Informatique - Représentation des nombres

1. Nombres entiers

1.1. Conversions dans Python

Écrire les fonctions de conversion de décimal en binaire ou de décimal en hexadécimal et réciproquement sans utiliser les fonctions de Python est un excellent exercice d'algorithmique.

Mais il existe des fonctions disponibles dans Python. Python note `0b01011010` un nombre en binaire et `0x1a2c3e` un nombre en hexadécimal.

- **bin** et **hex** convertissent un nombre en une chaîne de caractères donnant la représentation binaire ou hexadécimale

```
>>> bin(156)
'0b10011100'
```

```
>>> hex(156)
'0x9c'
```

- dans l'autre sens, une représentation binaire `0b011010` ou hexadécimale `0x1ab4` est automatiquement associée au nombre correspondant :

```
>>> eval('0b10011100')
156
```

```
>>> eval('0x9c')
156
```

```
>>> a=0x9c
>>> print(a)
156
>>> print(a*2)
312
```

Algorithmes de changement de base :

```
def base2(n):
    L=[]
    while n!=0 :
        L.append(n%2)
        n = n//2
    return L
```

```
>>> base2(26546)
[0, 1, 0, 0, 1, 1, 0, 1, 1, 1,
1, 0, 0, 1, 1]
```

```
def base10(n):
    L=[]
    while n!=0 :
        L.append(n%10)
        n = n//10
    return L
```

```
>>> base10(26546)
[6, 4, 5, 6, 2]
```

```
def base16(n):
    chiffres = "0123456789ABCDEF"
    hex = "0x"
    while n!=0:
        hex = hex + chiffres[n%16]
        n=n//16
    return hex
```

```
>>> base16(26546)
'0x2B76'
```

Et dans l'autre sens, pour obtenir un entier à partir de sa représentation décimale :

```
def bin2dec(L):
    base = 2
    n = 0
    for k in range(len(L)):
        n = n + L[k]*base**k
    return n
```

qui donne, si $n = \underline{a_{p-1}a_{p-2}...a_1a_0}_2$, $n = \sum_{k=0}^{p-1} a_k 2^k$

```
>>> bin2dec(base2(26546))
26546
```

Autre version en utilisant le schéma de Horner :
Si $n = \underline{a_{p-1}a_{p-2}...a_1a_0}_2$,
alors $n = ((...(((a_{p-1})2+a_{p-2})2+a_{p-3})2+...)2+a_1)2+a_0$

```
def bin2decHorner(L):
    base = 2
    p = len(L)
    n = 0
    for k in range(n):
        n = (n + L[n - 1 - k])*base
    return n
```

1.2. Encodage des entiers

L'encodage des entiers naturels dans un programme utilise simplement la représentation binaire.

On utilise un bit pour savoir si ce nombre est positif ou négatif : 0 pour positif et 1 pour négatif.

Ce qui donne :

$\boxed{0}100$ $\boxed{1}100$
↓ ↓
est un nombre positif est un nombre négatif

Mais on ne se contente pas de rajouter ce bit devant le nombre entier, car cette méthode pose problème, notamment la présence d'un zéro positif et d'un zéro négatif

$\boxed{0}000$ $\boxed{1}000$
↓ ↓
zéro positif zéro négatif

On utilise **"le complément à deux"**. Si la représentation se fait sur n bits (n = 32 pour un entier usuel), on code un entier relatif a dans l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$:

- par a si a est positif,
- par $2^n + a$ si a est négatif.

Exemple sur 8 bits : $n = 8$, $a = -4$

Le nombre entier 4 est représenté par $\underline{00000100}$. L'entier -4 est codé par 256-4 qui est représenté par $\underline{11111100}$

Le plus grand entier positif représentable est 127 car $\underline{01111111} = 127$ et le plus petit entier négatif représentable est -128 car $\underline{10000000} = \underline{11111111} - \underline{01111111} = 256 - 1 - 127 = 256 - 128$.

On parle de complément à 2 car l'opération $2^n + a$ peut être effectuée en binaire en échangeant les chiffres 1 et 0 (complément à 2) puis en ajoutant 1 en binaire :

$-35 \rightarrow \underline{00100001}$ $(= 35) \rightarrow \underline{11011110}$ $(= 220) \rightarrow \underline{11011111}$ $(= 221) \rightarrow 221$

2. Nombres flottants

2.1. Partie fractionnaire

Principe de la représentation binaire d'un décimal

Pour représenter un nombre à virgule en binaire, on sépare la partie entière de la partie fractionnaire du nombre :

pour un réel a positif, la partie entière est $x = \lfloor a \rfloor$ et la partie fractionnaire est $y = a - x$ avec $y \in [0, 1[$.

Exemple : Pour $a = 2{,}34375$, la partie entière de a est 2 et sa partie fractionnaire est 0,34375.

Pour représenter une partie fractionnaire en binaire, on utilise des puissances négatives de 2 :

ici $0,34375 = 0,25 + 0,0625 + 0,03125 = \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^5} = \frac{0}{2^1} + \frac{1}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5}$

D'où la représentation $a = \underline{10,01011}$

2.2. Conversion en binaire des parties fractionnaires

Si $y \in [0,1[$, on multiplie y par 2 et on note b_1 la partie entière de $2y$ et y_1 la partie fractionnaire de $2y$.

