Chapitre 8 : Représentation des nombres

Bibliothèque et primitives introduites dans ce cours : `bin`.

1 Numération et bases

Pour représenter le nombre $n \in \mathbb{N}$, seuls dix chiffres sont nécessaires. La notation décimale s'appelle aussi notation à position en base dix. Formellement, si n est le nombre $a_k a_{k-1} \cdots a_1 a_0$, où a_0, a_1,..., a_k sont des chiffres compris entre zéro et neuf, alors on a l'égalité :

$$n = \sum_{j=0}^{k} a_j 10^j.$$

Le choix de faire des paquets de dix, c'est-à-dire le nombre de chiffres utilisés pour représenter les nombres entiers, est conventionnel. On aurait tout aussi bien pu décider d'utiliser deux, trois, douze, vingt, soixante chiffres,.... Dans ce cas, si n désigne le nombre $a_k a_{k-1} \cdots a_1 a_0$ écrit en base b, alors les chiffres a_0, a_1,..., a_k sont compris entre zéro et $b-1$, et on a l'égalité :

$$n = \sum_{j=0}^{k} a_j b^j.$$

On parle alors d'écriture en base b, où b désigne le nombre de chiffres utilisés. Dans ce qui suit, si une suite de chiffres désigne un nombre écrit dans une base autre que 10, on indiquera la base b choisie de la façon suivante : $\overline{1011}^2$ désigne le nombre $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$ écrit en base 2.

Remarque 1.1.

Exercice 1.2. On considère qu'en base 11, on note les chiffres de 0 à 9 puis A. Que vaut le nombre $\overline{8A0A}^{11}$?

La mémoire des ordinateurs est constituée d'une multitude de petits circuits électroniques qui, chacun, ne peuvent être que dans deux états : sous tension ou hors tension. Techniquement, il serait possible de construire des circuits ayant plus de deux états, correspondant à différentes valeurs de tensions, mais les risques d'erreurs dans le stockage et la transmission de ces états deviennent vite beaucoup plus importants que les avantages qu'on pourrait en tirer. Comme il fallait donner un nom à ces états, on a décidé de les appeler 0 et 1, mais on aurait pu tout aussi bien les appeler A et B, froid et chaud, faux et vrai, etc. Une telle valeur, 0 ou 1, est dite booléenne. On l'appellera booléen, chiffre binaire ou encore bit. Un tel circuit à deux états s'appelle un circuit mémoire un bit. L'état dans lequel se trouve un circuit mémoire un bit est représenté par le symbole 0 ou par le symbole 1. L'état d'un circuit composé de plusieurs de ces circuits est représenté par une suite finie de 0 et de 1, que l'on appelle un mot. Par exemple, le mot 100 décrit l'état d'un circuit composé de trois circuits mémoire un bit, respectivement dans l'état 1, 0 et 0.

Exercice 1.3.
On imagine un ordinateur dont la mémoire est constituée de n circuits mémoire un bit. Quel est le nombre d'états possibles de la mémoire de cet ordinateur ?

Les ordinateurs comptent donc naturellement en base deux et chaque circuit représente un des deux chiffres de cette base : 0 ou 1. Le nombre $13 = \overline{1101}^2$ est donc représenté dans la mémoire d'un ordinateur par le mot 1101, c'est-à-dire par quatre circuits respectivement dans les états 1, 0, 1 et 1. Dans la mémoire des ordinateurs, les circuits mémoire un bit sont souvent groupés par huit (les octets) et on utilise des nombres exprimés en notation binaire sur un, deux, quatre ou huit octets, soit 8, 16, 32 ou 64 bits. Cela permet de représenter les nombres de 0 à $\overline{1111\ 1111}^2 = 255$ sur un octet, de 0 à $\overline{1111\ 1111\ 1111\ 1111}^2 = 65535$ sur deux octets,... Lorsque l'on manipule des entiers naturels uniquement représentés de cette façon, dans certains langages on les appelle entiers non signés. En Python, toutefois, aucun type de données ne donne directement accès à ce genre de représentation : on ne peut manipuler que des entiers relatifs.

Exercice 1.4.

1. Écrire le nombre 37 en base 2.
2. Réaliser directement l'addition de 37 et 17 en binaire.

Exercice 1.5.
Écrire une fonction `binaire` qui prend en entrée un entier `n`, puis renvoie son écriture en base 2, sous forme d'une liste ne contenant que des 0 et des 1.

Remarque 1.6.

2 Représentation des entiers relatifs

2 A) Notation en complément à deux

Il faut étendre aux entiers relatifs la représentation binaire des entiers naturels. Une solution consiste à réserver un bit pour le signe de l'entier et à utiliser les autres pour représenter sa valeur absolue. Ainsi, avec des mots de 16 bits, en utilisant 1 bit pour le signe et 15 bits pour la valeur absolue, on pourrait représenter les entiers relatifs de $-\overline{111\ 1111\ 1111\ 1111}^2 = -32767$ à $\overline{111\ 1111\ 1111\ 1111}^2 = 32767$. Cependant, cette méthode a plusieurs inconvénients, notamment l'existence de deux zéros, l'un positif et l'autre négatif. On applique alors une autre méthode, qui consiste à représenter un entier relatif par un entier naturel. Si on utilise des mots de 16 bits, on peut représenter les entiers relatifs compris entre -32768 et 32767 : on représente un entier relatif x positif ou nul comme l'entier naturel x et un entier relatif x strictement négatif comme l'entier naturel $x + 2^{16} = x + 65536$, qui est compris entre 32768 et 65535.

