Chapitre 2 : Représentation binaire des nombres réels

Vous avez vu que Python pouvait manipuler des nombres décimaux particuliers appelés nombres *flottants* qui correspondent aux représentations en machine de nombres réels. Dans ce chapitre, nous allons voir comment on peut écrire en binaire un nombre réel et comment sont encodés les nombres flottants.

I. Ecriture binaire d'un nombre réel

En notation décimale, les chiffres à droite de la virgule représentent les dixièmes, les centièmes, les millièmes, etc.
De la même manière, en binaire, les chiffres à droite de la virgule représentent les demis, les quarts, les huitièmes, les seizièmes, etc.

Par exemple, le nombre $(1{,}1101)_2$ est le nombre 1 et 1/2 et 1/4 et 1/16.

Conversion de la base 2 vers la base 10

Les chiffres à gauche de la virgule correspondent à des puissances de 2 positives, ceux situés à droite correspondent à des puissances de 2 négatives.

Exemple **A quel nombre décimal correspond l'écriture binaire 101,1101 ?**

Puissances de 2	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
Ecriture binaire	1	0	1	,	1	1	0	1

On a donc : $(101{,}1101)_2 = 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4} = 4 + 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = 5{,}8125$.

Conversion de la base 10 vers la base 2

On commence par convertir la partie entière (à gauche de la virgule) : $5 = (101)_2$.

Pour la partie décimale (à droite de la virgule), on effectue des multiplications successives par 2. Après chaque multiplication, la partie entière n'est pas reportée. On poursuit le calcul jusqu'à obtenir 1.

Exemple **Quelle est l'écriture binaire du nombre décimal 5,8125 ?**

On convertit 5 en binaire. Facile : $5 = (101)_2$.
On fait des multiplications successives par 2 sans reporter la partie entière :

$0{,}8125 \times 2 = \mathbf{1}{,}625$
$0{,}625 \times 2 = \mathbf{1}{,}25$
$0{,}25 \times 2 = \mathbf{0}{,}5$
$0{,}5 \times 2 = \mathbf{1}$

Les parties entières (0 ou 1) donnent les chiffres après la virgule de l'écriture binaire de $0{,}8125$, à lire de haut en bas :

$$0{,}8125 = (0{,}\mathbf{1101})_2$$

Finalement : $5{,}8125 = (101{,}1101)_2$.

A faire **Exercice 1**

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS

Ecritures infinies

Il existe des nombres dont l'écriture binaire sera infinie (et périodique). Par exemple, $1{,}2 = (1{,}001100110011\ldots)_2$, le cycle « 0011 » se répétant à l'infini.
Comme on ne peut pas représenter en machine un mot infini, il ne sera pas possible de représenter de manière exacte certains nombres réels (beaucoup d'entre eux !). Nous verrons cela un peu plus tard avec tous les problèmes que cela engendre.

II. Représentation des nombres réels

Dans un ordinateur, les nombres à virgule (réels) sont codés en virgule flottante. On parle de nombres flottants (le type float de Python). La représentation binaire en machine d'un nombre flottant s'inspire de l'écriture scientifique des nombres décimaux dont voici quelques rappels.

Ecriture scientifique

L'écriture scientifique permet d'uniformiser la façon d'écrire des nombres décimaux. Par exemple,

$$3542 \quad \text{s'écrit} \quad +3{,}542 \times 10^3$$
$$-0{,}0724753 \quad \text{s'écrit} \quad -7{,}24753 \times 10^{-2}$$

Dans cette écriture, on distingue :

- Un *signe* ($+$ ou $-$) ;
- Un nombre décimal, appelé *mantisse*, compris dans l'intervalle $[1;10[$ (1 inclus et 10 exclu). Dans les deux exemples, il s'agit de $3{,}542$ et $7{,}24753$;
- Un entier relatif n, appelé *exposant*. Dans les deux exemples, il s'agit de 3 et -2.

Ainsi, de manière générale, l'écriture scientifique d'un nombre décimal est :

$$\pm m \times 10^n$$

La norme IEEE 754

La norme IEEE 754 est la plus utilisée pour représenter les nombres flottants. Ils sont représentés sur 32 bits (format appelé « simple précision » ou binary32) ou sur 64 bits (format appelé « double précision », ou binary64) sous la forme :

$$s.m \times 2^n$$

où :

- s est le **signe** du nombre, codé sur 1 bit (0 pour $+$ et 1 pour $-$) ;
- n son **exposant** en puissance de 2, codé sur 8 bits (en format 32 bits) ou sur 11 bits (en format 64 bits) ;
- m sa **mantisse** codée sur 23 bits (en format 32 bits) ou sur 52 bits (en format 64 bits).

