Lycée Loges

Chapitre 3

Codage des nombres en Python

Simon Dauguet
simon.dauguet@gmail.com

14 novembre 2024

Le système de numération occidental est en base 10. C'est un très ancien système de numération qui est apparu naturellement à cause de nos 10 doigts. Il a été utilisé dès le 3ème millénaire av JC par les égyptiens. Mais toutes les civilisations n'utilisaient pas ce système décimal. Les babyloniens, par exemple, utilisaient un système sexagésimal (en base 60). Les mayas comptaient en base 20. Mais lors d'échanges commerciaux, le nombre de bêtes dans un troupeau ne changeait pas simplement en passant une frontière. Il fallait donc pouvoir transcrire un nombre donné dans un système de numération dans un autre système. C'est ce que nous allons voir ici.

Nous allons expliciter le fonctionnement du codage d'un nombre dans un autre système que le système décimal usuel. Bien sûr, le but essentiel est de comprendre le codage binaire (en base 2) qui est le langage de base de tout système informatique.

Dans ce cours, nous allons comprendre comment fonctionne l'ordinateur pour manipuler les nombres. Dans la mesure où les ordinateurs fonctionnent sur un système binaire, nous allons donc voir comment coder les nombres binaires. Cette partie sera donc un peu théorique. Néanmoins, comme le but de ce cours n'est pas de faire un cours de maths, les démonstrations ne seront pas parfaites et un peu lacunaires. Pour plus de rigueur, voir le prof de maths.

On va s'attacher à l'aspect théorique mais aussi et surtout aux contraintes physiques auxquelles sont soumis les ordinateurs et comment ces contraintes imposent un comportement des ordinateurs

différent de la théorie. On essaiera de comprendre quelles ont été les solutions techniques apportées à ces contraintes physiques et, bien entendu, comprendre les problèmes qui sont engendrés. Entre autre, on donnera une explication aux erreurs d'arrondis que fait Python.

Table des matières

1 Nombre en base b : principe théorique

1.1 Codage en base b

Le système de numération adoptée est un système positionnel. Écrire un entier en base b correspond alors à l'écrire en somme de puissance de b où chaque position représente un nouvelle puissance de b et les chiffres correspondent au nombre de cette puissance nécessaire. Ces chiffres doivent donc être inférieur strictement à b, sans quoi, on change de puissance de b et donc de position.

Proposition 1.1 (Décomposition en base b) :
Soit $N \in \mathbb{N}$ et $b \in \mathbb{N}$, $b \geq 2$.
Alors il existe $n \in \mathbb{N}$ et $a_0, \ldots, a_n \in \{0, \ldots, b-1\}$, tel que

$$N = \sum_{k=0}^{n} a_k b^k$$

Démonstration :
C'est une histoire de division euclidienne. Ce sera vu plus en détail dans le chapitre sur l'arithmétique. On sait qu'il existe $a_0 \in \{0, \ldots, b-1\}$ et $q_0 \in \mathbb{N}$ tel que $N = a_0 + bq_0$ (division euclidienne). On recommence avec q_0 : $\exists (a_1, q_1) \in \{0, \ldots, b-1\} \times \mathbb{N}$ tel que $q_0 = a_1 + bq_1$. Et donc $N = a_0 + b(a_1 + bq_1) = a_0 + ba_1 + b^2 q_1$. Etc. Bien sûr la suite des (q_k) est décroissante et stationnaire à 0 à un moment n. D'où le résultat. □

Cette démo n'est pas très satisfaisante, mais ça suffira pour les besoins de l'informatique.

Définition 1.1 (Nombre en base b) :
Un nombre N est dit écrit en base b s'il est écrit suivant les puissances de b, autrement dit si

$$N = \overline{a_n a_{n-1} a_{n-2} \cdots a_1 a_0}^b = \sum_{k=0}^{n} a_k b^k$$

où $n \in \mathbb{N}$ et $\forall k \in \{0, \ldots, n\}$, $a_k \in \{0, \ldots, b-1\}$.

Par convention, lors de l'écriture d'un nombre en base b, on classe toujours par puissance de b en mettant la puissance de b la plus grande qui apparaît, le plus à gauche. On fait ensuite apparaître toutes les puissances de b jusqu'à b^0 qui sera le plus à droite.

Exemple 1.1 :
On a $\overline{1023}^4 = 1 \times 4^3 + 0 \times 4^2 + 2 \times 4^1 + 3 \times 4^0 = 75$. Et $789 = \overline{789}^{10}$. Mais ce cas n'est pas très intéressant. On a surtout $789 = 2 \times 7^3 + 103 = 2 \times 7^3 + 2 \times 7^2 + 5 = \overline{2205}^7$.

Exemple 1.2 :
Écrire en base 6 le nombre 724. Inversement, écrire en base 10, le nombre $\overline{25613}^{8}$.

Remarque :
Pour écrire un nombre en base b, il faut utiliser b symboles différents. Tant que la base est plus petite que 10, on a des symboles prédéfinis. Il suffit de choisir les même que pour la base 10 sans utiliser certains des symboles et de compter avec ceux là. Par exemple, en comptant en base 3, on a $\overline{0}^{3}$, $\overline{1}^{3}$, $\overline{2}^{3}$, $\overline{10}^{3}$, $\overline{11}^{3}$, $\overline{12}^{3}$, $\overline{20}^{3}$, ...

Dans le cas où la base est plus grande que 10, il faut alors utiliser de nouveaux symboles pour ne pas mélanger les deux systèmes (décimal et nouveau). Par exemple, le système hexadécimal est un système assez répandu qui est un système, comme son nom l'indique, en base 16. Les symboles sont, dans l'ordre croissant, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Et bien sûr, $F+1=10$. En base 16.

!!! ATTENTION !!!