Cette manière de représenter les entiers relatifs s'appelle la notation en complément à deux. L'entier relatif -1 est représenté comme l'entier naturel 65535, c'est-à-dire par le mot 1111 1111 1111 1111. On notera qu'il reste facile de déterminer le signe d'un entier représenté sous cette forme : un entier relatif positif ou nul est représenté par un entier naturel dont le premier bit vaut 0 ; a contrario, un entier relatif strictement négatif est représenté par un entier naturel dont le premier bit vaut 1. Plus généralement, avec des mots de n bits, on peut représenter les entiers relatifs compris entre -2^{n-1} et $2^{n-1} - 1$.naturel x (compris entre 0 et $2^{n-1} - 1$) et un entier relatif x strictement négatif comme l'entier naturel $x + 2^n$ (compris entre 2^{n-1} et $2^n - 1$).

Exemple 2.1. Écrivons les représentations en complément à deux de -37 et 17 pour obtenir leur somme en binaire.

2 B) Dépassement de capacité

Dans les versions Python 2.x, le type `int` désigne des entiers relatifs représentés sur 32 bits (quatre octets), donc dont les valeurs vont de $-2\ 147\ 483\ 648$ à $2\ 147\ 483\ 647$. Cependant, si l'on travaille sur une machine 64 bits, la représentation des entiers sera également faite sur 64 bits : on disposera donc des entiers de $-9\ 223\ 372\ 036\ 854\ 775\ 808$ à $9\ 223\ 372\ 036\ 854\ 775\ 807$. L'ensemble des valeurs de type `int` dépend donc de la machine sur laquelle Python est exécuté.

Puisque les nombres représentables de cette façon sont limités, on peut se demander ce qui se passe lorsque l'on atteint ces limites. Par exemple, si $a = 260$, alors :

$$a^4 = 4\,569\,760\,000 = \overline{1\ 00010000\ 01100001\ 00000001\ 00000000}^2.$$

Si l'on reste dans une représentation sur 32 bits, le bit le plus à gauche est perdu : on ne mémorise que le mot $\overline{00010000\ 01100001\ 00000001\ 00000000}^2 = 274\ 792\ 704$, qui n'est évidemment pas le résultat attendu. On appelle ce phénomène dépassement arithmétique (overflow en anglais).

Il faut donc changer de représentation pour éviter de perdre la valeur du résultat. En Python, ce changement de représentation est fait automatiquement.La seule limite pour la représentation des entiers, qu'ils soient naturels ou relatifs, est la mémoire disponible sur la machine. Les exemples précédents de dépassements arithmétiques ne se produisent donc pas.

Lorsque l'on veut représenter un entier x, on découpe la représentation binaire de x par paquets de 15 bits et on stocke les entiers correspondants dans un tableau d'entiers naturels sur 16 bits. Ce tableau contient donc

les chiffres de x en base 2^{15}. De plus, à ce tableau on associe un entier taille qui donne le nombre de chiffres de x en base 2^{15} et dont le signe est le signe de x.

Par exemple, le nombre 4 569 760 000 $= \overline{1\ 00010000\ 01100001\ 00000001\ 00000000}^2$ comporte 33 bits, que l'on découpe donc en 100 010000011000010 000000100000000. On a alors :

$$\overline{000000100000000}^2 = 256\ ,\ \overline{010000011000010}^2 = 8386 \text{ et } \overline{100}^2 = 4.$$

Le nombre 4 569 760 000 est donc représenté en mémoire par le tableau `[256, 8386, 4]`, auquel on associe le nombre 3 pour signifier qu'il y a trois chiffres et que l'entier à représenter est positif. Le nombre -4 569 760 000 sera représenté par le même tableau associé au nombre -3. Enfin, cas particulier, le nombre 0 est représenté par un tableau vide associé au nombre de chiffres 0.

3 Représentation des nombres décimaux

3 A) Codage des nombres de $[0, 1[$

De façon tout à fait analogue à la représentation des nombres entiers naturels, par exemple, on a

$$0.241 = 2.10^{-1} + 4.10^{-2} + 1.10^{-3}.$$

De façon générale, dans une base quelconque, si b est en entier naturel supérieur ou égal à 2, pour tout $x \in [0, 1[$, il existe un entier $q \in \mathbb{N}$ et $b_{-1}, \dots, b_{-q}$ tels que

$$n = b_{-1}b^{-1} + b_{-2}b^{-2} + \cdots + b_{-q}b^{-q} + r_q$$

avec $0 \leqslant r_q < b^{-q}$. On dit alors que l'écriture

$$x = \overline{b_{-1}b_{-2}\cdots b_{-q}}^b$$

est une approximation de x en base b à b^{-q} près.

