I_{01} Représentation de l'information

PCSI 2024 – 2025

But du chapitre : l'objectif du chapitre est d'avoir quelques notions de « comment les nombres sont représentés en machine » pour comprendre les conséquences pour le programmeur.

I Comment comptons-nous ?

1. Nombre entiers

Nous représentons généralement un nombre sous la forme de plusieurs chiffres (dits arabes) mis les uns à la suite des autres, par exemple 3507. Cette représentation à l'aide de dix chiffres (0, 1, 2, ... , 9) signifie :

$$3000 + 500 + 00 + 7 = 3 \times 10^3 + 5 \times 10^2 + 0 \times 10^1 + 7 \times 10^0.$$

On dit à cause de cela que le nombre est représenté en base 10 . Lorsqu'il y aura un doute sur la base utilisée, elle sera indiqué en indice du nombre, par exemple 3507_{10}.
Pour représenter des entiers relatifs, on se contente de rajouter un signe moins devant un nombre représenté en base 10 : -357.

2. Nombres réels

Pour un nombre réel, on procède de façon similaire, mais en rajoutant des chiffres après une virgule, par exemple : $18{,}53$. Ce nombre est lui aussi représenté en base 10 et signifie :
$10 + 8 + 0.5 + 0.03 = 1 \times 10^1 + 8 \times 10^0 + 5 \times 10^{-1} + 3 \times 10^{-2}$.
Parmi les nombres réels, on peut distinguer ceux qui peuvent s'écrire avec un nombre fini de chiffres après la virgule[1], appelés nombres décimaux $(3\,;\,3{,}14)$, de ceux qui ne le peuvent pas $(1/3\,;\,\pi$ par exemple).

Remarque : En France, le séparateur décimal est la virgule. Dans les pays anglo-saxons (et donc en informatique), le séparateur décimal est le point.

II Comment compte un ordinateur ?

1. Nombre de chiffres disponibles

Pour nous : pour pouvoir compter en base 10, il nous faut dix chiffres différents. Si nous voulons représenter les nombres en base 16, il nous faut alors seize chiffres différents. Cette base, appelée hexadécimale, est fréquemment utilisée en informatique et fait appel aux chiffres 0, 1, ..., 9, A, B, C, D, E, F . Le nombre $B2F_{16}$ représente donc en base 10 : $11 \times 16^2 + 2 \times 16^1 + 15 \times 16^0 = 2863$.

Remarque : Pour rentrer un nombre en hexadécimal en python, il faut le préfixer par $0x$, à l'inverse, pour afficher la représentation hexadécimale d'un nombre, il existe la fonction hex.

Pour un ordinateur : Au niveau du processeur[2] avec les technologies actuelles, un ordinateur dispose de deux chiffres : 0, 1 . Il devra donc nécessairement utiliser la base deux.

1. Sous-entendu en base 10. Si l'on change de base, les nombres pouvant s'écrire avec un nombre fini de chiffres après la virgule changent : par exemple en base 3, le nombre $1/3$ s'écrit simplement $0{,}1$. Ce point fait que certains nombres décimaux qui nous semblent facile à appréhender n'auront pas une représentation exacte en machine.
2. Ce n'est pas tout à fait vrai pour certaines technologies de stockages, comme pour les SSD MLC/TLC/QLC où l'on peut considérer que l'on dispose de 4, 8 ou 16 chiffres.

2. Représentation des entiers naturels

Le principe des différentes bases a déjà été vu, la base pertinente est ici la base deux.

Exemple : 101_2

Représentation en base 2 :	1	0	1
Puissance de deux :	2^2	2^1	2^0

$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$

Remarque : Les nombres en binaires sont parfois soulignés pour indiquer qu'ils sont en binaire : $1011_2 = \underline{1011}$

Une fois en mémoire, tous les nombres sont les uns à la suite des autres : comment savoir si on regarde la suite de chiffres 10011101011100010101111 s'il s'agit d'un seul nombre ou de plusieurs mis bout à bout ? Pour palier à ce problème, on utilise des nombres représentés par un nombre de chiffres fixé à l'avance, on dit aussi un nombre de bits.

Dans la mémoire des ordinateurs, les informations sont souvent rassemblés par groupe de huit bits : les octets (bytes en anglais). Les nombres manipulés sont généralement représentés par un, deux, quatre ou huit octets, c'est-à-dire 8, 16, 32 ou 64 bits.