⚠ On voit ici l'importance de préciser la base dans laquelle on compte. La seule base dont on peut se permettre d'omettre de préciser est la base du système décimal, c'est-à-dire la base 10. C'est la base dans laquelle on compte naturellement par convention.

Pour toutes les autres bases, il faudra préciser. En d'autre termes, il faudra toujours indiquer à quoi correspond le symbole 10. Il peut correspondre à 7 en base 7, 9 en base 9, 20 en base 20 etc.

Exercice 1 :
Écrire en base 10 les nombres $\overline{1A5F}^{16}$ et $\overline{5AB0}^{13}$.

Remarque :
On peut additionner en base b. Il suffit d'additionner chiffre par chiffre comme on le fait en base 10, en se rappelant que l'on compte dans une nouvelle base. Donc 6+1 ne fait pas forcément 7. La multiplication est aussi possible mais l'écriture propre et formelle est assez compliquée et ça n'amène pas grand chose.

Proposition 1.2 (Plage des entiers codé avec un nombre de symboles fixés) :
Soit $n \in \mathbb{N}^*$ et $b \in \mathbb{N}^*$.
Avec n symboles, on peut coder en bases b tous les entiers de $[\![0, b^n - 1]\!]$

Démonstration :
Notons $\beta = b - 1$ le plus grand symbole disponible.
Alors, le plus grand entier codé avec n symboles est

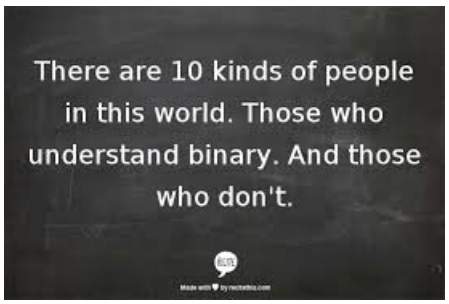
$$\underbrace{\overline{\beta \dots \beta}}_{n}^{b} = \sum_{k=0}^{n} \beta b^k = \beta \frac{b^n - 1}{b - 1} = b^n - 1.$$

□

Proposition 1.3 (Nombre de symbole pour le codage d'un entier) :
Soit $N \in \mathbb{N}$ et $b \in \mathbb{N}^*$.
Pour représenter l'entier N en base b, il faut $\lceil \log_b(N) \rceil$ symboles.

Démonstration :
Soit $n \in \mathbb{N}$ le nombre symbole nécessaire pour coder l'entier N en base b.
Alors $b^{n-1} < N \leq b^n$. Donc $n - 1 < \frac{\ln(N)}{\ln(b)} = \log_b(N) \leq n$. Et donc $n = \lceil \log_b(N) \rceil$. □

Autrement dit, pour coder un entier N en base b, il faut $\lfloor \log_b(N) \rfloor + 1$ ou $\log_b(N)$, selon si $\log_b(N) \in \mathbb{N}$ ou non.



Définition 1.2 (Codage binaire) :
Le codage binaire est l'écriture en base 2. En base 2, $\overline{1}^2 + \overline{1}^2 = \overline{10}^2$.

Exercice 2 :
Écrire en base 2 le nombre 2531. Convertir en base 10 le nombre $\overline{10001001101}^2$.

Nom du système de numération	Nombre de digits = base	Caractères des digits	Exemples
Binaire	2	0, 1	$10101_{(2)}$
Octal	8	0, 1, 2, 3, 4, 5, 6, 7	$650465_{(8)}$
Décimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	$1890732_{(10)}$
Hexadécimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	$4AF23E_{(16)}$

Définition 1.3 (`bin` et `hex`) :

- La fonction `bin(n:int) -> str` permet d'avoir le codage en binaire d'un entier n. Cette commande ne fonctionne que pour les entiers. Les deux caractères `"0b"` en début de chaînes ne sont pas à prendre en compte.
- La fonction `hex(n;int) -> str` permet d'avoir le codage hexadécimal d'un entier n. Cette commande ne fonctionne que pour les entiers. Les deux caractères `"0x"` en début de chaîne ne sont pas à prendre en compte.

Exemple 1.3 :

```
>>> bin(3)
"0b11"
>>> hex(-17)
"-0x11"
```

1.2 Opérations en base b

Les opérations usuelles ne dépendent pas de la base dans laquelle sont décrit les nombres. Les opérations fonctionnent de la même manière. Il faut juste faire attention aux retenues et à la somme des symboles.

Exemple 1.4 :

Calculons $\overline{243}^5 + \overline{342}^5$

	2	4	3
+	3	4	2
1	1	4	0

ainsi $\overline{243}^5 + \overline{342}^5 = \overline{1140}^5$

En effet,

- $\overline{243}^5 = 2 \times 5^2 + 4 \times 5 + 3 = 73$,
- $\overline{342}^5 = 3 \times 5^2 + 4 \times 5 + 2 = 97$ et
- $\overline{1140}^5 = 1 \times 5^3 + 1 \times 5^2 + 4 \times 5 = 170$.

Calculons $\overline{1032}^4 \times \overline{201}^4$

			1	0	3	2
×				2	0	1
			1	0	3	2
+	2	1	3	0	▪	▪
	2	2	0	0	3	2

ainsi $\overline{1032}^4 \times \overline{201}^4 = \overline{220032}^4$

En effet,

- $\overline{1032}^4 = 1 \times 4^3 + 3 \times 4 + 2 = 78$,
- $\overline{201}^4 = 2 \times 4^2 + 1 = 33$ et
- $\overline{220032}^4 = 2 \times 4^5 + 2 \times 4^4 + 3 \times 4 + 2 = 2574$.

Exercice 3 :

Calculer la somme et le produit de $\overline{210}^3$ et $\overline{102}^3$.

