

Chapitre :

Types et valeurs



I. Entiers naturels

1. Codage binaire

Une machine électrique fonctionne si elle a du courant, ne fonctionne pas si elle n'en a pas. Ainsi fait l'ampoule : elle s'allume avec du courant et reste éteinte sans (suffisamment de) courant. Ces deux états « courant » et « pas de courant » peuvent être notés respectivement 1 et 0. On obtient ce que l'on appelle un codage binaire. C'est sur ce codage que fonctionnent les ordinateurs, qui sont capables de manipuler des informations sous cette forme. Manipuler signifie entre autres recevoir les informations et en envoyer.

L'alphabet de l'ordinateur n'est donc composé que de deux lettres : 0 et 1.

Une telle lettre est appelée **bit**, mot qui est la contraction de **binary digit** (chiffre binaire).

La machine ne traite généralement pas qu'un bit à la fois, mais des paquets de bits.

Un paquet de huit bits est appelé un **octet**.

Exemple Voici un octet (dans le langage binaire) : 10011101.

On peut voir un octet comme un nombre, mais attention ! L'écriture est trompeuse. Ce sont des nombres écrits en **binaire**, en mathématiques on dit en **base 2**. Pour éviter l'ambiguïté on écrit $(10011101)_2$.

Notre notation habituelle des nombres est un codage en base 10 : nous utilisons dix symboles de chiffres pour écrire les nombres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Pour coder en base 2 nous n'allons utiliser que deux symboles : 0 et 1. Pour compter, le principe est le même qu'en base 10, sauf qu'il ne faut pas oublier que l'on ne dépasse pas le chiffre 1.

Comptons en binaire : 0, 1, 10, 11, 100, 101,...

Rappelons que tout nombre (en base 10) peut être décomposé à l'aide de puissances de 10 :

Exemple $(562)_{10} = 5 \times 100 + 6 \times 10 + 2 \times 1 = 5 \times 10^2 + 6 \times 10^1 + 2 \times 10^0$.

Remarque On peut observer que :

- la puissance la plus petite (qui se trouve pour le chiffre le plus à droite) est 0 ;
- pour un nombre de 3 (resp. n) chiffres, la puissance (de dix) la plus grande est 2 (resp. $n - 1$).

Il y a donc une sorte de décalage auquel il faut faire attention.

Un nombre en base 2 est, lui, décomposé (en base 10!) à l'aide de puissances de 2 sur le même principe :

Exemple $(101)_2 = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$.

En calculant le nombre de droite (écrit en base 10 rappelons-le), on obtient : $4 + 1 = 5$.

On a donc bien : $(101)_2 = (5)_{10}$.

On vient de voir dans cet exemple comment traduire (ou coder) un nombre écrit en base 2 en nombre en base dix.

Réciproquement, comment coder un nombre écrit en base 10 en un nombre écrit en base 2 ?

On peut imaginer au moins deux méthodes, nous en proposons une seule ici.

Soit a un nombre entier écrit en base 10.

Soit b son code binaire, que nous supposons composé de n bits :

$b = (b_{n-1} \dots b_2 b_1 b_0)_2$, avec les b_i valant 0 ou 1.

On appelle b_0 le **bit de poids faible** et b_{n-1} le **bit de poids fort**.

Alors a se décompose sous la forme suivante en base 10 :

$$a = b_{n-1}2^{n-1} + \dots + b_22^2 + b_12 + b_0 \quad (E)$$

On remarque que l'on peut factoriser une partie de l'écriture dans (E) par 2 :

$$a = 2(b_{n-1}2^{n-2} + \dots + b_22 + b_1) + b_0$$

Par conséquent, le bit de poids faible b_0 , qui est strictement inférieur à 2, est le reste de la division entière de a par 2. Le nombre $b_{n-1}2^{n-2} + \dots + b_22 + b_1$, quotient de cette division, est un nombre dont le code binaire est composé de $n - 1$ bits et dont le bit de poids faible est b_1 . On lui applique la même opération que précédemment, et on continue ainsi jusqu'à obtenir le bit de poids fort de b (obtenu au premier quotient valant 0).

Exemple Soit $a = 241$.

