RND. Nombres réels

I/ Introduction

Dans les chapitres précédents, on a appris à représenter des nombres entiers relatifs en langage machine. Python permet de représenter avec exactitude chacun de ses nombres, il lui suffit pour cela de prendre le nombre d'octets nécessaires.

Qu'en est-il avec les nombres réels ? Ces nombres sont de différents types : entier comme 3, décimal comme -1,6, rationnel comme 5/3, irrationnel comme $\sqrt{2}$. Il y a une **infinité de nombre réels** entre deux bornes quelconques, or, il n'existe qu'un nombre fini d'octets !

A savoir : Il est **impossible** de **représenter exactement chaque nombre réel**, on n'obtiendra la plupart du temps qu'une **valeur approchée**.

II/ Représentation d'un nombre réel en écriture binaire

Contrairement à la représentation des nombres entiers relatifs qui suivent une logique de calcul (cela interdit par exemple d'assigner à un bit le signe du nombre), celle des nombres réels est abstraite pour une machine qui ne connaît que deux états à savoir 0 et 1. On va donc ici pouvoir attribuer un bit pour gérer le signe du nombre.

A savoir : le choix fait est **0** pour les **nombres positifs** et **1** pour les **nombres négatifs**.

On ne s'occupera donc désormais que des nombres positifs.

Comment écrire un nombre décimal en base 2 ? On peut s'aider d'un tableau de puissances de deux.

2^i	…	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	…
3	0	0	0	1	1	0	0	0	0
5	0	0	1	0	1	0	0	0	0
4,5	0	0	1	0	0	1	0	0	0
0,125	0	0	0	0	0	0	0	1	0

En écriture binaire :
Le nombre 3 s'écrit 11.
Le nombre 5 s'écrit 101.
Le nombre 4,5 s'écrit 100,1.
Le nombre 0,125 s'écrit 0,001.

Un lien vidéo expliquant la méthode pour convertir un nombre décimal en écriture binaire :
https://www.youtube.com/watch?v=Tmg8BgiGPes
Auteur : NovelClass, durée : 11 min 29 sec.

Exemples :

Méthode à connaître :
Convertir $6{,}375_{10}$ en écriture binaire.

La partie entière est 6 soit 110_2.
Ensuite, on calcule :
0,375 x 2 = 0,75 => Valeur du bit représentant 2^{-1} : 0
0,75 x 2 = 1,5 => Valeur du bit représentant 2^{-2} : 1
0,5 x 2 = 1,0 => Valeur du bit représentant 2^{-3} : 1
La partie décimale valant 0, on stoppe les calculs ici.
On a alors : $6{,}375_{10} = 110{,}011_2$

On a vu que les nombres réels sont la plupart du temps représentés par des valeur approchées.

Convertir $0{,}1_{10}$ en écriture binaire.

La partie entière est 0.
Ensuite, on calcule :
0,1 x 2 = 0,2 => Valeur du bit représentant 2^{-1} : 0
0,2 x 2 = 0,4 => Valeur du bit représentant 2^{-2} : 0
0,4 x 2 = 0,8 => Valeur du bit représentant 2^{-3} : 0
0,8 x 2 = 1,6 => Valeur du bit représentant 2^{-4} : 1
0,6 x 2 = 1,2 => Valeur du bit représentant 2^{-5} : 1
0,2 x 2 = 0,4 => Valeur du bit représentant 2^{-6} : 0
etc.

On constate que la partie en bleu va se répéter indéfiniment : on ne peut donc pas écrire 0,1 de manière exacte en écriture binaire.

III/ Représentation d'un nombre réel dans un ordinateur

1/ Approche de la norme IEEE-754

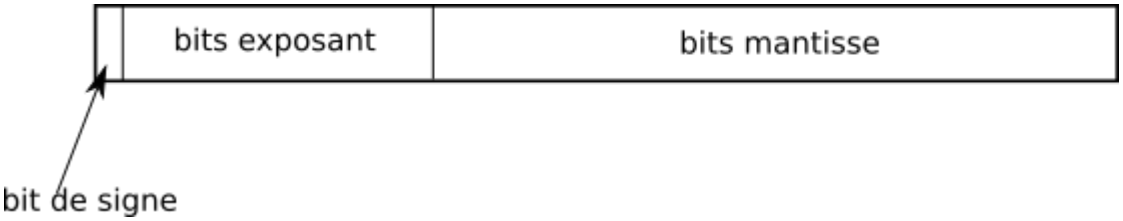
La **norme IEEE-754** est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique. La première version de cette norme date de 1985.

On s'intéressera ici à deux formats associés à cette norme : le format dit *"simple précision"* et le format *dit "double précision"*.

Le format *"simple précision"* utilise 32 bits (soient 4 octets) pour écrire un nombre flottant alors que le *format "double précision"* utilise 64 bits. (soient 8 octets). Dans la suite on travaillera principalement sur le format 32 bits.

Que cela soit en simple précision ou en double précision, la norme IEEE754 utilise :

- 1 bit de signe (1 si le nombre est négatif et 0 si le nombre est positif),
- des bits consacrés à l'exposant (8 bits pour la simple précision et 11 bits pour la double précision),
- des bits consacrés à la mantisse (23 bits pour la simple précision et 52 bits pour la double précision).