Les opérations avec les nombres binaires sont les mêmes que les opérations avec les nombres en base 10. Il suffit de les faire bit par bit en se rappelant que $\overline{1}^2 + \overline{1}^2 = \overline{10}^2$. Et puisqu'un exemple est parfois plus éloquent qu'un long discours :

L'addition

			Poids	64	32	16	8	4	2	1
	2	6				1	1	0	1	0
+	6	0	+		1	1	1	1	0	0
	8	6		1	0	1	0	1	1	0

La multiplication

			Poids	64	32	16	8	4	2	1
	1	3					1	1	0	1
x		5	x					1	0	1
	6	5					1	1	0	1
						0	0	0	0	
					1	1	0	1		
				1	0	0	0	0	0	1

Exercice 4 :

Écrire en base 11 les nombres 827 et 489. Puis faire la somme de ces nombres en base 11.

Calculer $\overline{123}^{12} \times \overline{A30}^{12}$.

1.3 Codage en base b en Python

1.3.1 De la base b à la base décimale

Définition 1.4 (Fonction de "décodage" décimale (`int`)) :
La fonction `int(code:str, b:int) -> int` est la fonction de transtypage qui transforme en entier en base décimale au sens large. Dans le cas où `code` est une chaîne de caractère représentant un entier dans la base `b`, la fonction renverra cet entier en base décimale.

Par convention, les symboles à utiliser sont, dans l'ordre et selon l'entier b de base, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, G, H, ... en majuscule.

Exemple 1.5 :

```
>>> int("1101",2)
13
>>> int("3A",11)
43
```

Exercice 5 :
Sans utiliser la fonction `int()` de "décodage" (mais de transtypage, oui), écrire une fonction `b2dec(N:str, b:int) -> int` qui renvoie l'entier `N` décrit en base `b` dans la base décimale.

1.3.2 De la base décimale à la base b

Pour passer de l'écriture décimale à l'écriture en base b, deux méthodes s'offrent à nous :

- tant que N n'est pas nul, on compte le nombre de fois que l'on soustrait la plus grande puissance de b possible ce qui nous donne le symbole de poids fort, et on recommence.
- tant que N n'est pas nul, N est divisé par b et le reste fourni le symbole de poids faible, et on recommence.

Exemple 1.6 :
Écrivons 3162 en base 6.
On rappelle que $6^2 = 36$, $6^3 = 216$, $6^4 = 1296$ et $6^5 = 7776$.
On remarque $6^4 \leqslant 3163 < 6^5$ ainsi l'écriture en base 6 aura 5 symboles.

		6^4	6^3	6^2	6^1	6^0
3163	-2592	2	2	3	5	1
571	-432					
139	-108					
31	-30					
1	-1					
0						

Par divisions successives par 6, cela donne

$$\begin{aligned} 3163 &= 6 \times 527 + 1 \\ 527 &= 6 \times 87 + 5 \\ 87 &= 6 \times 14 + 3 \\ 14 &= 6 \times 2 + 2 \\ 2 &= 6 \times 0 + 2 \end{aligned}$$

Ainsi on a bien dans les deux cas, $3163 = \overline{22351}^{6}$

Exercice 6 :
Déterminer, avec chacune des méthodes, l'écriture en base 8 de 73 et de 5603.

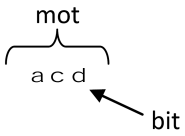
Exercice 7 :
Écrire une fonction `dec2b(N: int, b: int) -> str`, qui renvoie l'écriture de l'entier `N` dans la base b en utilisant les divisions successives.

2 Le codage binaire en informatique

2.1 Bits, Mots

Définition 2.1 (Bit, Mot, Octet) :

- L'information élémentaire est le bit. Qui peut valoir soit 0 soit 1. Les bits sont les lettres de l'alphabet binaire.
- Un mot est une suite de bits. Il peut comporter un nombre variable de bits. Dans un mot, on appelle bit de poids faible le bit le plus à droite (donc correspondant à la puissance de 2 la plus faible) et le bit de poids fort le bit non nul le plus à gauche dans le mot (donc le bit correspondant à la puissance de 2 la plus forte).
- Un octet est un mot de 8 bits.



Exemple 2.1 :
Le mot $\overline{10100100}^{2}$ est un octet. 1 est le bit de poids fort du mot précédent (et de poids 7) et 0 est le bit de poids faible de poids 0.

Les systèmes informatiques fonctionnent en mots de 8 bits, *i.e.* en octets. Les données sont codées en langage binaire et sont sauvegardées sous forme d'octets.

Définition 2.2 (Poids d'un fichier) :
Le poids d'un fichier informatique correspond donc au nombre de cases nécessaires pour stocker les données de ce fichier. Le poids d'un fichier est donc le nombre d'octets nécessaires pour la sauvegarde, la mémorisation de ce fichier. L'unité est donc l'octet (o) et le poids est mesuré avec tous les multiples de cette unité (ko, Go, To).

On mesure la capacité d'une mémoire informatique également en octets. C'est le nombre d'octets que l'on peut inscrire sur cette mémoire. Une clé USB de 8 Go est donc une mémoire pouvant contenir environ $8\times10^9 = 8\,000\,000\,000$ octets de données et donc aussi environ $8\times8\times10^9 = 64\,000\,000\,000$ bits.

Le mot "environ" est important ici. Tous fonctionne en puissance de 2. Le nombre d'octets est aussi à compter en puissance de 2 et pas en puissance de 10.

Remarque :
pour simplifier la lecture, il est courant de grouper les bits par paquets de 4. Ainsi $\overline{101110}^2$ se note $\overline{0010\ 1110}^2$ ou $(0010\ 1110)_2$.

2.2 Codage des entiers relatifs

Nous venons de voir comment coder un entier naturel. C'est à dire un entier positif. Mais dans le cas d'un entier négatif, il y a une information supplémentaire à manipuler, qui est le signe de cet entier. Il faut donc trouver une solution physiquement viable (on rappelle que les machines sont des objets physiques et que les données sont physiquement mémorisées par blocs de 8 cases).