- | | |
|--|--|
| • $a = 2 \times 120 + 1$ donc $b_0 = 1$ | • $15 = 2 \times 7 + 1$ donc $b_4 = 1$ |
| • $120 = 2 \times 60 + 0$ donc $b_1 = 0$ | • $7 = 2 \times 3 + 1$ donc $b_5 = 1$ |
| • $60 = 2 \times 30 + 0$ donc $b_2 = 0$ | • $3 = 2 \times 1 + 1$ donc $b_6 = 1$ |
| • $30 = 2 \times 15 + 0$ donc $b_3 = 0$ | • $1 = 2 \times 0 + 1$ donc $b_7 = 1$ |

Le dernier quotient étant nul, on en déduit que $(241)_{10} = (11110001)_2$

Dans la machine, on pose généralement une limite à la taille des nombres que l'on utilise, souvent en multiple d'octets.

Ainsi, si on se limite à 8 bits, on ne peut manipuler les entiers qu'entre 0 $(00000000)_2$ et 255 $(11111111)_2$.

En général on utilise plutôt 4 octets (32 bits) ou 8 octets (64 bits).

2. Opérations sur les entiers

Les méthodes de calculs sont les mêmes qu'en base 10 lorsque l'on pose les opérations, avec les retenues. Mais il faut se rappeler que l'on ne dépasse pas le chiffre 1.

Exemple La somme de $(1101)_2$ et $(1001)_2$ donne $(10110)_2$.

Remarque Si la machine limite à 4 bits l'écriture des entiers, tout ce qui « dépasse » est coupé, autrement dit le résultat précédent devient faux (on aurait alors $13 + 9 = 6!$).

► **Exercices** : fiche sur les nombres

3. Autres bases, hexadécimal

Pour une base quelconque b , tout nombre n peut s'écrire (en décimal) sous la forme :

$$n = a_{n-1}b^{n-1} + \dots + a_2b^2 + a_1b + a_0$$

Son écriture en base b est alors $(a_{n-1} \dots a_1 a_0)_b$

On obtient les chiffres a_i en faisant des divisions euclidiennes successives par b , comme nous l'avons vu pour le binaire.

Exemple Soit le nombre (en base 10) 10 à écrire en base 3 : $10 = 3 \times 3 + 1$, $3 = 1 \times 3 + 0$, $1 = 0 \times 3 + 1$. Alors $10 = (101)_3$.

Inversement, soit $(21)_3$, alors $(21)_3 = 2 \times 3^1 + 1 \times 3^0 = 2 \times 3 + 1 \times 1 = 6 + 1 = 7$.

Une base particulière utilisée en informatique est la base 16, autrement dit le codage hexadécimal. Pour celle-ci, on a besoin de 16 caractères, donc on ajoute des lettres :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Elle est particulièrement intéressante car 16 est une puissance de 2, et pas n'importe laquelle : $16 = 2^4$. Cela implique qu'un octet peut s'écrire à l'aide de deux chiffres en hexadécimal (c'est à dire en base 16).

Dans les conversions, il faut tenir compte du fait que l'on a les correspondances suivantes :

$$A \leftrightarrow 10, B \leftrightarrow 11, C \leftrightarrow 12, D \leftrightarrow 13, E \leftrightarrow 14, F \leftrightarrow 15$$

Exemple Le nombre $(11010101)_2$ s'écrit $(D5)_{16}$.

En effet, en partageant en 2, on a 1101 qui vaut $2^3 + 2^2 + 1 = 13$ et est donc codé en D et 0101 qui vaut $2^2 + 1 = 5$ et est codé en 5.

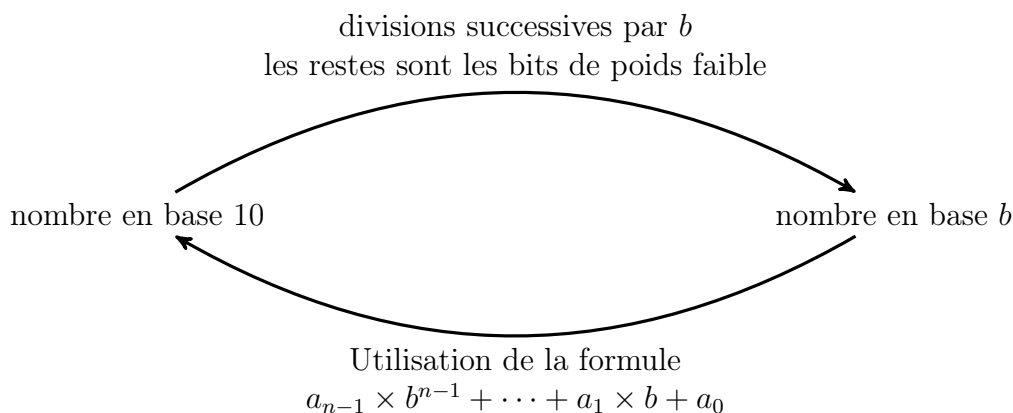
Alors $11010101_2 = 1101_2 \times 10000_2 + 0101_2 = D_{16} \times 10_{16} + 5_{16} = D0_{16} + 5_{16} = D5_{16}$.

