intégralité d'un ensemble d'éléments dont les cases sont consécutives en mémoire (séquence), comme une suite de caractères (chaîne de caractères) ou comme une suite d'autres types d'éléments représentés dans Python par les suites.

*Le parcours d'une séquence permet de traiter l'intégralité des éléments, du premier jusqu'au dernier, pour réaliser une ou plusieurs opérations spécifiques. Nous pouvons le faire avec l'itérative **« while »** que nous venons de découvrir, mais il est plus judicieux d'utiliser le deuxième type d'itérative qui est justement prévu à cet effet. Il s'agit du couple d'instructions : **« for ... in ... »***

chaine.py

```
# Récupérer tous les caractères d'un texte
print("Récupérer chaque caractère d'un texte")
texte = input("Saisissez votre texte : ")
for caractère in texte : print(caractère, '.', sep='', end='')
```

Shell Python : [évaluer chaine.py]

```
Récupérer chaque caractère d'un texte
Saisissez votre texte : Bienvenue à tous
B.i.e.n.v.e.n.u.e. .à. .t.o.u.s.
```

*Comme vous pouvez le constater, cette structure de boucle est très compacte. Elle nous évite d'avoir à définir et à incrémenter une variable spécifique (un compteur, nécessaire dans le cas du **« while »**) pour gérer l'indice du caractère que vous voulez traiter à chaque itération (c'est Python qui s'en charge).*

*La structure **« for ... in ... »** ne montre que l'essentiel, à savoir que la variable **« caractère »** contiendra successivement tous les caractères de la chaîne **« texte »**, du premier jusqu'au dernier. L'instruction **« for »** permet donc d'écrire des boucles, dans lesquelles l'itération traite successivement tous les éléments d'une séquence donnée.*

*La fonction **« print() »** dispose de deux arguments particuliers qui nous permettent d'adapter l'affichage du résultat suivant notre convenance. Par défaut, après l'appel à cette fonction, le curseur d'affichage se place systématiquement à la ligne suivante. De plus, lorsque nous affichons plusieurs éléments consécutifs, ils sont placés sur la même ligne avec un espace séparateur.*

*Il est possible de modifier ce comportement par défaut grâce, respectivement, aux deux arguments **« end »** et **« sep »** dont les termes sont assez évocateurs. Dans l'exemple traité, je demande à ne plus avoir un retour à la ligne systématique, donc d'afficher la suite de caractères les uns à coté des autres et de plus sans espace séparateur.*

Parcours d'une séquence avec le type « range » et la boucle for

Le type **« range »** est une classe (sujet traité lors de la prochaine étude) qui représente une séquence immuable de nombres qui est couramment utilisé pour réaliser des itérations en corrélation avec la boucle **« for »**. Cette classe possède deux constructions possibles dont voici les possibilités :

- ***range(stop)***
- ***range(start, stop, step=1)***

*Si vous appelez le constructeur **« range() »** avec un seul argument, il génère par défaut une séquence de nombres entiers de valeurs croissantes en commençant par **« 0 »**. Le nombre de valeurs générées correspond au nombre proposé par l'argument, sachant que l'argument fourni n'est jamais dans la liste générée, puisque nous commençons par la valeur **« 0 » (n-1)**.*

*Vous pouvez utiliser le deuxième constructeur, qui vous permet alors de choisir la valeur de départ (plutôt que la valeur **« 0 »**) de votre séquence de nombres et aussi de choisir éventuellement le pas d'incrémentation (**« 1 »** par défaut) qui peut d'ailleurs être négatif.*

Shell Python :

```
>>> for valeur in range(10) : print(valeur, end=' ')
0 1 2 3 4 5 6 7 8 9

>>> for valeur in range(1,11) : print(valeur, end=' ')
1 2 3 4 5 6 7 8 9 10

>>> for valeur in range(5,13) : print(valeur, end=' ')
5 6 7 8 9 10 11 12

>>> for valeur in range(0, 18, 3) : print(valeur, end=' ')
0 3 6 9 12 15

>>> for valeur in range(-1, -10, -1) : print(valeur, end=' ')
-1 -2 -3 -4 -5 -6 -7 -8 -9

>>> for valeur in range(5, -6, -1) : print(valeur, end=' ')
5 4 3 2 1 0 -1 -2 -3 -4 -5
```

*Grâce à cette classe **« range »** et à la boucle de type **« for ... in ... »**, vous disposez de plein d'opportunité pour construire toutes les itératives qui vous intéressent, sachant que le cas le plus fréquent reste la première solution. Je vous propose d'ailleurs de revoir le codage du calcul de la factorielle d'un nombre en prenant l'itérative de type **« for...in... »** plutôt que l'itérative de type **« while »**.*

factorielle.py

```
# Interaction avec l'utilisateur
print("Calcul de la factorielle d'un nombre : n!")
n = int(input("Introduisez la valeur de n : "))

# Calcul de la factorielle
calcul = 1
for i in range(2, n+1): calcul *= i

# Affichage du résultat
print(n, "! = ", calcul, sep='')
```

Shell Python : [évaluer factorielle.py]

```
Calcul de la factorielle d'un nombre : n!
Introduisez la valeur de n : 42
42! = 1405006117752879898543142606244511569936384000000000
```

*Par rapport à la boucle **« while »**, le script est beaucoup plus réduit et peut-être plus intuitif avec cette boucle **« for...in... »** et la classe **« range »**.*

*Toutefois, il faut bien comprendre que nous utilisons un type de données **« range »** qui est relativement sophistiqué et qui prend pas mal de place en mémoire puisqu'il génère une séquence. Le temps est également plus conséquent. L'interpréteur est beaucoup plus véloce avec la manipulation de simples entiers dans le script qui utilise la boucle **« while »**.*

Appartenance d'un élément à une séquence – in

L'instruction **« in »** peut très bien être utilisée indépendamment de **« for »**, pour vérifier si un élément fait bien partie ou non d'une séquence. Cette instruction **« in »** est alors toujours associée à une sélection ou une exécution conditionnelle, **« if »** ou **« elif »**.

chaine.py

```
# Décomposition en voyelles et consonnes
print("Connaître le nombre de consonnes et de voyelles dans un texte")
texte = input("Saisissez votre texte : ")

voyelles = 'aeiouyAEIOUYàâéèêëùîï'
chiffres = '0123456789'
nbVoyelle = nbConsonne = nbChiffre = nbAutre = 0

for caractère in texte :
    if caractère in voyelles : nbVoyelle += 1
    elif caractère in chiffres : nbChiffre += 1
    elif 'A' < caractère <= 'Z' or 'a' < caractère <= 'z' : nbConsonne += 1
    else : nbAutre += 1

print(nbVoyelle, "voyelles,", nbConsonne, "consonnes,", nbChiffre, "chiffres,", nbAutre, "autres")
```

Shell Python : [évaluer chaine.py]

```
Connaître le nombre de consonnes et de voyelles dans un texte
Saisissez votre texte : 55, rue de l'informatique
9 voyelles, 9 consonnes, 2 chiffres, 5 autres
```

*Dans cet exemple, nous nous servons de **« in »** pour vérifier le type d'appartenance pour chacun des caractères composant un texte.*