Nous allons ici faire le raisonnement de la construction d'un bon système de sauvegarde des entiers. On va commencer par une approche naïve et donc mauvaise, analyser les problèmes de cette méthode et proposer ensuite un codage qui corrige les problèmes de l'approche naïve.

Solution naïve

La première idée est de considérer un bit en plus devant le code binaire d'un nombre pour coder le signe et de choisir 0 pour les nombres positifs et 1 pour les nombres négatifs. Par exemple, 7 se coderait $\boxed{0}\boxed{1}\boxed{1}\boxed{1}_{(2)}$ et -7 se coderait $\boxed{1}\boxed{1}\boxed{1}\boxed{1}_{(2)}$. Avec des octets, le premier est réservé pour le signe et les 7 restant permettraient de coder les chiffres. Donc un octet permettrait de coder tous les nombres dans l'intervalle $[\![-127, 127]\!]$:

0	1	1	1	1	1	1	1	127

0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	-0
1	0	0	0	0	0	0	1	-1
1	0	0	0	0	0	1	0	-2

1	1	1	1	1	1	1	1	–127

Le premier problème serait le 0 qui aurait 2 codages différents. Mais ce n'est pas tellement le plus gros souci.

Bien sûr, il y a des nombres plus grands que 127. Donc pour coder un nombre, on pourrait alors commencer par un octet qui coderait le nombre d'octets suivants sur lesquels serait inscrit le code du nombre voulu. Donc d'un point de vue technique, cette solution est parfaitement envisageable pour le moment.

Mais le problème majeur est au niveau des opérations. Idéalement, il faudrait que le codage soit compatible avec l'addition. On a vu que, lorsque l'on code des nombres en binaire, il est tout à fait possible de compter normalement avec ceux là, de faire des additions. Il faudrait donc trouver un codage physique qui puisse prendre en compte le signe des nombres que l'on manipule et qui soit compatible avec l'addition. Et avec ce codage naïf, l'addition ne fonctionne pas :

	0	0	0	0	0	0	1	0	+2
+	1	0	0	0	0	0	1	0	-2
=	1	0	0	0	0	1	0	0	-4

On voit qu'avec ce système, la somme ne fonctionne pas comme il faudrait. Et l'on ne peut pas redéfinir le notion de somme. Ce serait beaucoup trop compliqué.

On cherche donc :

- Un codage des entiers relatifs qui prenne en compte le signe
- Le code des entiers naturels doit être cohérent avec la version théorique donnée dans le paragraphe précédent
- L'addition dans ce codage doit encore être fonctionnel, quels que soient les signes des entiers à sommer.

Solution par décalage

Comme dans la solution précédente, ce sont les négatifs qui posent problèmes et pas les positifs, il pourrait suffire de "rendre tous le monde positif". Pour cela, on pourrait additionner une quantité permettant de n'avoir à coder que des entiers positifs. Par exemple, si les entiers sont condés sur n bits, on pourrait ajouter à tous les entiers 2^{n-1}, de sorte que tous les entiers soient positifs et coder alors ces entiers là.

Sur 4 bits, on aurait $6 + 2^3 = 14$ donc on encodera 6 par $\overline{1110}^2$ et puisque $-6 + 8 = 2$, -6 par $\overline{0010}^2$.

Zéro s'encoderait par $0 + 8 = \overline{1000}^2$ mais si l'on fait la somme $6 - 6$ on obtient $\overline{1110}^2 + \overline{0010}^2 = \overline{0000}^2$ qui n'est pas l'encodage du zéro par cette méthode.

Donc cette méthode règle le problème de l'unicité de l'écriture du 0, mais pas la compatibilité avec les opérations.

Codage en complément à 2 (*aka* code C2) :

L'idée est de voir les nombres négatifs comme en négatif au sens photographique du terme. L'idée est de voir les choses comme une sorte de tube à essai dont on comptabiliserait la partie remplie ou la partie vide. Un nombre positif correspond à la partie remplie, et un nombre négatif correspond

alors au vide du tube. Et on peut exprimer le vide comme le volume de la boîte totale MOINS la partie pleine. Autrement dit, un nombre négatif s'exprime comme étant un volume totale fixe moins un nombre positif. Comme le volume du tube est fixe, il suffit de coder l'entier positif à retranché. On se ramène donc à ne coder que des entiers positifs. Ce qui semble être une bonne idée pour l'addition. C'est exactement ce que l'on va faire :

Prenons le nombre $a = 81$. Essayons de coder le nombre $b = -a$. Le code $C2$ de b correspond alors au codage naturel de $2^8 - a = 175$. Et comme on a plus que des codages naturels, les opérations seront alors fonctionnelles :

a=81	0	1	0	1	0	0	0	1
$b = 2^8 - 81 = 175$	1	0	1	0	1	1	1	1

Vérifions maintenant que $a + b = 0$:

a		0	1	0	1	0	0	0	1
b	+	1	0	1	0	1	1	1	1
R	=	1	0	0	0	0	0	0	0

Dans la mesure où le résultat doit être sur 8 bits seulement, on n'a pas la place de coder le 1 supplémentaire et donc, on l'abandonne. On retrouve alors bien 0. Tout se passe bien.

Ce code ne fonctionne qu'avec une boîte de taille bien définie (le volume fixé du tube à essai). Il faut donc définir au préalable, dans la configuration de la machine, le nombre fixé d'octets qui seront alloués au codage d'un nombre et par rapport auquel on pourra coder les négatifs. Ce "volume" est donc fixé par la définition du langage utilisé. Utiliser alors un nombre dont le codage dépasserait les capacités de ce "volume" résulterait alors en une erreur machine.

Définition 2.3 (Codage C2) :
Le codage en complément à 2 (ou codage C2) est un codage des entiers relatifs répondant aux critères suivants :

(i) Définir un nombre n d'octets codant un nombre (ou $8n$ bits) permettant de coder les entiers $[\![-2^{8n-1}+1, 2^{8n-1}-1]\!]$.