Exemple On cherche le codage en hexadécimal du nombre 30 en base 10. On a :

$$\begin{aligned} 30 &= 16 \times 1 + 14 \rightarrow E \\ 1 &= 16 \times 0 + 1 \rightarrow 1 \end{aligned}$$

Ainsi, $(30)_{10} = (1E)_{16}$.

Pour résumer, voici le schéma de conversion à retenir :



Dans le cas où l'on souhaite passer d'une base b_1 à une base b_2 (les deux différentes de la base 10), on passe par l'étape intermédiaire de la base 10 :

$$\text{nombre en base } b_1 \longrightarrow \text{nombre en base 10} \longrightarrow \text{nombre en base } b_2$$

► **Exercices** : fiche sur les nombres

II. Booléens (bool)

Une variable de type bool ne peut prendre que deux valeurs : **True** (vrai) et **False** (faux).

Une **expression booléenne** est composée de booléens et d'opérateurs. Nous utiliserons les opérateurs logiques AND, OR et NOT, c'est à dire ET, OU et NON.

Pour chaque opérateur, on peut donner une table de vérité.

Exemple Table de vérité du AND, qui ne vaut **True** que si les deux opérandes valent **True** :

A \ B	True	False
True	True	False
False	False	False

Tableau à double entrée

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Tableau simple

Exemple Table de vérité du OR, qui vaut **True** si au moins l'une des deux opérandes vaut **True**.

A or B	True	False
True	True	True
False	True	False

Exemple Le NOT se contente d'échanger **True** en **False** et inversement.

A	not A
True	False
False	True

Les conditions que l'on met dans une structure algorithmique conditionnelle (Si, ou Tant que) sont des booléens.

Autrement dit, une expression comme $5 > 3$ est un booléen, dont la valeur est **True**.

L'évaluation d'une expression booléenne se fait de gauche à droite. C'est à dire que par exemple pour évaluer l'expression $a \text{ OR } b$, si a est évaluée à **True**, le résultat est **True**, sinon le résultat est celui de l'évaluation de b .

On parle de **séquentialité** des opérateurs AND et OR.

► **Exercices** : fiche sur les booléens

III. Chaînes de caractère (str)

Les chaînes de caractère sont des textes.

⊗ **Activité** : Recherches suivantes possibles pour des exposés courts :

- Comment sont manipulées les variables de type **str** en Python ? En particulier :
 - * Quelles opérations peut-on faire sur ces variables (voir les opérations sur les séquences) ?
 - * Présenter quatre ou cinq des méthodes disponibles, avec des exemples.

* Quels sont les séquences d'échappement ? À quoi servent-elles ?

Ces recherches pourront s'appuyer sur la documentation de la librairie Python.

- Donner une liste d'encodages de textes courants en expliquant leur historique (sans entrer dans les détails trop techniques).
- Où (ou comment) peut-on rencontrer des problèmes dus à l'encodage des caractères ?
- Détailler un encodage particulier : ASCII, UTF-8

IV. Entiers relatifs (int)

⊗ **Activité** : sur le codage des entiers relatifs

Pour représenter les entiers relatifs, il existe plusieurs idées. La première, naïve, consiste à utiliser le premier bit (de poids fort) comme un signe, en codant par 0 un nombre positif et par 1 un nombre négatif.

C'est un choix arbitraire mais qui a le mérite de ne pas transformer les nombres positifs, qui conservent comme code leur écriture binaire.

Cependant cette méthode rend l'opération de somme complexe, car la propriété suivante :

La somme (en binaire) des codes de deux nombres est le code de la somme des deux nombres

qui est intéressante à avoir pour simplifier les opérations, est fautive avec ce codage.

