Numération

Partie 3 : représentation des flottants

Objectifs :

Représentation approximative des nombres réels : notion de nombre flottant			
Calculer sur quelques exemples la représentation de nombres réels : 0.1, 0.25 ou 1/3.			
0,2 + 0,1 n'est pas égal à 0,3. Il faut éviter de tester l'égalité de deux flottants.			

I Partie entière- partie décimale

A En base 10

Un nombre décimal exprimé dans la base 10 est constitué de deux parties, séparées par une virgule :

- **Une partie entière :** elle est située avant la virgule. Il s'agit de la décomposition d'un nombre en puissance de 10 positives.
  Par exemple : $11 = 1 \times 10^1 + 1 \times 10^0$.
- **Une partie décimale** : elle est située après la virgule. Il s'agit de la décomposition d'un nombre en puissance de 10 négatives.
  Par exemple : $0{,}8125 = 8 \times 10^{-1} + 1 \times 10^{-2} + 2 \times 10^{-3} + 5 \times 10^{-4}$.

B Une première adaptation au binaire

Dans un premier temps, on peut imaginer le nombre binaire $1011{,}1101_2$.

- La partie entière est située avant la virgule : 1011. C'est une décomposition en valeurs positives des puissances de 2.

nombre binaire $= 1011_2$	1	0	1	1	
puissance de 2 correspondante	2^3	2^2	2^1	2^0	
	8	4	2	1	
	8		2	1	$= 11$

- La partie décimale est située après la virgule : 1101. C'est une décomposition en valeurs négatives des puissances de 2.

nombre binaire $= 1101_2$	1	1	0	1	
puissance de 2 correspondante	2^{-1}	2^{-2}	2^{-3}	2^{-4}	
	0,5	0,25	0,125	0,0625	
	0,5	0,25		0,0625	$= 0,8125$

On a donc $1011{,}1101_2 = 11{,}8125_{10}$.

II Une première écriture

/!\ le problème est que le caractère « , » n'est pas autorisé dans le langage binaire. Seuls 0 et 1 sont autorisés. Il faut utiliser une nouvelle norme.

A La notation scientifique des décimaux

En sciences, pour représenter les nombres décimaux, on utilise la **notation scientifique**.

Un nombre est en notation scientifique s'il s'écrit sous la forme $\pm a \times 10^n$

- $\pm$ est appelé le *signe*
- a est appelé la *mantisse* où $a \in [1;10[$. a s'écrit donc avec un chiffre non nul avant la virgule
- $n \in \mathbb{Z}$ est appelé l'*exposant*.

Par exemple

$11,8125 = +1,18125 \times 10^1$

Le signe est +, la mantisse est 1,18125 et l'exposant est 1.

B En binaire

Pour coder les nombres flottants en binaire, on va utiliser la même idée.

- Un signe
- Une mantisse qui commence par un 1
- Un exposant entier.

Ainsi on va avoir par exemple

- $1011,1101 = +1,0111101 \times 2^3$
  Le signe est +, la mantisse est 1,0111101 et l'exposant 3 car on a décalé la virgule de 3 rangs vers la droite.
- $-0,01011 = -1,011 \times 2^{-2}$
  Le signe est $-$, la mantisse est 1,011 et l'exposant est -2 car on a décalé la virgule de 2 rangs vers la gauche.

En utilisant cette norme on a nécessairement 1 comme chiffre le plus à gauche et on peut sous-entendre la virgule qui vient juste après. Il reste à voir comment on procède dans une machine où la place mémoire attribuée pour représenter un flottant est définie à l'avance. Avant voyons comment convertir en binaire la partie décimale d'un nombre.

C Conversion base 10 -binaire d'un nombre décimal

La conversion est différente pour la partie entière et pour la partie fractionnaire. Ainsi, pour convertir 11,75, on fait :

- Pour la partie entière, nous avons vu la méthode des divisions successives par 2.
  Par exemple on a $11 = 1011_2$
- Par la partie décimale on effectue des multiplications successives par 2 en conservant à chaque fois la partie entière du produit (0 ou 1) jusqu'à ce qu'il reste 0.

$0,75 \times 2 = 1,5 \rightarrow 1$
$0,5 \times 2 = 1 \rightarrow 1$ donc $0,75 = 0,110_2$
$0,0 time 2 = 0,0 \rightarrow 0$

Faisons de même avec le nombre décimal 0,578125.

$0,578125 \times 2 = 1,15625 \rightarrow 1$
$0,15625 \times 2 = 0,3125 \rightarrow 0$
$0,3125 \times 2 = 0,625 \rightarrow 0$
$0,625 \times 2 = 1,25 \rightarrow 1$ d'où l'on en déduit : $0,578125 = 0,10010100_2$
$0,25 \times 2 = 0,5 \rightarrow 0$
$0,5 \times 2 = 1,0 - \rightarrow 1$
$0,0 \times 2 = 0,0 \rightarrow 0$

/!\ Un nombre à développement décimal fini en base 10 ne l'est pas forcément en base 2.

$0,8 \times 2 = 1,6 \rightarrow 1$
$0,6 \times 2 = 1,2 \rightarrow 1$
$0,2 \times 2 = 0,4 \rightarrow 0$
$0,4 \times 2 = 0,8 \rightarrow 0$
$0,8 \times 2 = 1,6 \rightarrow 1$
...

