Informatique

Représentation des nombres

Partie 2

Représentation des nombres en machine

Cours

1 Nombres entiers naturels.

Comme nous l'avons vu, un entier naturel codé sur 8 bits a une valeur comprise entre 0 et 255

De manière générale, un entier naturel codé sur n bits a une valeur comprise entre 0 et $2^n - 1$.

Dans une image Python, les valeurs R, G, B sont des entiers sont codés sur 8 bits, et donc sont compris entre 0 et 255 : en les additionnant ou en les manipulant sans précaution, on risque d'avoir des erreurs « d'overflow ».

Dans des calculs « classiques » en Python, les entiers naturels sont codés dans le type « int » : dans ce type, le nombre de bits utilisés pour stocker un entier varie en fonction de la taille de cet entier.

L'entier 1 est codé sur 1 bit

Les entiers 2 et 3 sont codés sur 2 bits

Les entiers 4,5,6,7 sont codé sur 3 bits

Etc.

Une conséquence importante de la représentation des entiers dans Python est l'évaluation de la complexité des opérations sur les entiers. En effet, sur une représentation de taille fixe, la complexité des opérations arithmétiques est évaluable en fonction du nombre de bits utilisés. En Python, le nombre de bits étant variable, la complexité n'est pas évaluable simplement.

2 Nombres entiers relatifs

Nous allons maintenant voir deux méthodes pour représenter des entiers relatifs sur n bits. Le principe consiste à associer à tout entier entier relatif $Z \in [-2^{n-1}, 2^{n-1} - 1]$, un entier naturel N compris entre 0 et $2^n - 1$ (donc codé sur n bits).

2.1 Méthode naïve

Le principe consiste à choisir un bit qui représentera le signe de l'entier relatif codé Z et les $n - 1$ bits restants coderont la valeur absolue de Z.

Une possibilité serait de choisir le premier bit (poids fort) comme bit de signe, par exemple $\begin{cases} 0 \; si \; Z > 0 \\ 1 \; si \; Z < 0 \end{cases}$ et de prendre le reste pour coder $|Z|$. Dans ce système, on aurait, sur 4 bits :

$$\begin{cases} (\mathbf{0}101)_2 = +(101)_2 = +(5)_{10} \\ (\mathbf{1}001)_2 = -(001)_2 = (-1)_{10} \end{cases}$$

Ce choix conduit, d'une part à deux représentations de 0 ($0^+ = 0000$ et $0^- = 1000$), mais surtout, ne permet pas de garder les propriétés de la somme en binaire. En effet, $5 + (-1) = 4$ et 4 se code dans cette représentation : $(\mathbf{0}100)_2$, ce qui ne correspond pas du tout à la somme, en binaire de 0101 et 1001 !

2.2 Complément à 2

Définition : La représentation par complément à 2 sur n bits d'entier $Z \in \mathbb{Z}$ consiste à coder en binaire, sur n bits l'enter N défini par :

$$N = \begin{cases} Z \; si \; Z \geq 0 \\ Z + 2^n \; si \; Z < 0 \end{cases}$$

Par exemple, sur 3 bits ($n = 4$), on pourra coder les entiers relatifs de $-2^2 = -4$ à $2^2 - 1 = 3$. On a alors la correspondance suivante :

Z	-4	-3	-2	-1	0	1	2	3
N	4	5	6	7	0	1	2	3
$Codage\ en\ binaire$	$(100)_2$	$(101)_2$	$(110)_2$	$(111)_2$	$(000)_2$	$(001)_2$	$(010)_2$	$(011)_2$

On remarquera que le premier bit est toujours un bit de signe et qu'il n'y a qu'une représentation de 0.

Le gros avantage de ce choix tient dans le fait que la somme binaire d'entiers fonctionne toujours entre des entiers relatifs, en restant sur n bits (overflow si on dépasse une taille de n, c'est-à-dire ne pas tenir compte du bit qui apparaît à gauche) :

$$(-3)_{10} + (3)_{10} \leftrightarrow (101)_2 + (011)_2 = (\cancel{1}000)_2 \leftrightarrow (000)_2 = 0$$

$$(-3)_{10} + (1)_{10} \leftrightarrow (101)_2 + (001)_2 = (110)_2 \leftrightarrow (-2)_{10}$$

***Exercice 1** : Sur 8 bits, quels entiers relatifs peut-on représenter ? Donner la représentation de -41.*

Remarque : il existe d'autres méthodes pour obtenir ce codage binaire des nombres négatifs. Par exemple, on peut commencer par écrire le codage binaire de la valeur absolue sur n bits, puis garder tous les 0 à partir de la droite, ainsi que le premier 1 rencontré puis à inverser tous les autres bits.

$(42)_{10} = 101010$ → on l'écrit sur 8 bits $42 = (00101010)_2$ → on inverse tous les bits à partir du bits qui suit le premier 1 en partant de la droite : $-42 = (11010110)$

3 Nombres à virgule flottante

Dans cette partie, nous allons voir comment on peut représenter des nombres réels en machine. L'idée est de stocker de la façon la plus efficace possible des nombres réels sur un nombre de bits fixés. Dans la plupart des systèmes actuels, on stocke les nombres réels sur 64 bits.

3.1 Approche « naïve » : représentation en virgule fixe.

L'idée de cette représentation est simple : réserver une partie fixe des bits pour la partie entière et une autre pour la partie fractionnaire. Mais un tel système aurait un inconvénient majeur : le nombre de bits nécessaire pour coder de grands nombres devient vite très important. Par exemple, si l'on veut stocker les réels sur 64 bits en réservant la moitié des bits pour la partie entière et l'autre pour la partie fractionnaire (sans oublier un bit pour le signe), le plus grand nombre stockable sera : $2^{32} \approx 4{,}3 \times 10^{9}$ (et le plus proche de zéro sera $2^{-31} \approx 4{,}7 \times 10^{-10}$), ce qui est très limité.