Il existe cependant un codage permettant d'avoir cette propriété. Il se base sur l'idée de fixer un nombre limite n de bits pour le codage (ce qui limite donc le nombre d'entiers que l'on peut coder). À partir de cette limite, tout nombre possède alors un opposé obtenu par une opération appelée complément à 2ⁿ ou, par abus de langage, **complément à 2**.

Pour faire le complément à 2, il y a deux étapes :

- Faire le complément à 1 (c'est à dire permuter tous les 0 et les 1)
- Ajouter 1 au résultat obtenu à la première étape, en faisant attention aux éventuelles retenues à propager

Cette procédure a l'avantage d'être réversible facilement : pour retrouver le nombre de départ, il suffit d'en faire le complément à 2. C'est à dire que le complément à 2 du complément à 2 est le nombre de départ.

Lorsque l'on fait la somme d'un nombre et de son complément à 2, on obtient $10 \dots 0$ écrit sur $(n+1)$ bits. La limite des n bits fait que l'on obtient donc en fait $0 \dots 0$, autrement dit simplement 0, car on « coupe » ce qui dépasse du côté des bits de poids fort. Cela permet de considérer que sur n bits, un nombre et son complément à 2 sont opposés l'un de l'autre.

On utilise alors ces propriétés pour coder les nombres entiers relatifs, avec une méthode appelée « méthode du complément à 2 ».

Voici la **règle de codage par la méthode du complément à 2** :

Une fois fixée la limite n du nombre de bits du codage, soit a un nombre entier relatif.

- Si a est positif, alors le codage est simplement l'écriture en binaire de a , où l'on ajoute autant de 0 que nécessaire en tête pour avoir n bits.
- Si a est négatif, alors le codage de a est le complément à 2 de l'écriture binaire de sa valeur absolue sur n bits.

Exemple Sur 6 bits, avec la méthode du complément à 2 :

- Pour obtenir le codage de -12 , qui est négatif :
 - * on prend le codage de 12 : 001100 (pour cela, faire les divisions successives par 2) ;
 - * on prend le complément à 1 (on inverse les bits) : 110011 ;
 - * on ajoute 1 : 110100.

Le codage de -12 par la méthode du complément à 2 sur 6 bits est donc 110100.

- Le codage de 12 est tout simplement 001100, autrement dit l'écriture binaire de 12, puisque ce nombre est positif.

Pour décoder, on applique la méthode suivante :

- Si le code commence par un 0, alors c'est le code d'un nombre positif. Ce code est simplement l'écriture en binaire du nombre cherché. Il suffit donc de traduire cette écriture binaire en une écriture en base 10 à l'aide de la formule déjà connue.
- Si le code commence par un 1, alors c'est le code d'un nombre négatif. On applique le complément à 2 sur le code. L'écriture obtenue est alors l'écriture en binaire de l'opposé (positif) du nombre cherché. Il suffit donc de traduire cette écriture binaire en une écriture en base 10 à l'aide de la formule déjà connue.

Exemple Sur 6 bits, on veut décoder :

- 110100 : le code commence par 1, c'est donc celui d'un nombre négatif.
On fait le complément à 2 :
* 001011 (complément à 1)
* 001100 (ajout de 1)
L'écriture 001100 est celle en binaire de l'opposé du nombre cherché.
Or $(001100)_2 = 2^3 + 2^2 = (12)_{10}$, donc le nombre codé par 110100 est le nombre -12 .
- 001100 : le code commence par 0, c'est donc celui d'un nombre positif. C'est donc l'écriture en base 2 du nombre recherché. Or $(001100)_2 = (12)_{10}$, donc le nombre codé par 001100 est le nombre 12.

On a les propriétés suivantes :

- Sur n bits, on peut représenter les nombres compris entre -2^{n-1} et $2^{n-1} - 1$
- Les nombres négatifs ont tous le bit de poids fort égal à 1,
- les écritures des nombres de 0 à $2^{n-1} - 1$ représentent les nombres positifs correspondants.
- Avec ce codage, la somme devient « naturelle », car on a la propriété suivante :
La somme (en binaire) des codes de deux nombres est le code de la somme des deux nombres

► **Exercices** : fiche sur les nombres (suite)

V. Nombres flottants (float)

Nous avons vu que l'on pouvait représenter les nombres entiers en machine, si ces nombres sont compris entre deux bornes qui dépendent du nombre d'octets utilisés. Pour les nombres réels, la difficulté est plus grande : il en existe de toute forme (décimaux comme $-2,56$, rationnels comme $\frac{1}{3}$, ou même irrationnels comme $\sqrt{2}$) et surtout, il existe toujours une infinité de nombres entre deux nombres réels quelconques.