On peut vérifier que l'on a bien 1 + 8 + 23 = 32 bits pour la simple précision et 1 + 11 + 52 = 64 bits pour la double précision.

Tous ces bits sont ensuite **concaténés** (mis à la suite) et c'est ainsi que l'on représente un nombre réel dans un ordinateur.

Il faut **impérativement** que le nombre soit sous la forme **1,XXXXXXXX * 2^n** (au signe près) où X est un chiffre binaire. 1,XXXXXXXXX est appelé **mantisse** et *n* **exposant**.
Remarque : c'est le même principe que pour l'écriture scientifique.

Exemples :
$6{,}375_{10} = 110{,}011_2 = 1{,}10011 \times 2^2$.
$0{,}125_{10} = 0{,}001 = 1{,}0 \times 2^{-3}$.

Rappel : en binaire, pour multiplier par deux, on décale simplement la virgule à gauche. C'est bien sûr l'opération inverse lorsque l'on divise par deux.

2/ Représentation des nombres réels avec la norme IEEE-754

On s'intéresse au format en 32 bits (le principe est le même pour celui en 64 bits) :

- Signe du nombre : sur un seul bit, 0 pour un nombre positif et 1 pour un nombre négatif.
- Exposant : sur 8 bits donc 256 valeurs soient entre -127 et 128. Pour éviter d'avoir un exposant négatif, on va ajouter 127 artificiellement à sa valeur : on l'appelle **exposant décalé**.
  En pratique, les valeurs -127 et 128 sont réservées pour des cas particuliers (voir la fin du chapitre).
- Mantisse : c'est le nombre 1,XXXXXXXXX. Comme elle commence toujours par 1, on va l'omettre dans la représentation : on parle alors de **pseudo-mantisse**. Cela permet de gagner en précision.

Exemple :
$6{,}375_{10} = 110{,}011_2 = 1{,}10011 \times 2^2$.

Ce nombre est positif, le premier bit vaut 0.
L'exposant vaut 2 soit 127 + 2 = 129 pour l'exposant décalé. En binaire, on a $129_{10} = 1000\ 0001_2$
La mantisse vaut 1,10011 : on conserve 10011.
La représentation du nombre $6{,}375_{10}$ est alors :

Signe (1 bit)	Exposant (8 bits)	Pseudo-mantisse (23 bits)
0	1000 0001	100 1100 0000 0000 0000 0000

Remarque : il faut penser à compléter avec des « 0 » **l'exposant** (à gauche !) et la **pseudo-mantisse** (à droite !) si nécessaire pour avoir le bon nombre de bits.

On a donc : $6{,}375_{10} = 0100\ 0000\ 1100\ 1100\ 0000\ 0000\ 0000\ 0000_2$ *(Il n'y a pas d'espace en réalité, c'est juste pour faciliter la lecture.)*

Remarque : on peut aussi représenter ce nombre en base 16, cela permet d'alléger un peu son écriture.
Il vient immédiatement : $6{,}375_{10} = 40CC0000_{16}$.

On peut bien sûr retrouver un nombre réel en base 10 à partir de son écriture en binaire suivant la norme IEEE-754.

Exemple :
Quelle est la valeur en base décimale du nombre 10111110100000000000000000000000 codé selon la norme IEEE-754 en simple précision ?

En décomposant ce nombre :

- Signe du nombre (1 bit) : 1, c'est donc un nombre négatif.
- Exposant (8 bits) : $01111101_2 = 1x2^0 + 0x2^1 + 1x2^2 + 1X2^3 + 1x2^4 + 1x2^5 + 1x2^6 + 0x2^7 = 125$. Il s'agit ici de l'exposant décalé, il faut enlever 127 : il vaut donc -2.
- Mantisse (23 bits) : que des zéros, la mantisse vaut donc 1,0.

Le nombre décimal vaut donc $-1{,}0 \times 2^{-2}$ soit $-0{,}25_{10}$.

3/ Conséquences de cette représentation en programmation

La **plage des nombres flottants** pouvant être **représentés** est donc **limitée**. Pour une *« précision double »* (celle que Python utilise par défaut), on peut représenter les nombres jusqu'à environ $1{,}8 \times 10^{308}$.

D'autre part, les valeurs approchées induisent des erreurs qui peuvent être gênantes, voici une vidéo « Les nombres à virgule flottante » - Programmation Python lycée **»** : https://www.youtube.com/watch?v=DRDxK4ZGzfk
Auteur : Hatier Education, durée 5min 15 sec.

Attention donc lors de manipulations de **nombres flottants** en programmation en particulier lors de **tests d'égalités** !

Aller plus loin : la norme IEEE-754 est plus complexe que celle exposée mais cela ne fait partie du cours de NSI.

Quelques informations sur ce site : http://dictionnaire.sensagent.leparisien.fr/IEEE%20754/fr-fr/

Représentation du zéro : https://www.youtube.com/watch?v=usOqvWzKmq8