3.2 Représentation en virgule flottante : les flottants.

3.2.1 Principe

La représentation en virgule flottante est l'équivalent de l'écriture scientifique, mais en binaire.

Pour bien comprendre, prenons l'exemple de l'écriture scientifique de $-289456{,}5 = -2{,}894565.10^{5}$.

Voici le vocabulaire qui sera utilisé en « virgule flottante » :

$-2{,}894565.10^{5}$			
$-$	2	894565	5
Signe	Caractéristique	Mantisse	Exposant
	Partie significative		

Remarques :

- Le signe est codé par 0 (pour $+$) ou 1 (pour $-$)
- La caractéristique est un chiffre de 1 à 9, elle ne peut pas être nulle car dans ce cas, la notation scientifique n'est pas respectée : $\mathbf{0},022 = \mathbf{0},22.10^{-1} = 2{,}2.10^{-2}$

En binaire, on va utiliser le même principe mais avec des puissances de 2 au lieu de puissance de 10 :

$$(-2{,}894565.10^{5})_{10} = -(100011010101010110000{,}1)_{2}$$

Ce nombre s'écrit sous forme scientifique binaire comme suit :

$$-(100011010101010110000{,}1)_{2} = -(1{,}00011010101010110000 1)_{2}.2^{18}$$

Pour stocker ce nombre, on a donc besoin des informations suivantes :

$-$	1	00011010101010110000 1	18
Signe	Caractéristique	Mantisse	Exposant
	Partie significative		

Comme précédemment, la caractéristique ne peut être nulle. En base 10, elle pouvait valoir un chiffre de 1 à 9. Maintenant, elle ne peut plus être différente de 1. Plus besoin de la stocker : on gagne un bit !

Finalement, il suffit de stocker les informations suivantes :

$-$	$00011010101011000 1$	18
Signe	Mantisse	Exposant

Ces informations ne peuvent être stockées directement que sous forme binaire, avec des 0 et des 1. (signe et puissance en base 10).

Les choix suivants sont effectués pour les flottants codés sur 32 bits :

- Le signe est stocké sur 1 bit : 0 pour $+$; 1 pour $-$.
- La mantisse est stockée sur 52 bits : on ajouter des
- L'exposant est stocké sur 11 bits. L'exposant est un nombre entier qui peut être positif ou négatif. On aurait donc pu le stocker par un complément à 2, mais le choix est fait de le coder en binaire par décalage.
  Le principe est le suivant : sur 8 bits, on peut stocker $2^8 = 2 \times 2^7$ entiers : de $-(2^7 - 1)$ à 2^7.
  - $-(2^7 - 1) = -2^7 + 1$ est codé par $(00000000)_b$
  - $-(2^7 - 1) = -2^7 + 2$ est codé par $(00000001)_b$
  - $-(2^7 - 1) = -2^7 + 3$ est codé par $(00000010)_b$
  - …
  - 2^7 est codé par $(11111111)_b$

  Cela revient à ajouter $2^7 - 1 = 127$ puis à coder l'entier naturel obtenu en binaire.

$$18 + 127 = (10010001)_b$$

Finalement, en 32 bits, on aura une représentation de $-289456{,}5$ par :

Représentation sur 32 bits	1	10010001	00011010101011000010000
	Signe (1 bit)	Puissance (8 bits)	Mantisse (23 bits)
$(-289456{,}5)_{10} \leftrightarrow 11001000100011010101011000010000$			

3.2.2 Formats de la norme

La norme IEEE 754 défini les différents formats de stockage de flottants, voyons les trois principaux :

	Nombre de bits	Bit signe	Bits exposant	Bits implicite	Bits mantisse	Décalage
Simple précision	32	1	8	1	23	127
Double précision	64	1	11	1	52	1023
Quadruple précision	128	1	15	1	112	16383

Par défaut, le type « float » de Python est en double précision.

Exercice 2 *: Coder* $(-10{,}125)_{10}$ *en flottant simple précision.*

Exercice 3*: Coder* $(2100)_{10}$ *en flottant simple précision.*

***Exercice 4**: On veut coder la vitesse de la lumière, 299 792 458, en flottant simple précision. On donne le résultat suivant. Que peut-on en conclure ? De quel ordre est l'erreur commise ?*

```
Entrée[19]:
print(bin(299792458))
0b10001110111100111100001001010

Entrée[25]:
print(len("10001110111100111100001001010"))
29
```

3.3 A retenir :

- On peut montrer qu'en simple précision, on a une précision d'environ 7 chiffres significatifs.
- En double précision, on a une précision d'environ 16 chiffres significatifs.
- Le type float de Python est en double précision :

```
>>> import sys
>>> sys.float_info
sys.float_info(
max=1.7976931348623157e+308,
max_exp=1024,
max_10_exp=308,
min=2.2250738585072014e-308,
min_exp=-1021,
min_10_exp=-307,
dig=15,
mant_dig=53,
epsilon=2.220446049250313e-16,
radix=2,
rounds=1)
```

- Attention aux erreurs d'arrondis dans des calculs avec les flottants.
- Il ne faut jamais commparer de manière stricte des flottant. On utilisera plutôt une comparaison à un nombre petit, mais plus grand que la précision des flottants (15 chiffres significatifs)

```
>>> 0.1 + 0.2 == 0.3
False
>>> abs(0.1 + 0.2 - 0.3)<1e-10
True
```

QR Code poru accéder au Notebook Capytale :