Ainsi, en machine, un flottant est représenté en format 32 bits (simple précision) par un mot binaire de la forme

1 bit	8 bits	23 bits
signe	exposant	mantisse

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS

et en format 64 bits (double précision) par un mot binaire de la forme

1 bit	11 bits	52 bits
signe	exposant	mantisse

Exemple **Représentation machine du nombre 5,8125**

On sait que $5{,}8125$ est positif donc le bit de signe sera 0 :

signe	exposant	mantisse
0	?	?

Rappelons que $5{,}8125 = (101{,}1101)_2$. Par analogie avec l'écriture scientifique, on peut aussi écrire ce nombre binaire : $1{,}011101 \times 2^2$ en décalant la virgule de deux rangs vers la gauche. En faisant cela, on a fait apparaître :

- la mantisse : $m = 1{,}011101$
- l'exposant : 2

Il reste maintenant à voir comment sont codés la mantisse et l'exposant.

Codage de la mantisse

Pour représenter les flottants, la base choisie est la base 2 (contrairement à l'écriture scientifique qui est la base 10) donc la mantisse est dans l'intervalle $[1;2[$. Il s'agit donc d'un nombre de la forme :

$$m = 1,xx\ldots xx$$

Comme cette mantisse commence toujours par le chiffre 1, il a été choisi de ne pas coder ce « 1 » mais uniquement les chiffres *après* la virgule.

Exemple (suite) **Représentation machine du nombre 5,8125**

La mantisse de ce nombre est $m = 1{,}011101$. Comme le « 1 » à gauche de la virgule n'est pas codé, la mantisse sera codée par 011101**00…0** en ajoutant autant de zéros que nécessaires pour arriver à 23 bits (simple précision) ou 52 bits (double précision) :

signe	exposant	mantisse
0	?	**011101****00…0**
		(23 ou 52 bits)

Codage de l'exposant

L'exposant est codé sur 8 bits ou 11 bits selon le format utilisé. Sur 8 bits on peut coder 256 valeurs : les entiers compris entre -127 et 128. Sur 11 bits on peut coder 2048 valeurs : les entiers compris entre -1023 et 1024 (voir Chapitre 1, Séquence 7).

L'exposant est un entier relatif mais la norme IEEE 754 n'utilise pas l'encodage par complément à 2 des entiers relatifs. Elle prévoit un décalage qui dépend de l'encodage utilisé :

- Dans le format simple précision (32 bits), le décalage est 127, c'est-à-dire qu'il faut ajouter 127 à l'exposant.
- Dans le format double précision (64 bits), le décalage est de 1023.

L'objectif est d'obtenir un nombre positif pour coder l'exposant. En effet, dans le format simple précision, en procédant à ce décalage de 127, on obtient des valeurs positives :

Exposants signés	−127	−126	…	0	…	127	128
Entier n à coder	0	1		127		254	255

+ 127

Exemple (suite) **Représentation machine du nombre 5,8125**

Rappelons que $5{,}8125 = (101{,}1101)_2 = (1{,}011101 \times 2^2)_2$ donc l'exposant est 2.
La norme IEEE 754 ne prévoit pas de coder +2 en complément à deux mais d'ajouter 127 en format simple précision (1023 en format double précision) : on obtient alors $2 + 127 = 129$.

Il ne reste plus qu'à coder l'entier 129 en binaire : $(10000001)_2$. On ajoute éventuellement des zéros (à gauche !) pour compléter les 8 bits réservés à l'exposant.