On constate qu'il ne restera jamais 0. 0,8 n'a pas de développement décimal fini en base 2. En binaire il est donc impossible de coder le nombre 0,8 avec un nombre fini de bits. Par exemple en Python on peut afficher, à l'aide du module decimal, la valeur telle qu'elle est stockée en mémoire :

```
>>>from decimal import Decimal
>>>Decimal(0.8)
 Decimal('0.8000000000000000444089209850062616169452667236328125')
```

On constate que ce n'est pas exactement 0,8.

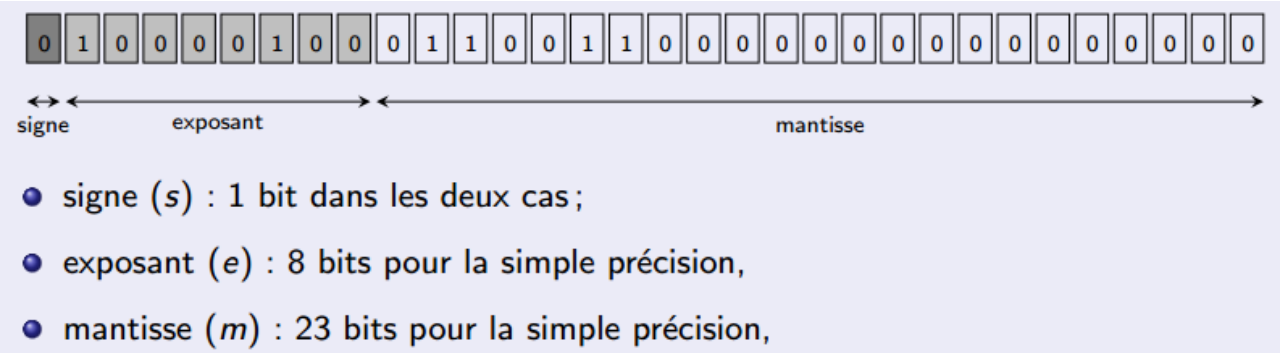
Il en est de même pour les nombres 0,1 et 0,2, ce qui explique pourquoi, toujours en Python :

```
>>> 0.1 + 0.2 == 0.3
False
```

III La norme IEEE 754 (simple précision – 32 bits)

La norme IEEE (Institute of Electrical and Electronics Engineers) 754 est une norme sur l'arithmétique à virgule flottante. Elle est la norme la plus employée actuellement pour le calcul des nombres à virgule flottante dans le domaine informatique. Python, comme d'autres langages, utilise la double précision sur 64 bits, mais nous n'en parlerons pas. En simple précision (32 bits), elle indique dans l'ordre :

- 1 bit de signe,
- 8 bits d'exposant (−126 à 127) : le décalage vaut donc exposant – 127,
- 24 bits de mantisse. Le premier bit à 1 est implicite, c'est-à-dire qu'il n'est pas représenté.



Le nombre en base 10 correspondant est : $(-1)^{signe} \times 2^{exposant+127} \times 1.mantisse$

Exemple : représentation de 11,75

$11{,}75_{10} = 1011{,}11_2$, soit $1{,}01111_2 \times 2^3$ avec un décalage de +3.

- signe = 0 (nombre positif)
- exposant = $127 + 3 = 130_{10} = 10000010_2$
- mantisse = 01111 : le premier 1 n'est pas représenté

$11{,}75_{10}$ = 0 10000010 01111000000000000000000

Il existe des valeurs spéciales qui représentent le dépassement de capacité (overflow) `inf` et les opérations non valides `nan` (Not A Number). Que l'on peut voir par ces exemples en Python :

```
>>>x = 1e200
>>> x * x
inf
>>> x * x * 0
Nan
```

Les conversions implicites peuvent aussi provoquer des erreurs :

```
>>> y = 10 ** 400 # y est un entier
```

```
>>> y + 0.5 # Tentative de conversion implicite d'entier à
flottant
OverflowError: int too large to convert to float
```

Les arrondis posent d'autres difficultés. Par exemple, un calcul simple montre les imprécisions du calcul :

```
>>> 1.2 * 3
3.5999999999999996
```

L'associativité de l'addition n'est pas respectée :

```
>>> Decimal(1.6+(3.2+1.7))
Decimal('6.5')
>>>Decimal((1.6+3.2)+1.7)
 Decimal('6.500000000000000888178419700125232338905334472656 25')
```

La distributivité de la multiplication n'est pas respectée :

```
>>> Decimal(1.5*(3.2 + 1.4))
Decimal('6.899999999999999467092948179924860596656799316406 25')
>>>Decimal(1.5*3.2+1.5*1.4)
Decimal('6.900000000000000355271367880050092935562133789062 5')
```

On constate qu'il est imprudent de faire des programmes qui testent des égalités entre flottants.

On retiendra cet exemple :

```
>>> 0.1 + 0.2 == 0.3
False
```

Pour résoudre ce problème, on fera plutôt un test de comparaison :.

```
>>> x = 0.1+0.2
>>> y = 0.3
>>>abs(x-y) <1e-12
True
```

On considère ici que les deux nombres sont égaux quand la valeur absolue de leur différence est inférieure à 10^{-12} .

En simple précision, on peut approcher les nombres compris entre -3,402 823 46 × 10^{38} et 3,402 823 46 × 10^{38} .

On retient que les nombres flottants sont une représentation approximative des nombres réels. La norme IEE 754 définie un encodage et des règles d'arrondi. Les opérations sur les flottants n'ont pas toujours les mêmes propriétés que les mêmes opérations sur les réels. De plus, il est dangereux de comparer des flottants.

Pour en savoir plus :
https://fr.wikipedia.org/wiki/IEEE_754