Représentations des données

Objectif : Toute machine informatique ne manipule que des représentations des données : des suites formatées de bits (contraction de *binary digits*) c'est-à-dire de 0 et de 1. Les différents types de données (nombres entiers ou réels, chaînes de caractères, valeurs booléennes, images, sons) sont donc représentées selon un codage dépendant de leur nature. Le codage des images et des sons ne sera pas étudié dans ce cours où nous nous limitons, dans une 1re partie, aux nombres et aux caractères et, dans une 2e partie, aux fonctions logiques.

Plan du cours :

- Le codage binaire des entiers positifs est tout d'abord envisagé. Appliqué à l'informatique, cette représentation admet des limites fixées par la taille de l'objet en mémoire (exprimée en octet). La prise en compte des nombres entiers négatifs, donc d'un signe, induit de nouvelles limitations.
- Les nombres réels étant impossibles à représenter exactement (du fait de la partie décimale illimitée), ils sont approximés par une structure appelée « nombres flottants » qui se décline en plusieurs modalités selon la précision requise. On verra que cette approximation peut conduire à des erreurs importantes à propos desquelles il faut toujours rester vigilent.
- Une chaîne de caractères étant une suite de caractères (lettres minuscules, majuscules, chiffres, signes de ponctuation, symboles mathématiques), nous étudierons ici les différentes façons de coder les caractères, c'est-à-dire de remplacer ces caractères par des nombres qui seront eux-mêmes écrits en binaire dans l'ordinateur. Nous envisagerons donc le codage ASCII et ses variantes (latin-1 par exemple) et la norme Unicode ainsi que son dérivé, l'UTF-8. La mise en forme des textes (traitements de texte, pages WEB) sera étudiée au chapitre 4.
- Les fonctions booléennes traduisant les opérations logiques (et, ou, non, etc.) sont envisagées dans une deuxième partie de ce chapitre. Elles sont utilisées dans les langages de programmation mais aussi dans l'architecture interne des ordinateurs. Elles peuvent être décrites par des tables ne contenant que des 0 (faux) et des 1 (vrai), parfois appelées « tables de vérité » et aussi par un langage symbolique ; nous envisagerons comment passer d'une représentation à l'autre.

1. Représentation des nombres entiers

Aperçu historique :

Depuis le moyen âge, on écrit les nombres entiers naturels en notation décimale à position. La notation est dite décimale car on utilise la base dix où les valeurs des dix chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9 sont multipliées par une puissance de dix. Cette caractéristique n'étant pas utilisable directement par un ordinateur qui ne manipule que des 0 et des 1, on utilise la notation binaire à position où les valeurs des deux chiffres 0 et 1 sont multipliées par une puissance de deux. Si les premières expressions du binaire en occident remonte au XVIIe siècle (tables de Thomas Harriot (1560-1621), code bilitère de Francis Bacon (1561-1626), traité de rhabdologie de John Napier (1550,1617) expliquant les opérations binaires, étude de Juan Caramuel y Lobkowitz (1606,1682)),

Gottfried Wilhelm Leibniz (1646-1716) expose l'arithmétique binaire devant l'Académie des sciences de Paris en 1703 (p.86 du Mémoire ci-dessous). L'arithmétique binaire est utilisée par les systèmes électroniques les plus courants car les deux chiffres 0 et 1 s'y traduisent par la tension ou le passage d'un courant. Par exemple, le 0 peut être représenté par l'état bas (tension ou courant nul) et 1 par l'état haut (tension qui existe, courant qui passe).