(ii) Les entiers naturels sont codés avec le codage binaire naturel sur $8n-1$ bits (donc de 0 à $2^{8n-1}-1$).

(iii) Les entiers négatifs non nuls $-a$ (pour $1 \leq a \leq 2^{8n-1}$) sont codés par le codage binaire naturel de $2^{8n} - a$ sur les n octets disponibles pour le codage d'un nombre positif (donc de 2^{8n-1} à $2^{8n}-1$).

(iv) Lors d'une addition, les éventuels bits qui dépassent le quota maximum d'octets autorisé pour le codage d'un nombre ne sont pas pris en compte. Autrement dit, lors d'une addition, on tronque le résultat au n derniers octets qui le code.

$1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \quad 2^8$

1	1	1	1	1	1	1	1	2^8-1	$\rightsquigarrow$	$(2^8-1)-2^8=-1$
1	1	1	1	1	1	1	0	2^8-2	$\rightsquigarrow$	$(2^8-2)-2^8=-2$
				⋮						
1	0	0	0	0	0	0	1	2^7+1	$\rightsquigarrow$	$(2^7+1)-2^8=-2^7-1$
1	0	0	0	0	0	0	0	2^7	$\rightsquigarrow$	$2^7-2^8=-2^7$

} Entiers négatifs

0	1	1	1	1	1	1	1	2^7-1
0	1	1	1	1	1	1	0	2^7-2
				⋮				
0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0

} Entiers positifs

Proposition 2.1 (Codage C2) :
Le codage C2 sur n octets vérifie donc :

- On peut coder tous les nombres dans l'intervalle $[\![-2^{8n-1}+1, 2^{8n-1}-1]\!]$.
- L'entier 0 ne s'écrit que $000\ldots0_{(C2)}$.
- Tous les entiers positifs commencent par un 0, le plus grand d'entre eux étant $0111\ldots11_{(C2)} = 2^{8n-1}-1$.
- Tous les entiers négatifs commencent par un 1, le plus petit d'entre eux étant $100\ldots00_{(C2)} = 2^{8n} \rightsquigarrow 2^{8n-1}-2^{8n} = -2^{8n-1}$.

Remarque :
En pratique, pour coder un entier négatif dans le codage C2, il suffit de coder l'entier (positif) en binaire, échanger tous les 0 et 1, et ajouter 1.

Exemple 2.2 :
Ainsi sur 1 octet :

$$\text{complément à } 2 \text{ de } 5 = \begin{array}{l} \overline{0000\ 0101}^2 \\ \overline{1111\ 1010}^2 + \overline{0000\ 0001}^2 = \overline{1111\ 1011}^2 \end{array} \quad \text{complément à } 2 \text{ de } -5$$

car $\overline{1111\ 1011}^2 = 2^8 - 1 - 2^2 = 251$ et $251 - 2^8 = -5$.

Exercice 8 :
Sur 1 octet, déterminer la représentation en complément à 2 de -10 et -37.

Exercice 9 :
Préciser les nombres suivants encodés par complément à 2 : $\overline{1100\ 0101}^{C2}$ et $\overline{0011\ 1100}^{C2}$

2.3 Les flottants : norme IEEE 754

On aurait pu choisir d'avoir une représentation dite à virgule fixe qui impose un certain nombre de bits pour la partie entière d'un nombre et le reste pour la partie fractionnaire. L'inconvénient est que cette méthode restreint drastiquement la quantité de nombre que l'on peut coder.

Par exemple, si on garde 4 bits pour la partie entière et 4 pour la partie décimale, 14 s'écrira $\overline{1110,0000}^2$ tandis que $0,625 = 2^{-1} + 2^{-3}$ s'écrira $\overline{0000,1010}^2$.

On ne pourra pas encoder l'entier 130 malgré les 4 bits inutilisés de la partie fractionnaire, ni encoder le nombre $0,65625 = 2^{-1} + 2^{-3} + 2^{-5}$ malgré les 4 bits inutilisés de la partie entière.

Ainsi une solution largement utilisée est une représentation dite à virgule flottante. L'idée de la norme IEEE 754 est d'encoder une partie des nombres réels avec une représentation binaire similaire à la représentation scientifique en base 10.

Un nombre flottant est encodé par une sorte de représentation scientifique en base 2 : $N = (-1)^s 1, M \times 2^e$. On code donc sur 64 ou 32 bits (selon la nature de l'ordinateur) les entiers s, M et e.

Définition 2.4 (Base, Mantisse, Exposant) :
Soit $N \in \mathbb{D}$ un nombre. Alors, en passant par l'écriture en base 2, on peut écrire N sous la forme $N = (-1)^s 1, M \times 2^e$ où $e \in \mathbb{Z}$ et M est un entier écrit en base 2. M s'appelle la *mantisse*, s le *signe* et e *l'exposant*.

- flottant simple précision (32 bits) :

$\underbrace{\text{signe (s)}}_{1}$	$\underbrace{\text{exposant (e)}}_{8}$	$\underbrace{\text{mantisse (m)}}_{23}$

.

- flottant double précision (64 bits) :

$\underbrace{\text{signe (s)}}_{1}$	$\underbrace{\text{exposant (e)}}_{11}$	$\underbrace{\text{mantisse (m)}}_{52}$

.

Pour des soucis de cohérence, on rajoute que :

- Le signe est déterminé par $(-1)^s$
- L'exposant détermine la puissance de 2 par laquelle on multiplie la mantisse. Elle donne l'ordre de grandeur. Pour pouvoir écrire des petits ET grands nombres, on a besoin que la puissance puisse être négative. On opère donc un décalage comme pour le système de codage des entiers : exposant $=$ puissance (réelle) $+$ décalage. Le décalage étant soit $2^7 - 1 = 137$ si on code sur 32 bits, soit $2^{10} - 1 = 1023$ si on code sur 64 bits (donc selon le nombre de bits alloués pour le codage de l'exposant).