Dans le format simple précision (sur 32 bits), on obtient finalement :

signe	exposant	mantisse
0	**10000001**	01110100…0
	(8 bits)	(23 bits)

Dans le format double précision (sur 64 bits), on ajouterait 1023 à la puissance pour obtenir 1025 $(2 + 1023)$ dont l'écriture binaire est $(10000000001)_2$. On obtiendrait :

signe	exposant	mantisse
0	**10000000001**	01110100…0
	(11 bits)	(52 bits)

Bilan : En format simple précision, le nombre réel 5,8125 est représenté sur 32 bits en machine par le mot :

0 10000001 01110100000000000000000

En format double précision, il est représenté sur 64 bits en machine par le mot :

0 10000000001 0111010000000000000000000000000000000000000000000000

A faire **Exercices 2, 3, 4 et 5**

III. Une représentation approximative

Codage approché de certains réels

Tous les nombres réels dont l'écriture en base est infinie ne peuvent être représentés de manière exacte en machine. Par exemple, on a vu que le nombre décimal $1{,}2$ a une écriture binaire infinie : $1{,}2 = (1{,}001100110011\ldots)_2 = +1{,}001100110011\ldots \times 2^0$

La mantisse de ce nombre est $m = 1{,}001100110011\ldots$ donc elle est infinie. Or, il n'y a que 23 ou 52 bits réservés pour coder la mantisse. L'ordinateur doit donc tronquer la mantisse à 23 ou 52 bits. Cela signifie qu'en format simple précision, la mantisse du nombre $1{,}2$ est codée par le mot binaire de 23 bits

$$\underbrace{00110011001100110011001}_{23\text{ bits}}\ \cancel{100110011\ldots}$$

les autres bits ne pouvant pas être codés.

Seule une valeur tronquée de la mantisse peut être codée et donc la représentation en machine du nombre réel $1{,}2$ n'est qu'une approximation du réel $1{,}2$. Autrement dit, **le nombre flottant `1.2` n'est qu'une valeur approchée du nombre réel $1{,}2$.**

Cet exemple n'en est qu'un parmi tant d'autres : il y a une infinité de nombres réels qu'il est impossible de représenter de manière exacte en machine.

Méfiance

En particulier, dans un langage de programmation, il faut se méfier lorsque l'on manipule des nombres flottants. Par exemple, ajouter ou soustraire des flottants peut conduire à des résultats surprenants (voir ci-contre)

```
Entrée [1]: 1.2-1.0
Out[1]: 0.19999999999999996

Entrée [2]: 0.5 - 0.2 - 0.2 - 0.1
Out[2]: -2.7755575615628914e-17

Entrée [3]: 9007199254740992.0 + 1.0 + 1.0
Out[3]: 9007199254740992.0

Entrée [4]: 1.0 + 1.0 + 9007199254740992.0
Out[4]: 9007199254740994.0
```

```
Entrée [5]: 0.1 + 0.2 == 0.3
Out[5]: False

Entrée [6]: from math import sqrt
            sqrt(2.0)**2 == 2.0
Out[6]: False

Entrée [7]: 9007199254740992.0+1.0==9007199254740992.0
Out[7]: True
```

De même, il ne faut jamais tester l'égalité entre deux flottants (voir ci-contre).

Impossibilité de coder tous les nombres réels

Voici l'écriture binaire en format double précision de deux flottants.

Nombre flottant	Représentation format double précision (mantisse sur 52 bits)
`1.5`	$+1.1000000000000000000000000000000000000000000000000000 \times 2^0$
`1.5000000000000002`	$+1.1000000000000000000000000000000000000000000000000001 \times 2^0$

On comprend aisément qu'il n'y a pas de flottant entre `1.5` et `1.5000000000000002`. Le flottant `1.5000000000000001` est donc représenté comme `1.5`.

La précision possible avec une mantisse sur 52 bits se situe au niveau dernier bit qui vaut 2^{-52} soit environ 2×10^{-16}. L'écart entre 1.5 et `1.5000000000000002` étant égal à 2×10^{-16}, on ne peut pas trouver un flottant compris entre les deux.

A retenir

Les nombres **flottants** sont une **représentation approximative** des nombres réels dans un ordinateur. En particulier, il n'est pas possible de représenter de manière exacte en machine tous les nombres réels. La manipulation de nombres réels par un langage informatique est donc à prendre avec précaution car elle peut engendrer des résultats surprenants, en particulier il ne faut jamais tester l'égalité entre deux flottants.

Sources :

- Documents ressources du DIU EIL, C. DECLERCQ, Université de Nantes.
- *Prépabac spécialité 1ère NSI*, C. Adobet, G. Connan, G. Rozsavolgyi, L. Signac, éditions HATIER
- *Numérique et Sciences Informatiques*, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, Ellipses.