On multiplie y_1 par 2 et on note b_2 sa partie entière et y_2 sa partie fractionnaire...

Les chiffres b_i donnent la représentation $y = \underline{0, b_1 b_2 b_3 ... b_p ..}$

Remarque : Certains nombres décimaux ont une représentation binaire infinie, comme par exemple $0,4 = \underline{0,011001100110011\ldots}$

2.3. Représentation d'un nombre réel : nombre flottant

Principe : Un nombre à virgule x est représenté dans une notation similaire à la notation scientifique : sous la forme $x = s.m.2^e$ où s est le signe $+1$ ou -1, m est la mantisse qui est un nombre à virgule normalisé, et e est l'exposant.

La mantisse contient les chiffres significatifs.

Par exemple, 299792458 s'écrit en notation scientifique $2,99792458.10^8$.

L'encodage des nombres flottants est normalisé par la norme IEEE 754.

Dans cette norme, la mantisse est un nombre compris entre 1 inclus et 2 exclu.

Exemple : 299792458 sera représenté par $1,11681393534.2^{28}$

2.4. Encodage IEEE754

La norme IEEE754 prévoit deux encodages pour les nombres flottants :

- **float :** nombre flottant codé sur 4 octets soit 32 bits
- **double :** nombre flottant codé sur 8 octets soit 64 bits

Pour Python, le type `float` se code par défaut en double précision (sur 8 octets soit 64 bits)

Encodage en simple précision sur 4 octets - "float"

Un réel x est stocké sous la forme approché (s, m, e)

$$x = s.m.2^e \text{ où}$$

- $s \in \{-1, +1\}$ représente le signe codé sur 1 bit,
- e représente l'exposant entier relatif codé $e + 127$.

  On utilise 8 bits, on peut donc avoir $e + 127 \in [\![0, 255]\!]$ mais les valeurs extrêmes 0 et 255 sont réservées pour d'autres usages, alors $e + 127 \in [\![1, 254]\!]$ soit $e \in [\![-126, +127]\!]$.

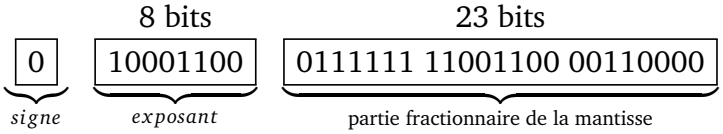
  Il y a une limite à la représentation des nombres flottants.
- $m \in [1, 2[$ représente la mantisse codée sur 24 bits.

  En fait, la mantisse est de la forme $m = \underline{1, a_0 a_1 a_2 a_3 a_4 \ldots}$.

  On ne code pas le 1 et on ne stocke que 23 bits pour le codage de m.

  Il y a perte de précision pour la représentation des nombres flottants.

  Exemple : pour $\underline{01000110\,00111111\,11001100\,00110000}$:



On a $s = 0$ et $e = 140 - 127 = 13$. Alors, le nombre flottant associé à ce binaire est donné par :

$$(-1)^s 2^{e-127}\left(1 + \sum_{i=1}^{23} b_i 2^{-i}\right)$$
$$= 2^{140-127}\left(1 + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-7} + +2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-18} + 2^{-19}\right)$$
$$= 12275.046875$$

Remarques

- On ne peut pas représenter tous les nombres exactement : 0,4 (par exemple) ne se représente pas de manière exacte en flottant.
- En simple précision (sur 4 octets), 0,4 est représenté par le nombre binaire :

  00111110 11001100 11001100 11001101

  qui vaut :

$$2^{125-127}\left(1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-23}\right) \simeq 0.400000005960465$$

Encodage en double précision sur 8 octets - "double"

Un réel x est stocké sous la forme approché (s, m, e)

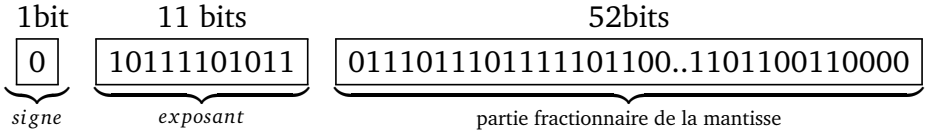
$$x = s.m.2^e \text{ où}$$

- $s \in \{-1, +1\}$ représente le signe codé sur 1 bit,
- e représente l'exposant entier relatif codé $e + 1023$. On utilise 11 bits (et on réserve les valeurs extrêmes) : on peut donc avoir $e \in [\![-1022, +1023]\!]$,
- $m \in [1, 2[$ représente la mantisse codée sur 53 bits.

  En fait, la mantisse est de la forme $m = \underline{1, a_0 a_1 a_2 a_3 a_4 \ldots}$.

  On ne code pas le 1 de la partie entière de m donc on ne stocke que 52 bits pour m.

  Exemple de représentation :



Remarques

- C 'est l'encodage utilisé par défaut dans Python.
- On ne peut toujours pas représenter tous les nombres exactement.
- Dans Python, 0.4 est représenté par 0.39999999999999997 avec la double précision