Informatique V
Représentation informatique des nombres

Laurent Kaczmarek

PCSI^2 2013-2014
Lycée Louis Le Grand

Lundi 23 septembre 2013

Écriture en base b

Voici un résultat que nous avons démontré dans le cours de Mathématiques :

Écriture en base b d'un entier naturel n

Soient $n \in \mathbb{N}^*$ et $b \in \mathbb{N} \setminus \{0, 1\}$.

- Il existe un unique entier naturel m et un unique $(m+1)$-uplet $(c_0, c_1, \ldots, c_m)$ d'entiers appartenant à $\{0, 1, \ldots, b-1\}$ tels que

$$n = c_0 + c_1 \cdot b + c_2 \cdot b^2 + \cdots + c_m \cdot b^m \quad \text{et } c_m \neq 0$$

  ce que l'on écrit de manière condensée $n = (c_m \ldots c_1 c_0)_b$.
- On dit qu'il s'agit du développement en base b de l'entier naturel n. Les c_i sont les chiffres de n en base b.

Écriture en base b

L'algorithme suivant a été prouvé dans le cours de Mathématiques :

Algorithme de décomposition en base b

Algorithme 1 : Décomposition en base b

Données : deux entiers naturels n tels que $b \geqslant 2$;
Résultat : chiffres de l'écriture en base b de n;
Initialisation : $i \leftarrow 0$;
tant que $n \neq 0$ **faire**
    $c_i \leftarrow$ reste dans la division euclidienne de n par b;
    $n \leftarrow$ quotient dans la division euclidienne de n par b;
    $i \leftarrow i + 1$;
fin
Renvoyer $(c_m \cdots c_0)$ où m est le plus grand des entiers naturels k tel que $c_k \neq 0$.

Écriture en base b

Dans la pratique...

on applique l'algorithme en effectuant des divisions successives à partir de $q_0 = n$:

q_0	b		
c_0	q_1	b	
	c_1	q_2	b
		c_2	q_3

et ainsi de suite jusqu'à obtenir $q_m \neq 0$ et $q_{m+1} = 0$.

On a alors $n = (c_m \cdots c_1 c_0)_b$

Exemple

- Écrire 673 en base $b = 2$.
- On trouve $673 = (1010100001)_2$.

Codage des entiers relatifs sous Python

Le type int

- Rappelons que les informations sont stockées en machine sous forme binaire, ainsi il en va des entiers. Ainsi, si une machine dispose de n bits pour coder ses entiers (décomposés en base 2), elle pourra stocker 2^n entiers distincts.
- Le type `int` contient des entiers positifs et des entiers négatifs. Afin de les distinguer, un bit (généralement celui de poids fort) est utilisé pour décrire le signe : 0 l'entier est positif, 1 il est négatif.

Le cas des entiers relatifs

Avec ces conventions, afin de stocker l'entier m codé en base 2 tout en utilisant un bit de signe, on peut utiliser différentes représentations des entiers négatifs...

- *Bit de signe + Codage de la valeur absolue.* Cette représentation présente l'inconvénient d'avoir 2 représentations pour 0. De plus, l'addition de nombres de signes opposés n'est pas simple.
- *Complément à 1 :* Bit de signe + Inverse des bits de la valeur absolue. Cette représentation présente l'avantage de pouvoir changer un nombre en son opposé de manière très simple. Cependant, 0 est toujours codé de deux faÃğons distinctes et l'addition de deux entiers s'avère difficile.
- *Complément à 2 :* Codage de $2^n - |m|$. Cette dernière solution est la solution retenue.

Le cas des entiers relatifs

- Ainsi, l'ensemble des entiers représentés sur n bits est compris entre : 0 et $2^{n-1}-1$ pour les entiers positifs, -2^{n-1} et -1 pour les entiers négatifs.
- L'ensemble des entiers codés sur 16 bits est donc $[\![-32768, 32767]\!]$ alors que l'ensemble des entiers codés sur 32 bits est $[\![-2147483648, 2147483647]\!]$.
- En utilisant la représentation en complément à 2, si un nombre positif a est représenté par l'écriture binaire $(a_{n-1} \cdots a_0)_2$, alors $-a$ sera obtenu :
  - en échangeant chacun des bits de $(a_{n-1} \cdots a_0)_2$ et en ajoutant 1.
  - en recherchant le bit 1 le plus à droite et à inverser les bits se trouvant strictement à sa gauche.