Il est donc impossible de représenter tous les nombres réels, y compris entre deux bornes données. Nous ne pouvons d'ailleurs stocker qu'une partie finie des décimales d'un nombre réel donné (autrement dit, pas de valeur exacte pour $\frac{1}{3}$ ou π). Certains nombres seront alors représentés de manière exacte, et d'autres de manière approchée.

On rappelle dans un premier temps qu'un nombre décimal est un nombre qui peut s'écrire sous la forme $\frac{z}{10^n}$ avec z entier et n entier naturel.

Exemple $2,15 = \frac{215}{100} = \frac{215}{10^2}$.

De manière similaire, un **nombre dyadique** est un nombre qui peut s'écrire sous la forme $\frac{z}{2^n}$ avec z entier et n entier naturel.

Exemple $\frac{13}{4} = \frac{13}{2^2}$.

Pour obtenir le développement dyadique, c'est à dire l'écriture binaire, d'un nombre dyadique $\frac{z}{2^n}$, on prend le binaire correspondant à z et on insère la virgule avant le n -ième bit en partant de la droite.

Exemple $13 = (1101)_2$, donc $\frac{13}{2^2} = (11,01)_2$.

Cela signifie que $\frac{13}{4} = 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = 2 + 1 + 0 + 0,25 = 3,25$

Rappel $2^{-n} = \frac{1}{2^n}$

Une autre méthode, qui fonctionne avec n'importe quel nombre écrit sous forme décimale, consiste à prendre l'écriture binaire de la partie entière, puis de chercher l'écriture binaire de la partie décimale. Pour cela, on multiplie successivement par 2 la partie décimale et on prend le chiffre des unités du résultat comme bit, puis on refait la même chose en prenant la partie décimale si elle est non nulle :

Exemple Pour 3,25, on a déjà $3 = (11)_2$.

Ensuite,

$$0,25 \times 2 = 0,5 \rightarrow 0$$

$$0,5 \times 2 = 1 \rightarrow 1$$

Ainsi, on retrouve bien que $3,25 = (11,01)_2$

De même que tous les nombres ne sont pas décimaux, tous les nombres ne sont pas dyadiques, y compris certains nombres décimaux.

Exemple $(0,1)_{10} = (0,000110011001100110011\dots)_2$

En effet, en appliquant la méthode vue plus haut :

$$0,1 \times 2 = 0,2 \rightarrow 0$$

$$0,2 \times 2 = 0,4 \rightarrow 0$$

$$0,4 \times 2 = 0,8 \rightarrow 0$$

$$0,8 \times 2 = 1,6 \rightarrow 1$$

$$0,6 \times 2 = 1,2 \rightarrow 1$$

et on retombe sur 0,2, donc on va avoir une répétition de 0011

Par conséquent, la représentation (en binaire!) de $(0,1)_{10}$ dans la machine sera forcément tronquée. Cela explique que, lorsque l'on tape $0.1 + 0.2$ dans Python, le résultat n'est pas celui attendu (on obtient 0.30000000000000004).

 À cause de cela, faire des tests d'égalité entre nombres réels est généralement à éviter, puisque par exemple $0.1 + 0.2 == 0.3$ est évalué à **False**.

La représentation des nombres dits flottants (float) est régit par la norme IEEE-754.

Ensuite, on peut retenir que tout nombre dyadique non nul peut s'écrire sous la forme :

$(1, b_1 b_2 \dots b_k)_2 \times 2^n$ avec n entier relatif.

La suite de bits $(b_1 b_2 \dots b_k)$ est appelée la **mantisse**, et le nombre n est appelé **exposant**.

Sur un codage en 64 bits on a :

- Le bit fort représente le signe (0 pour +, 1 pour -) ;
- les 11 bits suivants représentent l'exposant ;
- les 52 autres bits représentent la mantisse.

► **Exercices** : fiche sur les nombres (suite)

VI. Types construits

⊗ **Activité** : Utiliser le fichier d'activité notebook qui reprend la plupart des éléments indiqués dans le cours ci-dessous.