- La mantisse démarre toujours à 1. On ne le code donc pas. La mantisse correspond donc au code binaire de la partie fractionnaire.

Proposition 2.2 (Résumé du codage des flottants) :
Un flottant est donc codé de la manière suivante :

- En simple précision (32 bits) : $flt = (-1)^s \times 1, f \times 2^{e-127}$
- En double précision (64 bits) : $flt = (-1)^s \times 1, f \times 2^{e-1023}$.

Remarque :
La notation d'un nombre sous la forme $N = (-1)^s 1.M \times 2^e$ a plusieurs avantages :

- Le nombre d'octets alloués pour coder e, donc la taille de e, permet de donner la taille des nombres représentables avec ce système.
- La taille de la mantisse M permet d'avoir la précision à laquelle on calcule.
- L'addition fonctionne toujours avec cette notation.

Exemple 2.3 :
En base 10, on a $0.203125 = 203125 \times 10^{-6}$. En base 2, $\overline{0.203125}^{10} = \overline{1101}^2 \times 2^{-6}$.

Exemple 2.4 :
Considérons le flottant simple précision codé par $flt = \overbrace{1}^{s}\ \overbrace{1000\ 1001}^{p+127}\ \overbrace{1011\ 1011\ 0100\ 0000\ 0000\ 000}^{fraction}$.

- le signe est $(-1)^1 = -1$ donc flt est négatif
- $e = \overline{1000\ 1001}^2 = 137$ donc la puissance $p = 10$
- $f = \overline{1011\ 1011\ 0100\ 0000\ 0000\ 000}^2$ correspond à $2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-7} + 2^{-8} + 2^{-10}$ (*i.e.* $m = 1 + f = 1,7314453125$)

Ainsi le flottant est
$$\begin{aligned} flt &= -(1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-7} + 2^{-8} + 2^{-10}) \times 2^{10} \\ &= -(2^{10} + 2^9 + 2^7 + 2^6 + 2^5 + 2^3 + 2^2 + 1) \\ &= -1773 \end{aligned}$$
On peut aussi voir que $-m \times 2^{10} = -1,7314453125 \times 2^{10} = -1773$

Exercice 10 :
Déterminer la valeur du flottant $0'1000\ 1000'0110\ 1111\ 0110\ 0000\ 0000\ 000$

Inversement, pour écrire un nombre en flottant, deux méthodes existent :

- soit on décompose le nombre en puissance de 2 (puissance négative comprise)
- soit on décompose la partie entière, puis tant que la partie fractionnaire n'est pas nulle on la multiplie par 2 et cette nouvelle partie entière donne le poids de la puissance de 2 correspondante.

Exemple 2.5 :
Encodons le nombre $274,15625$ en flottant simple précision.

Pour la partie entière, $274 = 2^8 + 2^4 + 2^1$, la partie fractionnaire donne

- pour $k = -1 : 0,15625 \times 2 = 0,31250$ on associe le poids 0 à la puissance 2^{-1}
- pour $k = -2 : 0,3125 \times 2 = 0,625$ on associe le poids 0 à la puissance 2^{-2}
- pour $k = -3 : 0,625 \times 2 = 1,25$ on associe le poids 1 à la puissance 2^{-3}
- pour $k = -4 : 0,25 \times 2 = 0,50$ on associe le poids 0 à la puissance 2^{-4}
- pour $k = -3 : 0,5 \times 2 = 1$ on associe le poids 1 à la puissance 2^{-5}

Ainsi $274,15625 = 2^8 + 2^4 + 2^1 + 2^{-3} + 2^{-5} = +(1 + 2^{-4} + 2^{-7} + 2^{-11} + 2^{-13}) \times 2^8$
donc $p = 8 \Rightarrow e = p + 127 = 135 = \overline{1000\ 0111}^2$
Au final le nombre $274,15625$ est encodé par $0'1000\ 0111'0001\ 0010\ 0010\ 1000\ 0000\ 000$

Exercice 11 :
Déterminer l'écriture flottante 32 bits de $-219,625$

Remarque :
Dans le cas où le nombre ne s'écrit pas comme somme de puissance de 2, il est impossible d'encoder de manière exacte ce nombre comme flottant. Selon les situations, on prendra son encodage par défaut ou celui par excès.

Considérons $N = 0,1$,

pour $k = -1 : 0,1 \times 2 = 0,2$ on associe le poids 0 à la puissance 2^{-1}
$\vdots$
pour $k = -4 : 0,8 \times 2 = 1,6$ on associe le poids 1 à la puissance 2^{-4}
pour $k = -5 : 0,6 \times 2 = 1,2$ on associe le poids 1 à la puissance 2^{-5}
$\vdots$
pour $k = -8 : 0,8 \times 2 = 1,6$ on associe le poids 1 à la puissance 2^{-8}
pour $k = -9 : 0,6 \times 2 = 1,2$ on associe le poids 1 à la puissance 2^{-9}
$\vdots$
pour $k = -12 : 0,8 \times 2 = 1,6$ on associe le poids 1 à la puissance 2^{-12}

pour $k = -13$: $0,6 \times 2 = 1,2$ on associe le poids 1 à la puissance 2^{-13}

$\vdots$ etc.

Ainsi la représentation flottante de $0,1$ ne sera pas exacte. Puisqu'en simple précision il y a 23 bits pour la mantisse, on pourra utiliser les flottants a ou b suivants (la norme IEEE754 prévoit en vrai 4 manières d'arrondir) :

$$\begin{aligned} a &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-24} + 2^{-25} \\ &= \left(1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21}\right) \times 2^{-4} \\ &\approx 0,099\ 999\ 994\ 039\ 5 \\ b &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-24} + 2^{-25} + 2^{-27} \\ &= \left(1 + 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-23}\right) \times 2^{-4} \\ &\approx 0,100\ 000\ 001\ 49 \end{aligned}$$

2.4 Limitation physique : conséquences et contournement avec Python

L'ordinateur étant limité physiquement, il ne peut travailler qu'avec un nombre fini (même s'il est potentiellement grand) de décimales. Les calculs effectués sont donc des approximations. Ce qui entraîne certains problèmes.