Exercice : quel est le nombre d'entiers que l'on peut représenter à l'aide d'un octet ?
Il s'agit d'un exercice de dénombrement. Pour le premier chiffre, on a deux choix. Pour le deuxième chiffre, on a aussi deux choix, indépendant du choix précédent, et ainsi de suite, on a donc $2 \times 2 \times 2 \cdots \times 2 = 2^8$ nombres possibles, c'est-à-dire que l'on peut représenter les nombres de 0 à $2^8 - 1 = 255$ par exemple (ou 1 à 256, etc...)

Remarques :

- Lorsque l'on parle de processeur 32 bits ou 64 bits, il s'agit du nombre de chiffres utilisé par le processeur pour représenter les adresses mémoires.
- On n'a parlé ici que d'entiers naturels, ce type d'entier en mémoire est dit non signé . Il existe dans certains langages de programmation, mais pas en python qui ne manipule que des entiers relatifs.

3. Entiers relatifs

Pour représenter les entiers relatifs, une solution pourrait être de consacrer un des bits au signe (0 pour $+$ et 1 pour $-$ par exemple), mais si on faisait ça, il y aurait deux manières de représenter 0 : 1000 0000 et 0000 0000 pour des nombres représenté sur un octet (8 bits). Un autre solution a été adopté : le complément à deux.
Il s'agit d'utiliser un entier naturel pour représenter un entier relatif. Par exemple sur 8 bits, par convention les nombres compris entre 0 et 2^7 (exclu) seront les nombres positifs, et les nombres entre 2^7 et $2^8 - 1$ seront les nombres négatifs en leur soustrayant 2^8. Ainsi le nombre 1111 1111 représente le nombre -1_{10}. Le premier chiffre correspond donc toujours au signe, mais les sept chiffres suivant ne représentent plus la valeur absolue.

Définition : Représentation en binaire sur n bits d'un entier relatif : Si l'entier relatif x est positif ou nul, on le représente comme l'entier naturel x . S'il est strictement négatif, on le représente comme l'entier naturel $x + 2^n$.

Exemples (toujours avec un entier codé sur 8 bits) :

Représentation	Nombre en base 10
0000 0001	1
0000 0000	0
0001 0001	17
0100 0000	$2^6 = 64$
0111 1111	$2^7 - 1 = 127$
1000 0000	$2^7 - 2^8 = -128$
1001 0000	$2^7 + 2^4 - 2^8 = -112$
1111 1111	$2^8 - 1 - 2^8 = -1$
1111 1101	$2^8 - 1 - 2 - 2^8 = -3$

On peut donc représenter tous les nombres entre -2^7 et $+2^7 - 1$ en codant sur 8 bits et de façon plus générale, en codant ainsi sur n bits, on peut représenter tous les nombres entiers entre -2^{n-1} et $+2^{n-1} - 1$.
Exemple : nombre codé sur 64 bits : -2^{63} et $+2^{63} - 1 \simeq 10^{19}$.

Remarque : Cette façon de noter les nombres présente l'avantage de faciliter l'addition, la soustraction et la comparaison des nombres pour l'ordinateur.

4. Conséquences

- On ne peut pas représenter des nombres en dehors de l'intervalle $[-2^{n-1}; +2^{n-1} - 1]$
- Les calculs sur les nombres entiers sont des calculs exacts tant que le résultat est dans l'intervalle ci-dessus. (Par opposition aux nombres « réels » aux prochains paragraphes)

Définition : On parle de dépassement arithmétique ou d' overflow lorsque le résultat d'un calcul est en dehors de l'intervalle disponible pour les nombres considérés. Le résultat est alors différent du résultat attendu.

Exemple : Si dans un programme en C avec des nombres codés sur 8 bits on réalise l'opération $126 + 123$, on obtient le résultat suivant : -7. Expliquez cela. Simulation en C

$$\begin{array}{l} 0111\,1110\ (126_{10}) \\ \underline{+\ 0111\,1011\ (123_{10})} \\ = 1111\,1001\ (249_{10} = 256_{10} - 7_{10}) \end{array}$$

Remarque : En python, les entiers sont gérés de façon à éviter les problèmes d'overflow, (le nombre de bits allouer au résultat augmente si celui ci est trop grand ou trop petit), ce problème ne se présente pas de cette façon. Toutefois, lorsque les entiers dépassent une certaine taille, le calcul est considérablement ralenti. Ceci est du au fait que le processeur doit procéder à plusieurs calculs pour faire l'addition.