Lorsque l'on a un nombre important de données de même nature, il peut être intéressant de pouvoir les regrouper dans un même objet. C'est l'intérêt des types construits.

Les types et les syntaxes dont il sera question dans cette section seront ceux du langage Python.

Il y a plusieurs possibilités.

- La première est de considérer que les données sont ordonnées, avec chacune un indice, et qu'il est impossible de modifier par affectation l'une des valeurs. C'est le cas pour le type **tuple**.
- La seconde est d'autoriser la modification d'une valeur par affectation, en gardant un ordre sur les données, donné par un indice. Cela correspond au type **list**.
- Une troisième possibilité est d'associer les éléments non pas à un indice, mais à une clé. On obtient alors le type **dict**.

1. N-uplets

Un n-uplet est un objet de type tuple. C'est une suite ordonnée d'éléments qui peuvent être de n'importe quel type.

Pour définir un n-uplet, il suffit d'écrire n valeurs séparées par des virgules.

```
t1=1,"abc",3.5
t2=4,(2,5),6
```

Les deux n-uplets contiennent chacun trois éléments. Le deuxième élément du n-uplet t2 est lui-même un n-uplet, délimité par les parenthèses. L'affichage d'un tuple contient toujours les parenthèses

Pour définir un 1-uplet, il faut au choix écrire quelque chose comme :

```
t=1,
t=(1,)
```

Pour le tuple vide :

```
t=()
```

Pour accéder à un élément d'un tuple, il suffit de donner son indice (dont le plus petit est 0) entre crochets. On peut aussi accéder à une partie du tuple (noter que comme dans `range`, le dernier élément est celui de l'indice précédent celui qui est donné en deuxième). On peut également utiliser des indices négatifs pour partir de la fin. Enfin, si un élément d'un tuple est un tuple, on peut évidemment accéder à ses éléments.

```
>>> t1[1]
"abc"
>>> t2[0:2]
(4,(2,5))
>>> t1[-1]
3.5
>>> t2[1][0]
2
```

La fonction `len` donne la longueur d'un tuple

```
>>> len(t1)
3
```

 les éléments d'un tuple ne sont pas modifiables. On ne peut donc pas écrire `t1[2]=4` pour modifier le troisième élément de `t1`.

On peut utiliser les tuples comme retour de fonctions pour renvoyer plusieurs valeurs, et utiliser l'affectation multiple pour récupérer chacune des valeurs.

```
>>> def division(a,b):
    r=a
    q=0
    while r>= b:
        r=r-b
        q=q+1
    return q,r

>>> x,y=division(25,7)
>>> x
3
>>> y
4
```

Il est possible de concaténer (c'est à dire mettre bout à bout) deux tuples avec l'opération `+`. Par exemple, `t1+t2` est le tuple `(1,"abc",3.5,4,(2,5),6)`.

taper `2*t1` revient à faire `t1+t1`. On peut alors aller au delà et faire par exemple `5*t1`.

On peut tester l'appartenance d'une valeur à un tuple avec l'opérateur `in` :

```
>>> 4 in t2
True
>>> 2 in t2
False
```

2. Listes

Les listes se manipulent un peu comme les tuples, sauf qu'elles permettent en plus l'affectation. Pour définir une liste, il faut utiliser des crochets :

```
l1=[1,2,3,4]
singleton=['a']
l=[] # liste vide
```

Il existe plusieurs moyens de créer des listes.

- la fonction `list` permet de transformer tout itérable en liste. On peut par exemple l'utiliser avec la fonction `range` :

```
>>> l=list(range(4))
>>> l
[0,1,2,3]
```

- On peut utiliser aussi la méthode `append` d'une liste pour ajouter des éléments.

```
l=[]
for n in range(5):
    l.append(3*n)
```

à la fin de l'exécution du code précédent, la liste `l` est la liste `[0,3,6,9,12]`.

- On peut aussi faire des listes **par compréhension**. La syntaxe est la suivante :

```
l=[expression(i) for i in objet]
```

où `expression(i)` est une expression dépendant de `i` et `objet` est un objet itérable (comme une liste, un tuple, un range etc.).

```
impairs=[2*i+1 for i in range(5)]
```

L'expression peut être donnée par une fonction définie au préalable

```
def f(x):
    return x**2-1

l=[f(i) for i in range(2,4)]
```

À la fin de l'exécution, la liste `l` est la liste `[3,8]`.