Exercice 3.1. Déterminer une écriture en binaire de $0,4$.

Exercice 3.2.
Écrire une fonction `binaire_decimal` qui prend en entrée un nombre décimal `x` de $[0, 1[$, un entier n, puis renvoie son écriture en base 2, sous forme d'une liste à n termes ne contenant que des 0 et des 1.

3 B) Les flottants

La notation binaire permet donc de représenter des nombres à virgule *a priori*. En notation décimale, les chiffres à gauche de la virgule représentent des unités, des dizaines, des centaines,... ceux à droite de la virgule, des dixièmes, des centièmes, des millièmes, ... De même, en binaire, les chiffres à droite de la virgule représentent des demis, des quarts, des huitièmes, des seizièmes, ... Par exemple, $1,25 = 1+\frac{1}{4} = 2^0+2^{-2} = \overline{1,01}^2$. Toutefois, cette manière de faire n'est pas très commode pour représenter des nombres très grands ou très petits comme le nombre d'Avogadro ou la constante de Planck. On utilise donc une autre représentation similaire à la notation scientifique des calculatrices, sauf qu'elle est en base deux plutôt qu'en base dix et définie dans la norme IEEE 754. Un nombre est représenté sous la forme $sm2^n$ où s est le signe du nombre, n son exposant et m sa mantisse (un nombre décimal compris entre 1 inclus et 2 exclu). Quand on utilise 32 bits pour représenter un nombre à virgule, on utilise 1 bit pour le signe, 8 bits pour l'exposant et 23 bits pour la mantisse. Les nombres représentés sous cette forme sont appelés nombres à virgule flottante, puisque la virgule de la mantisse peut être déplacée par le biais de l'exposant.

Principe de codage des nombres flottants sur 32 bits :

3 C) Dépassement de capacité et problèmes de précision

De même que les entiers, les nombres à virgule flottante possèdent certaines limites. Les nombres à virgule flottante étant représentés sur un nombre donné de bits, il existe forcément un nombre maximal représentable dans ce format.

Exemple 3.3. Déterminons le plus grand nombre représentable en 32 bits avec la représentation précédente ainsi que le plus petit nombre strictement positif.

D'autre part, le résultat d'un calcul faisant intervenir deux nombres à virgule flottante peut ne pas donner un résultat représentable exactement. A priori, il n'est pas possible de représenter exactement $0,4$ en binaire. Même sans effectuer de calcul, les nombres décimaux ne sont pour la plupart pas représentables exactement dans ce format.

La représentation en virgule flottante sera donc forcément une valeur approchée de ce nombre. Par défaut, la norme IEEE 754 impose que les nombres à virgule soient arrondis à la valeur représentable la plus proche.

Par exemple, la valeur approchée choisie pour $0,4$ est donc la suivante :

$$\overline{0,01100110\ 01100110\ 01100110\ 01100110\ 01100110\ 01100110\ 01101000}^2 = \frac{7\ 205\ 759\ 403\ 792\ 794}{18\ 014\ 398\ 509\ 481\ 984}$$
$$\simeq 0,400\ 000\ 000\ 000\ 000\ 022\ 204\ 460\ 492\ 503.$$

À cause de la base utilisée, il est donc impossible de représenter exactement la plupart des nombres décimaux, plus précisément tous ceux qui ne s'écrivent pas sous la forme $\frac{k}{2^n}$.

D'autres erreurs d'arrondis se présentent lorsque l'on effectue des calculs, notamment entre des nombres dont les ordres de grandeur sont très différents. Par exemple, $1 + 2^{-53}$ est arrondi à 1. Par exemple :

```
>>> 1+2**-52
1.0000000000000002
>>> 1+2**-53
1.0
>>> 1+2**-53-1
0.0
```

La somme est en effet calculée en premier et arrondie à 1, puis la différence est calculée et le résultat vaut donc 0. En revanche, on a :

```
>>> 1-1+2**-53
1.1102230246251565e-16
```

Des erreurs d'arrondi similaires peuvent se produire avec toutes les opérations usuelles. On retiendra qu'il n'est pas possible de savoir de façon certaine si le résultat d'un calcul est égal à sa valeur théorique. Une variable flottante peut donc être nulle alors qu'elle ne devrait pas l'être, et inversement.

Remarque 3.4.

Voilà un dernier exemple illustrant les erreurs d'arrondi, qui fait apparaître des infinis et des NaN (pour *Not a Number*) : la valeur NaN est renvoyée lorsque la machine est confrontée à une forme indéterminée, dont elle ne peut calculer la valeur.

```
>>> a=2*10**(-312)
>>> x=1/a
inf
>>> 0*x
nan
```

```
>>> x-x
nan
>>> x/x
nan
>>> a*x
inf
```

La réponse dans le dernier cas exprime le fait que le produit d'un nombre non nul (ici a) par l'infini (ici x) est égal à un infini.