5. Nombres à virgule flottante

De la même façon qu'en base 10, on peut représenter un nombre réel en base deux. Ainsi,
$10{,}11_2 = 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 2{,}75$.
On pourrait choisir d'allouer un certain nombre de bits après la virgule et un certain nombre avant la virgule. Mais en procédant ainsi, la plage de nombres accessibles serait assez réduite : on choisit donc de se placer en notation scientifique.

Définition : La notation scientifique est la représentation d'un nombre décimal de forme $\pm a \times 10^n$ où $\pm$ est le signe, a est un nombre réel compris dans l'intervalle $[1{,}10[$ et appelé mantisse et n est un entier relatif appelé exposant. Par extension, en base 2, la notation scientifique est la représentation de la forme $\pm m \times 2^e$ où la mantisse m est dans l'intervalle $[1{,}2[$.

Astuce : Quels sont les chiffres que l'on peut mettre à gauche de la virgule en notation scientifique en base 10 ? 1, ..., 9 Et en base 2 ? seulement 1, il est donc inutile de le coder et il sera sous-entendu.

Un nombre réel sera donc représenté de la forme suivante : un triplet $[s,e,m]$. Lorsque l'on utilise 64 bits pour un nombre, 1 bit est utilisé pour le signe, 52 pour la mantisse et 11 pour l'exposant.

- Le premier bit représente le signe. $+$ est représenté par 0 et le signe $-$ par 1 , donc on peut écrire $(-1)^s$ si s est le bit de signe
- Les 11 bits suivants représentent l'exposant e. C'est un entier relatif compris entre -1022 et 1023 et on le représente par l'entier naturel $n = e + 1023$ (compris entre 1 et 2046, 0 et 2047 sont réservé pour des cas particuliers, dont les NaN : Not a Number et l'infini).
- Les 52 bits restant représentent la mantisse. Ce sont les 52 chiffres après la virgule en notation scientifique en base 2, le chiffre avant la virgule étant nécessairement 1.

Les nombres représentés ainsi sont appelés nombre à virgule flottante.

Exemples : Lien convertisseur binaire

- Quel nombre flottant est représenté par cette succession de bits :
  0100000000101100000000000000000000000000000000000000000000000000 ?
  $\underbrace{0}_{s}\ \underbrace{10000000010}_{e}\ \underbrace{1100000000000000000000000000000000000000000000000000}_{m}$
  Le signe est 0, donc il s'agit d'un nombre positif.
  L'exposant est $10000000010_2 = 1026_{10}$ donc il code pour le nombre $1026 - 1023 = 3$.
  La mantisse est 11000...0, le nombre est donc en base 2 : $1{,}11000...0 \times 2^3$ (ne pas oublier le 1 sous-entendu avant la virgule), il s'agit donc du nombre

$$+(1 + \frac{1}{2} + \frac{1}{4}) \times 2^3 = 14_{10}$$

- 1111111111100100000000000000000000000000000000000000000000000000 ?
  $\underbrace{1}_{s}\ \underbrace{11111111110}_{e}\ \underbrace{0100000000000000000000000000000000000000000000000000}_{m}$
  $s = 1 \Rightarrow -$ $\underbrace{11111111110}_{e} \Rightarrow 2^{2046-1023} = 2^{1023}$ $\underbrace{010...0}_{m} \Rightarrow 1{,}01_2 = 1{,}25_{10}$
  donc il s'agit du nombre $-1{,}25 \times 2^{1023} \simeq -1{,}12356 \times 10^{308}$
- Comment représenter le nombre $0{,}75$?
  Il s'agit de $0.5 + 0.25 = (1 + 0.5) \times \frac{1}{2} = +1{,}1_2 \times 2^{-1}$ donc
  $s = 0$ $e = 01111111110$ $m = 10000000...000$ (premier 1 sous-entendu)
  donc : $\underbrace{0}_{s}\ \underbrace{01111111110}_{e}\ \underbrace{1000000000000000000000000000000000000000000000000000}_{m}$

Par convention, un nombre dont la mantisse et l'exposant valent 0 représente 0 . Il y a donc un 0 positif (0000...000) et un 0 négatif (1000...000).

