NSI en première (2019-2020)
Résumé Types simples

1 Représentation numérique de l'information

2 Nombres entiers

3 Booléens

4 Nombres réels

Nous savons représenter en machine un nombre entier si ce nombre est compris entre deux bornes qui dépendent du nombre d'octets utilisés. Qu'en est-il pour un nombre réel quelconque ? Ces nombres sont de différents types : entier comme 3, décimal comme $-4{,}25$ ou $0{,}1$, rationnel comme $1/3$, irrationnel comme $\sqrt{2}$ et π. Il y a en a une infinité même entre deux bornes quelconques. De plus pour un nombre donné, nous ne pouvons stocker dans la mémoire d'une machine qu'un nombre fini de ses décimales. Donc, même entre deux bornes comme 1 et 2, nous ne pouvons représenter qu'un nombre fini de réels de manière exacte, et pour tous les autres nous utilisons une valeur approchée. Par exemple, les nombres 3, $-4{,}25$ et $5/8$ sont représentés de manière exacte, les nombres $0{,}1$, $8/5$ et π de manière approchée.

4.1 Représentation

Nous réglons tout de suite le problème du signe. Il suffit d'un bit en machine pour coder le signe. Le choix fait est 0 pour les nombres positifs et 1 pour les nombres négatifs.

Observons l'écriture en base deux d'un nombre décimal.

2^i	$\dots$	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	$\dots$
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
$0,25$	0	0	0	0	0	0	1	0	0

Le nombre 3 s'écrit 11 soit 1,1 $\times$ 10 en binaire, ($1,5 \times 2$ en décimal).
Le nombre 5 s'écrit 101 soit 1,01 $\times$ 100 en binaire, ($1,25 \times 2^2$ en décimal).
Le nombre $4,5$ s'écrit 100,1 soit 1,001 $\times$ 100 en binaire, ($1,125 \times 2^2$ en décimal).
Le nombre $0,25$ s'écrit 0,01 soit 1,0 $\times$ 0,01 en binaire, ($1,0 \times 2^{-2}$ en décimal).
Ces quatre nombres peuvent donc s'écrire sous la forme $1,\dots \times 2^p$ en décimal. La partie $1,\dots$ s'appelle la mantisse, p est l'exposant. C'est une écriture similaire à la notation scientifique d'un nombre $m \times 10^p$ où m est un décimal appartenant à $[1;10[$. Dans notre notation la mantisse appartient à $[1;2[$ et son écriture binaire commence donc toujours par un 1. Il n'est donc pas nécessaire de coder ce 1 en machine.

Pour représenter les nombres réels, nous utilisons une valeur approchée (qui est exacte pour certains). Cette valeur approchée est écrite sous sa forme scientifique en base deux, c'est-à-dire $s\ m \times 2^p$, où s est le signe $+$ ou $-$, m la mantisse avec $1 \leq m < 2$ et p l'exposant. Un nombre réel est donc représenté en machine par un nombre décimal d'un type particulier. En Python, ce sont les nombres du type **float**.

La norme IEEE754 définit précisément le codage des "nombres en virgule flottante", "floating point numbers" en anglais, et les standards pour les représenter en machine et calculer en binaire. On représente un nombre par un signe, une mantisse et un exposant. Sur 64 bits, la règle est la suivante :

— un bit est réservé pour le signe, 0 pour le signe $+$ et 1 pour le signe $-$;
— 11 bits pour l'exposant décalé e qui vaut $p + 1023$ avec la condition $-1022 \leq p \leq 1023$, donc $1 \leq e \leq 2046$ (les valeurs 0 et 2047 sont réservées pour coder par exemple $-\infty$ ou $+\infty$) ;
— 52 bits pour la mantisse tronquée m' qui vaut $m - 1$ avec la condition $1 \leq m < 2$.

Ces trois parties sont codées en binaire et concaténées pour former un nombre de 64 bits.

Le plus grand nombre flottant a une mantisse égale en binaire à 1,111 … 1 avec 52 chiffres égaux à 1 après la virgule et un exposant égal à 1023. Sa valeur est donc $(1 + 2^{-1} + 2^{-2} + \ldots + 2^{-52}) \times 2^{1023}$ soit $2^{1023} + 2^{1022} + \ldots + 2^{971}$.

```
>>> s = 0
>>> for i in range(971, 1024):
        s += 2.0 ** i

>>> s
1.7976931348623157e+308
```

Un nombre réel compris entre ce plus grand nombre et son opposé est alors représenté par un nombre décimal, un flottant qui est la meilleure approximation possible.

Pour $0,1 = 1,6 \times 2^{-4}$, le signe est +, donc le premier bit vaut 1. L'exposant décalé vaut $-4 + 1023 = 1019$, donc l'exposant est codé par 011 1111 1011.

La mantisse tronquée s'écrit en binaire : 1001 1001 … 1001 1001 … Il s'agit de l'arrondir. Comme le 53ème chiffre est un 1, on arrondit par valeur supérieure, (1001,1 est arrondi à 1010), et donc on code : 1001 1001 … 1001 1010.

Dans l'interpréteur Python :

```
>>> (0.2 + 0.1) - 0.3
5.551115123125783e-17
>>> 2 ** (-54)
5.551115123125783e-17
```

Exemples

Voyons deux exemples de représentation en machine.

