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 **bi**nary digi**t** (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 supposerons 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 :

$$a = b_{n-1}2^{n-1} + \cdots + b_2 2^2 + b_1 2 + b_0 \qquad (E)$$

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

$$a = 2\left(b_{n-1}2^{n-2} + \cdots + b_2 2 + b_1\right) + b_0$$

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 $b_{n-1}2^{n-2} + \cdots + b_2 2 + b_1$, 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 $b_0 = 1$
- $120 = 2 \times 60 + 0$ donc $b_1 = 0$
- $60 = 2 \times 30 + 0$ donc $b_2 = 0$
- $30 = 2 \times 15 + 0$ donc $b_3 = 0$
- $15 = 2 \times 7 + 1$ donc $b_4 = 1$
- $7 = 2 \times 3 + 1$ donc $b_5 = 1$
- $3 = 2 \times 1 + 1$ donc $b_6 = 1$
- $1 = 2 \times 0 + 1$ 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 :

$$n = a_{n-1}b^{n-1} + \cdots + a_2 b^2 + a_1 b + a_0$$

Son écriture en base b est alors $(a_{n-1}...a_1 a_0)_b$
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 : $10 = 3\times3+1$, $3 = 1\times3+0$, $1 = 0\times3+1$.
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).

Exemple Le nombre $(11010101)_2$ s'écrit $(D5)_{16}$.
En effet, en partageant en 2, on a 1101 qui est codé en D et 0101 qui est codé en 5.
Alors $11010101_2 = 1101_2 \times 10000_2 + 0101_2 = D_{16} \times 10_{16} + 5_{16} = D0_{16} + 5_{16} = D5_{16}$.

▶ **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`.

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

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

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 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)

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

Exemple Dans un codage sur 4 bits, le nombre -2 est représenté par $(1010)_2$

Mais les opérations sont rendues plus compliquées avec ce choix de codage.

Exemple Avec la somme « naturelle » des codages de 4 et -2, on n'obtient pas 2 !

Il faut donc trouver une autre idée.
Limitons-nous pour commencer, par simplicité, à une taille de nombres en binaires sur 4 bits.
quand on ajoute 15 (1111) et 1 (0001), on obtient 10000, autrement dit 0000 puisque le bit à gauche est perdu. Autrement dit, sur 4 bits, 15 est l'opposé de 1. On peut donc considérer que le nombre -1 peut être codé par 1111 sur 4 bits.

Exemple Quel nombre ajouter à $(0101)_2$ pour obtenir $(10000)_2$?

On remarque que pour trouver l'opposé, il suffit d'inverser tous les bits puis d'ajouter 1.
Le nombre obtenu par cette opération est appelé complément à 2^4 et plus simplement, par abus de notation, **complément à 2**.

Exemple Sur 6 bits, pour obtenir le codage de -12 avec le complément à 2 :
- on prend le codage de 12 : 001100.
- on prend le complément à 1 (on inverse les bits) : 110011.
- on ajoute 1 : 110100.

Le complément à 2 de -12 sur 6 bits est donc 110100.

Avec 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, et les écritures des nombres de 0 à $2^{n-1}-1$ représentent les nombres positifs correspondants.
Dans ce cas, la somme devient plus naturelle.

Exemple Somme de 6 et -2 avec la représentation par le complément à 2.

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

V. Nombres réels (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 (autre-

ment 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,
$0{,}25 \times 2 = 0{,}5 \to 0$
$0{,}5 \times 2 = 1 \to 1$
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 $(0{,}1)_{10} = (0{,}00011001100110011...)_2$
En effet, en appliquant la méthode vue plus haut :
$0{,}1 \times 2 = 0{,}2 \to 0$
$0{,}2 \times 2 = 0{,}4 \to 0$
$0{,}4 \times 2 = 0{,}8 \to 0$
$0{,}8 \times 2 = 1{,}6 \to 1$
$0{,}6 \times 2 = 1{,}2 \to 1$
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 :

$(1,b_1b_2\dots b_k)_2 \times 2^n$ avec n entier relatif.
La suite de bits $(b_1b_2\dots b_k)$ 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)