III Conséquence de la représentation en nombre flottant

1. Dépassement de capacité

De la même façon que pour les entiers relatifs, il est possible d'avoir des dépassement arithmétique « overflow » si le résultat du calcul est trop grand. Par exemple :

```
>>>10.0**308+10.0**308
inf #Trop grand
>>>inf/inf
nan #forme indéterminée
```

Par contre, il est aussi possible d'avoir des soupassement arithmétique (« underflow ») si le résultat des calculs est trop proche de 0. Par exemple :

```
>>> 1e-200*1e-30
1e-230
>>> 1e-300*1e-30
0.0
```

2. Arrondis

On sait en base 10 qu'il n'est pas possible de représenter le nombre $\frac{1}{3} = 0.333333333....$ avec un nombre fini de chiffres. Le même genre de cas se produit en base deux. Par exemple le nombre $1{,}6_{10}$ peut s'écrire sous la forme suivante :

$$\frac{1}{2^0}+\frac{1}{2^1}+\frac{1}{2^4}+\frac{1}{2^5}+\frac{1}{2^8}+\frac{1}{2^9}+\frac{1}{2^{12}}+\frac{1}{2^{13}}+...=1{,}1001100110110011..._2$$

Il n'est donc pas possible de le représenter de façon exacte avec une mantisse de 52 chiffres après la virgule et il est nécessaire d'avoir recours à des arrondis.

2.a. Précision des calculs

À cause des arrondis, certains calculs peuvent avoir des résultats inattendus.

Exemple en base 10 : On suppose que l'on sait écrire les nombres en notation scientifique en base 10, mais que l'on n'est capable de ne garder en mémoire que 4 chiffres significatifs (donc 3 après la virgule). On dispose de deux nombres : $a = 1{,}547 \times 10^2$ et $b = 9{,}430 \times 10^{-4}$. Quel sera alors le nombre $c = a + b$ une fois enregistré dans la mémoire ?
$a+b = 1547009{,}430 \times 10^{-4} = 1{,}547009430 \times 10^2$ et donc une fois stocké en mémoire, $c = 1{,}547 \times 10^2 = a$
Exemple en base 2 avec des flottants 64 bits :

$$1+2^{-55}=1,\underbrace{0000000000000000000000000000000000000000000000000000}_{52\text{ chiffres}}001_2$$

Mais le résultat sera arrondi car la mantisse n'est représenté que par 52 chiffres après la virgule, le résultat en mémoire sera donc le nombre

$$1,\underbrace{0000000000000000000000000000000000000000000000000000}_{52\text{ chiffres}}$$

C'est-à-dire exactement 1 . Même si on ré-itère l'opération un grand nombre de fois, le résultat sera toujours 1 :

```
a=1.0
for ii in range(100000000):
    a=a+2.0**-54
print(a)
```

Donne pour résultat `1.0`

```
a=1.0
b=0.0
for ii in range(100000000):
    b=b+2.0**-54
a=a+b
print(a)
```

Donne pour résultat `1.00000000555`

Exercice : estimer à combien de chiffres en base 10 environ correspond 52 chiffres après la virgule en base 2 ?
Il suffit de regarder 2 nombres « consécutifs » d'exposant 0. On change juste le dernier chiffre, par exemple $1{,}000...001 - 1{,}000...000 = 0{,}000...001 = 2^{-52} \simeq 2{,}2 \times 10^{-16}$. Cela correspond donc à 16 chiffres après la virgules environ (le 16^e n'est pas sur)

2.b. Non associativité de l'addition

```
>>> 1.0+(-1.0+10.0**-20)
0.0
>>> (1.0-1.0)+10.0**-20
1e-20
```

On voit sur cet exemple qu'en informatique, on n'a pas nécessairement $(a+b)+c = a+(b+c)$ à cause des problèmes d'arrondis : l'addition n'est pas associative.

2.c. Non pertinence du test d'égalité entre nombre à virgule flottante

```
>>> 0.99999999999999999==1
True
>>> 0.1+0.1+0.1==0.3
False
>>> 0.3-(0.1+0.1+0.1)
-5.551115123125783e-17
```

Autrement dit pour l'ordinateur, le nombre $0{,}99999999999999999$ et le nombre 1 sont égaux . De même le résultat du calcul $0{,}1+0{,}1+0{,}1$ n'est PAS le nombre $0{,}3$ et l'ordinateur calcule une différence de l'ordre de 10^{-17} entre ces deux nombres. Compte tenu des arrondis lors des calculs, il sera donc généralement pas pertinent de tester si deux nombres flottants sont égaux. On préfèrera tester si la différence entre les deux nombre est plus petite en valeur absolue qu'une précision ϵ définie aupartavant.

Table des matières