2.4.1 Le cas des entiers

Pour les entiers, python est fait en sorte de pouvoir allouer une quantité de bits variables de manière à pouvoir toujours coder les entiers. Autrement dit, python fonctionne normalement avec des nombres coder en 64 (ou 32) bits, sauf dans le cas des entiers où il peut utiliser plus de bits. De sorte que python manipulera toujours des entiers de manières exactes, sans approximations.

Toutefois, pour des questions de sécurité afin de ne pas trop surcharger la machine, des limitations sont mises pour ne pas dépasser trop de bits. Il est possible (on ne le fera pas) de demander à python de contourner ses précautions.

Exemple 2.6 :
Avec votre calculatrice, calculer 2024^{2024}. Que se passe-t-il ?

Mais ce n'est pas le cas dans tous les langages. Beaucoup de langages informatiques n'ont pas cette adaptation naturelle et les entiers trop grands sont alors "dénaturés". C'est le cas des calculatrices par exemple.

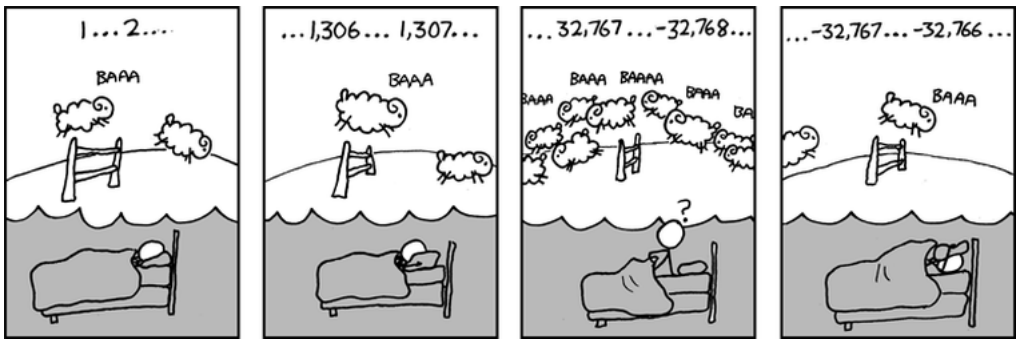
Ce phénomène est le dépassement arithmétique (*overflow* en anglais) que vous avez déjà pu rencontrer.

On rappelle que le codage des entiers relatifs se fait avec le codage en complément à 2. Avec ce code, on se retrouve également face à un problème. Par exemple, avec le codage C2, on a $2\ 000\ 000\ 000 = \overline{1110\ 1110\ 0110\ 1011\ 0010\ 1000\ 0000\ 000}^2$. On obtient donc

$$2 * 2\ 000\ 000\ 000 = \overline{1110\ 1110\ 0110\ 1011\ 0010\ 1000\ 0000\ 0000}^2$$

qui ne dépasse par 32 bits. Mais dans le code C2, ce nombre représente $2^{32} - 4\ 000\ 000\ 000 = -294\ 967\ 296$. Et pas 4000000000 ! Pour remédier à cela, les entiers relatifs sont alors découpés par paquets (souvent de 15 bits). Un entier est donc la liste de ses "morceaux" de 15 bits avec une donnée supplémentaire permettant de savoir combien il y a de morceaux et si le nombre est positif ou non. Mais on ne rentrera pas plus en détail.

Dans d'autres langages qui ne fonctionnent pas avec ce système, on peut se retrouver avec, soit des erreurs *overflow*, soit une somme de nombres positifs donnant un nombre négatif (comme dans l'exemple). Ce qui n'est pas tellement mieux. Il est alors impératif de veiller à ne pas dépasser les capacités du codage utilisé.



2.4.2 Les réels : dépassements et problèmes de précision

Comme on l'a vu, les flottants sont codés sur un certain nombre de bits. En général, les machines fonctionnent soit sur un système en 32 bits, soit 64 bits. Dans les deux cas, si on tombe sur un flottant qui se code sur plus de bits que le nombre de bits alloués pour la sauvegarde de nombres, les bits surnuméraires sont perdus.

Pour se simplifier la vie, raisonnons sur des entiers. Si on admet que l'on fonctionne avec une machine en 32 bits et que l'on calcule avec Python 260^5, c'est à dire 4569760000, qui se code $\overline{1\ 0001\ 0000\ 0110\ 0001\ 0000\ 0001\ 0000\ 0000}^2$ en binaire et donc sur 33 bits, le dernier bit, le bit le plus à gauche est donc normalement perdu. La machine devrait donc sauvegarder $\overline{0001\ 0000\ 0110\ 0001\ 0000\ 0001\ 0000\ 0000}^2 = 274\ 792\ 704$.

Le problème est le suivant : si l'on considère un réel très grand et avec un nombre de décimales infini, comment faire pour sauvegarder cette valeur. Par exemple, si on considère π^{2024}. Ce nombre est très grand et nécessite donc un grand nombre de bits pour conserver l'exposant et la mantisse. Plus précisément, si l'on considère un système à 64 bits, il y a au maximum 11 bits alloués pour la sauvegarde de l'exposant (donc le plus grand exposant est 1023) et 52 bits sont alloués à la mantisse. Avec un tel flottant, on dépasse les capacités de la machine et on se retrouve alors avec un problème de dépassement arithmétique (encore).