Le réel 20 s'écrit $+1,25 \times 2^4$ et le réel $-0,375$ s'écrit $-1,5 \times 2^{-2}$.

Codage du réel 20 : $20 = +1,25 \times 2^4$.

Le signe est $+$ donc le premier bit vaut 0.

L'exposant est 4 donc l'exposant décalé est $4 + 1023 = 1027$, soit 100 0000 0011 en binaire.

La mantisse est $1,25$ qui s'écrit en binaire 1,01. On garde la partie décimale 01 et on complète avec des zéros.

Le codage de 20 est donc : 0 100 0000 0011 0100 0000 … 0000.

Codage du réel -0.375 : $-0,375 = -1,5 \times 2^{-2}$.

Le signe est $-$ donc le premier bit vaut 1.

L'exposant est -2 donc l'exposant décalé est $-2 + 1023 = 1021$, soit 011 1111 1101 en binaire.

La mantisse est $1,5$ qui s'écrit en binaire 1,1. On garde la partie décimale 1 et on complète avec des zéros.

Le codage de $-0,375$ est donc : 1 011 1111 1101 1000 0000 … 0000.

Les remarques qui suivent sont utiles pour les calculs.

— L'écart entre deux nombres flottants consécutifs compris entre 1 et 2 est $\epsilon = 2^{-52}$.

— L'écart entre 2^{52} et 2^{53} est $\epsilon \times 2^{52} = 1$, donc il n'y a aucun flottant entre ces deux nombres.

— Avec un codage sur 64 bits, le plus petit nombre positif non nul est $1,0 \times 2^{-1022}$.

Tout ce qui précède concerne les nombres dit "normaux". Le mot "normal" signifie que la mantisse du nombre est égale à 1 plus la partie codée par 52 bits. Cela ne permet pas de coder 0.

On définit donc des nombres "subnormaux" pour lesquels la mantisse vaut 0 plus la partie codée par 52 bits, ceci afin de pouvoir coder 0 et d'avoir plus de flottants proches de 0. Le plus petit "subnormal" positif non nul est alors $2^{-52} \times 2^{-1022} = 2^{-1074}$.

4.2 Calculs

En Python, les nombres réels sont représentés par des nombres en virgule flottante qui sont du type `float`. L'écriture permet à Python de considérer un nombre comme étant du type `float`. Par exemple les écritures contenant un point décimal ou une puissance de 10 comme 3.14 ou 1. ou .0001 ou 1e12, définissent un nombre de type float. Ce peut être aussi le résultat d'un calcul comme une division `a/b`.

Quelques précautions

Les calculs en virgule flottante suivent les mêmes priorités que les calculs mathématiques usuels. Mais attention, en utilisant la définition des nombres flottants, on peut trouver des cas où le résultat peut surprendre. Par exemple en Python, `(1e15 + 1) - 1e15` a pour valeur 1, alors que `(1e16 + 1) - 1e16` a pour valeur 0. Pour la machine, `1e16 + 1` vaut `1e16`. Donc les calculs avec les flottants ne sont pas associatifs. On ne peut pas organiser n'importe quel calcul comme on le ferait en mathématiques en procédant à des regroupements.

De plus, il s'agit de calculs approchés et cela a des conséquences qu'il faut connaître :

— la précision est limitée et des arrondis peuvent parfois s'accumuler de manière importante ;

— il y a une plus petite valeur positive, sa valeur opposée, et entre les deux uniquement la valeur zéro, aucune autre valeur !

Par conséquent, le plus petit nombre positif divisé par 2 vaut 0 pour la machine et on ne peut pas savoir si deux nombres représentés par des flottants sont égaux ou pas. Donc, pour comparer deux nombres flottants, on ne teste pas une égalité entre les deux nombres. On vérifie s'ils sont suffisamment proches, ce "suffisamment" dépendant du contexte, en contrôlant si la valeur absolue de leur différence est inférieure ou égale à une précision donnée.

Le langage Python, gère des entiers très grands et permet de faire des calculs exacts avec des nombres qui dépassent la valeur maximale permise par le nombre de bits utilisés dans le codage de ces nombres. Ces calculs sont décomposés, peuvent nécessiter un temps non négligeable, mais ils sont exacts ! Pour les flottants, ce n'est pas le cas, il y a un plus grand nombre.

```
>>> 1.0e308
1e+308
>>> 1.0e309
inf
>>> 2.0**1023
8.98846567431158e+307
>>> 2.0**1024
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    2.0**1024
OverflowError: (34, 'Result too large')
```

Lorsque dans un calcul la valeur maximale est dépassée, le programme s'arrête et une erreur est signalée : `OverflowError`. Le programme doit être modifié. Cela peut avoir des conséquences graves. L'autodestruction de la fusée Ariane 5, lancée le 4 juin 1996, 39 secondes après le décollage a été causée par un problème d'overflow !

Lors de la mission Apollo 11 dans les années 1970, le module qui devait se poser sur la lune rencontra de graves problèmes. L'ordinateur embarqué était saturé et lançait des alarmes. L'alunissage se déroula quand même sans dégât grâce à la qualité de la programmation des logiciels embarqués dont Margaret Hamilton était responsable à la NASA. L'ordinateur était programmé pour se concentrer sur les tâches les plus prioritaires.