Il est possible de définir ainsi des listes de listes :

```
>>> l=[[x,y] for x in range(2) for y in range(2)]
>>> l
[[0,0],[0,1],[1,0],[1,1]]
```

Si les sous-listes sont de même longueur, on parle souvent de matrice.

On peut aussi ajouter des conditions :

```
>>> l=[x//2 for x in range(7) if x%2==0]
>>> l
[0, 1, 2, 3]
```

L'accès aux éléments se fait comme avec les tuples. La fonction `len` donne aussi la longueur d'une liste.

Nous avons vu la méthode `append` qui ajoute un élément à la fin. Il existe d'autres méthodes sur les listes, comme `insert`, `remove`, `count`, etc. Voir la librairie de Python pour plus de détails.

 le fait que les éléments d'une liste soient modifiables, cela demande d'être vigilant. En effet :

```
>>> l1=[1,2,3]
>>> l2=l1
>>> l2[1]=4
>>> l2
[1,4,3]
>>> l1
[1,4,3]
```

En fait, `l1` et `l2` partagent les mêmes **références** (ce sont des listes de références vers des objets), plus que les mêmes valeurs. Ainsi, en modifiant la valeur, on ne modifie pas les références, mais les objets qui sont pointés par ces références, ce qui fait que la liste `l1` est impactée par une modification d'une valeur dans la liste `l2`.

Pour éviter ces problèmes, pour une liste simple on peut simplement utiliser la fonction `list` :

```
>>> l1=[1,2,3]
>>> l2=list(l1)
>>> l2[1]=4
>>> l2
[1,4,3]
>>> l1
[1,2,3]
```

Pour des listes plus complexes (listes de listes, etc.) on peut utiliser la fonction `deepcopy` du module `copy`, ce qui permet de créer de nouvelles références et ainsi séparer en mémoire les deux listes.

```
from copy import deepcopy
l1=[[x,y] for x in range(2) for y in range(2)]
l2=deepcopy(l1)
```

Il peut être utile de le faire quand on donne à une fonction en argument une liste que la fonction peut modifier de manière interne mais qui ne doit pas l'être de manière globale.

3. Dictionnaires

Un dictionnaire (type `dict`) est une liste d'objets indexés non pas nécessairement par des entiers de 0 à (n-1), mais par des « clés », à la manière d'un dictionnaire.

```
dico1={"vrai":"true", "faux":"false"}
dico2={"a":"alpha", "b":"bravo", "c":"charlie"}
dico3={1:3,3:5,5:7}
```

Les éléments sont donnés sous la forme clé : valeur, le tout étant délimité par des accolades. On peut là aussi utiliser une construction par compréhension.

```
>>> d={x:x**2 for x in range(1,5)}
>>> d
{1: 1, 2: 4, 3: 9, 4: 16}
```

Les méthodes `keys` et `values` permettent respectivement d'obtenir les clés et les valeurs d'un dictionnaire. Pour obtenir les couples, il faut utiliser la méthode `items`.

L'opérateur `in` teste l'appartenance d'une clé (et non d'une valeur!).

L'itération sur un dictionnaire se fait donc sur la clé.

Par contre, on peut itérer sur les couples (clé,valeur) en utilisant la méthode `items` :

```
>>> for cle,val in dico2.items():
    print(cle,"donne",val)

a donne alpha
b donne bravo
c donne charlie
```

Pour obtenir la valeur associée à une clé, on donne la clé entre crochets :

```
>>> dico2["a"]
"alpha"
>>> dico3[3]
5
```

La fonction `len` donne là encore la longueur du dictionnaire (le nombre d'entrées, d'éléments). Pour un dictionnaire, il est facile d'ajouter un élément (pour la liste il faut utiliser une méthode comme `append`) :

```
dico2["d"]="delta"
```

Pour supprimer un élément, on utilise la fonction `del` :

```
del(dico3[3])
```

La méthode `get` permet d'obtenir la valeur associée à une clé, ou `None` si la clé n'est pas présente, ce qui évite les erreurs.

```
>>> v = dico2.get("c")
>>> v
"charlie"
>>> v = dico2.get("e")
>>> v
None
```

Pour les copies, on retrouve les mêmes comportements qu'avec les listes.

Pour plus de détails, d'utilisations, etc. sur les trois types tuples, listes et dictionnaires, voir la librairie Python.

► **Exercices** : fiche d'exercices sur les types construits.