Le cas des entiers relatifs

- Les calculs sur les entiers codés sur 8 ou 16 bits sont très rapides car la plupart d'entre eux sont disponibles dans le processeur.
- Le type `int` en Python ne distingue pas les entiers courts (codés sur 8 bits) des entiers longs (entiers dont la taille n'est limitée que par l'espace alloué à l'interpréteur). Pour manipuler ces entiers longs, l'interpréteur doit les transformer en entiers courts manipulables par le processeur en les représentant comme des listes d'entiers courts. Ces calculs sont reliés à l'arithmétique multi-précision.

Les flottants

L'information est stockée, dans un ordinateur, dans des cases mémoires. Ainsi, un ordinateur ne peut stocker qu'un nombre fini de données. Un nombre réel quelconque ne pourra donc pas être stocké en détaillant chacune de ses décimales.
Python utilise le type `float` pour représenter un sous-ensemble des nombres décimaux.

Principe

Un nombre n à *virgule flottant* est représenté via :

- son signe $\varepsilon \in \{-1, 1\}$,
- sa mantisse m à virgule fixe,
- son exposant e,
- sa base b,

définis tels que $n = \varepsilon \cdot m \cdot b^e$.

La norme IEEE754 fixe les contraintes

- de l'écriture binaire : $b = 2$,
- de nombres codés sur 32 bits (en *simple précision*) ou 64 bits (en *double précision*),
- d'une répartition des bits de la façon suivante :

	signe	exposant	mantisse	excès d'exp.
32 bits	1 bit	8 bit	23 bits	127
64 bits	1 bit	11 bits	52 bits	1023

- Le signe est codé par 0 si le nombre est positif, 1 s'il est négatif,
- La mantisse appartient à l'ensemble $[1, 2[$. Le chiffre avant la virgule est ainsi toujours un 1 et n'est pas stocké. Lorsque l'exposant est nul, la mantisse est précédée d'un 0 et le nombre est dit dénormalisé,
- à la valeur d'exposant est ajoutée la valeur d'excès d'exposant. Cela permet de stocker des entiers compris entre -2^{e-1} et $2^{e-1} - 1$, où e est le nombre de bits dévolus à l'exposant.

Exemple

Pour coder le nombre

$$\begin{aligned} 16.5 &= 2^4 + \frac{1}{2} \\ &= \overline{10000.1}^2 \\ &= \overline{1.00001}^2 \cdot 2^4 \end{aligned}$$

Le signe vaut 0, l'exposant $127 + 4 = 131 = \overline{10000011}^2$ et la mantisse 00001. Ainsi, 16.5 est codé par
0 10000011 00001000000000000000000.

Exceptions

Les valeurs où tous les bits (sauf celui du signe) sont à 0, le nombre est ± 0.0. Si l'exposant est $2^e - 1$ et la mantisse est nulle, le nombre représenté est $\pm\infty$. Si l'exposant est $2^e - 1$ et la mantisse est non nulle, le nombre représenté est *NaN* (ou Not a Number). Ce résultat est renvoyé dès qu'on effectue une opération arithmétique invalide.

Limites à ne pas franchir

Dépassement

- Les nombres flottants, contrairement aux nombres de type `int`, sont bornés.
- La présence du point affecte grandement les calculs. Python sait manipuler des entiers de taille arbitraire, par exemple il affiche en base 10 l'entier 2**1024 dans l'interpréteur. En revanche, 2.0**1024 donne lieu à message d'erreur dans l'interpréteur.

Précision

- Le calcul avec des nombres flottants peut négliger certains termes.
- Le comportement de certaines commandes se trouve impacté.

```
>>> 2**1023-int(2.**1023+5.)
0
```

Inadéquation du test à 0

- Les nombres décimaux écrits en base 10 dans l'interpréteur sont convertis dans la norme IEEE754 avec perte d'information si la décomposition en binaire du nombre est infinie.
- Un résultat étonnant :

```
>>> 0.2+0.2==0.4
True
>>> 0.1+0.1+0.1==0.3
False
```