On a une situation similaire avec les trop petits nombres. Par exemple, un flottant petit ne doit pas avoir un exposant plus petit que -1022 (plus petit nombre codé sur 11 bits). Dans ce cas, on est en dépassement arithmétique inférieur (ou *underflow* en anglais).

Si un flottant dépasse les limites de la machine (soit par un *overflow* soit un *underflow*), on aura, selon les cas, soit une approximation faite par la machine, soit une erreur. Un flottant trop petit pourra alors être considéré comme étant 0 par la machine.

Pour arranger ce genre de problème, Python fonctionne avec des arrondis. Chaque flottant est en réalité une valeur arrondie de sa valeur réelle pour que l'arrondi soit codable avec seulement 64 bits. Par exemple, en base 2, on a

$$0.4 = 0.0110011001100110011..._{(2)}$$

qui a donc un développement décimal infini en base 2. On le code alors de la manière suivante :

Signe	Exposant	Mantisse	Bits perdus
0	01111111101	1001100110011001100...110011001	10011001...

Les conventions d'arrondis (que nous ne détaillerons par car peu pertinentes pour nous) donne alors :

$$\begin{aligned} &\overline{0,0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 10}^2 \\ &= \frac{7\ 205\ 759\ 403\ 792\ 794}{18\ 014\ 398\ 509\ 481\ 984} \\ &\approx 0.4000000000000000022204460492503 \end{aligned}$$

Ce qui est considéré comme $0,4$ pour Python. Vous pouvez faire le calcul, vous verrez que la fraction est considérée comme étant $0,4$, c'est à dire $2/5$. Il est clair que ce n'est pas le cas.

Du fait de ces arrondis, on peut se retrouver avec des erreurs de calculs qu'il est nécessaire de savoir repérer et interpréter. Ces erreurs de calculs interviennent très souvent lorsque l'on somme des nombres avec des ordres de grandeurs très différents. Par exemple :

$$1 + 2^{-54} = \overline{1.0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 01}^2$$

qui est considéré comme 1 pour Python. L'ordre dans lequel on fait les opérations va correspondre à l'ordre dans lequel les approximations sont faites :

```
>>> 1+2**(-54)-1
0.0
```

La première somme est approximée à 1.0 à laquelle on retranche ensuite 1, ce qui donne 0.0.

```
>>> 1-1+2**(-54)
5.551115123125783e-17
```

La première somme donne 0 à laquelle on ajoute 2^{-54}. On calcul donc seulement 2^{-54} qui est alors directement arrondi au flottant codé sur 64 bits le plus proche de 2^{-54}.

!!! ATTENTION !!!

⚠ Attention donc ! L'addition n'est pas associative ni commutative pour Python !

$$\begin{aligned} \big(1 + (-1)\big) + 2^{-54} &\neq 1 + \big((-1) + 2^{-54}\big) \\ 3 \times 0.1 &\neq 0.3 \\ (1.6 + 3.2) + 1.7 &\neq 1.6 + (3.2 + 1.7) \\ 1.5 \times (3.2 + 1.4) &\neq 1.5 \times 3.2 + 1.5 \times 1.4 \end{aligned}$$

Remarque :
Il s'en suit que des tests de la forme `a==b` entre flottants n'a pas de sens, puisqu'on manipule en réalité des valeurs approchées. Il se pourrait alors que le test effectué par Python soit faux alors que théoriquement les deux valeurs devraient être les mêmes. Pour ce faire, il vaut mieux remplacer cette condition par `abs(a-b)<epsilon` qui a plus de sens compte-tenu du fait que Python fait des approximations tous le temps.

Il conviendra bien sûr de choisir une valeur de `epsilon` pas trop grande et qui a un sens par rapport au problème. Dans le cas d'une convergence par exemple, on ne choisira pas la même valeur que si l'on essaie de savoir si un astéroïde va heurter la Terre, ou bien même encore si l'on veut rendre la monnaie. Dans ce dernier cas, un erreur de calcul de moins de 1 centimes pour être négligé (on ne peut pas rendre moins de 1 centimes).

3 Codes des caractères - Information

Comme les nombres, les caractères sont encodés sur des octets.

Lors de l'essort des ordinateurs à la fin de la guerre, chaque fabricant proposait son système d'encodage. La communication entre matériels informatiques provenant de deux fabricants différents étaient par conséquent rarement compatibles.

Norme ASCII

Face à cette multiplicité d'encodage, la ANSI (*American National Standards Institute*) proposa dans les années 1960 une norme définissant 128 caractères codés sur 1 octet (le bit de poids fort était un bit de parité (le nombre de 1 dans l'octet devant être pair) il servait donc de clef de vérification de l'encodage).

Cela a donné la table ASCII ([`http://www.table-ascii.com/`], *American Standard Code for Information Interchange*) suivante (ici codée en hexadécimal) :

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	¡	=	¿	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	—	}	~	DEL

Exemple 3.1 :
Le caractère 'A' est codé en hexadécimal $\overline{41}^{16}$ soit 65 ou encore avec le bit de parité $\overline{0100\ 0001}^{2}$.

Le caractère '_' est codé par $\overline{5F}^{16}$ soit 95 ou encore avec le bit de parité $\overline{0101\ 1111}^{2}$.

En python, la fonction `ord(c: str) -> int` renvoie la valeur décimal du code ASCII du ca-

ractère `c` tandis que la fonction `chr(code: int) -> str` renvoie le caractère ayant pour code ASCII `code`.

Exemple 3.2 :
`ord('A')` renvoie 65 et `chr(97)` renvoie `'a'`.

Exercice 12 :
On souhaite une fonction `ascii2txt(cascii: str) -> str`, qui traduit une chaîne de caractères contenant la valeur des codes ASCII en hexadécimal en la chaîne de caractères à proprement parlé. Par exemple, `ascii2txt( "4D505349")` renverra `"MPSI"`

a) Que renvoie `ascii2txt( "496E666F4D617468")` ?

b) Proposer le code python de cette fonction.