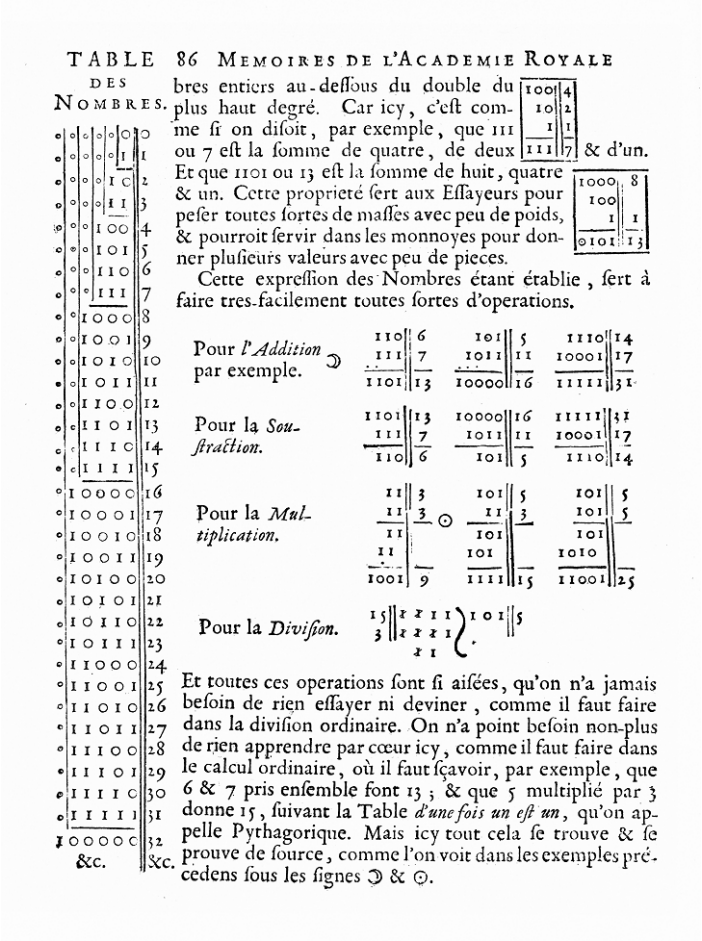


TABLE DES NOMBRES.

86 MEMOIRES DE L'ACADEMIE ROYALE

bres entiers au-deſſous du double du plus haut degré. Car icy, c'eſt comme ſi on diſoit, par exemple, que 111 ou 7 eſt la ſomme de quatre, de deux & d'un. Et que 1101 ou 13 eſt la ſomme de huit, quatre & un. Cette proprieté ſert aux Eſſayeurs pour peſer toutes ſortes de maſſes avec peu de poids, & pourroit ſervir dans les monnoyes pour donner pluſieurs valeurs avec peu de pieces.

Cette expreſſion des Nombres étant établie, ſert à faire tres-facilement toutes ſortes d'operations.

Pour *l'Addition* par exemple.

Pour la *Soustraction.*

Pour la *Multiplication.*

Pour la *Diviſion.*

Et toutes ces operations ſont ſi aiſées, qu'on n'a jamais beſoin de rien eſſayer ni deviner, comme il faut faire dans la diviſion ordinaire. On n'a point beſoin non-plus de rien apprendre par cœur icy, comme il faut faire dans le calcul ordinaire, où il faut ſçavoir, par exemple, que 6 & 7 pris enſemble font 13; & que 5 multiplié par 3 donne 15, ſuivant la Table *d'une fois un eſt un*, qu'on appelle Pythagorique. Mais icy tout cela ſe trouve & ſe prouve de ſource, comme l'on voit dans les exemples précedens ſous les ſignes ☽ & ☉.

&c. &c.

a. Écriture d'un nombre entier positif dans une base quelconque

DÉFINITION 1.1 (ÉCRITURE D'UN ENTIER EN BASE A) a étant un entier supérieur ou égal à 2, un entier positif n quelconque s'écrit en base a à l'aide des a chiffres inhérents à cette base sous la forme d'additions des puissances successives de cette base multipliées par des coefficients $a_k < a$:

- décomposition $n = a_0 \times a^0 + a_1 \times a^1 + a_2 \times a^2 + \cdots + a_p \times a^p$
- écritures condensée $n = a_p \dots a_2 a_1 a_0$ et indicée $n = a_p \dots a_2 a_1 a_{0_{(a)}}$

Remarques :

- Les chiffres inhérents à la base $a < 10$ sont les chiffres arabes de 0 à $a-1$. En base 8, par exemple, les chiffres sont 0, 1, 2, 3, 4, 5, 6 et 7. Les chiffres binaires ($a = 2$) sont 0 et 1. Pour une base $a \geqslant 10$, on ajoute les lettres a (pour 10), b (pour 11), etc. Ainsi, en base 16, on utilise les chiffres 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e et f.
- La notation de la base utilisée peut se faire de différentes façons mais nous utiliserons ici une notation indicée, la base étant indiquée en indice et entre parenthèses. Quant le contexte rend cette notation inutile (la base étant indiquée dans le texte) on la supprime. Ainsi, on écrira $459_{(10)}$, $110011001_{(2)}$ et $1af_{(16)}$ les nombres qui s'écrivent en base dix 459, 409 et 431. En Python, un nombre binaire (base deux) se préfixe par '0b' et un nombre hexadécimal (base 16)

se préfixe par '0x' ; ainsi en Python $110011001_{(2)}$ et $1af_{(16)}$ se notent $0b110011001$ et $0x1af$; ce type de notation s'impose du fait de l'écriture linéaire des instructions.

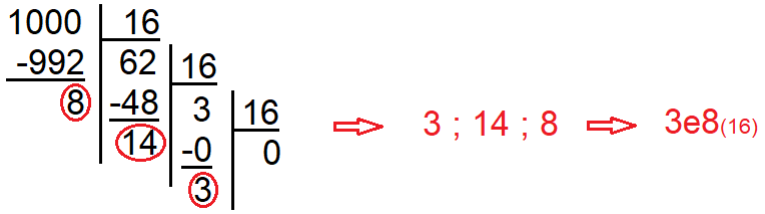
- La décomposition qui explique la valeur d'un nombre s'écrit généralement en base dix. Ainsi, dans les trois exemples cités ci-dessus, on a :
  — $459_{(10)} = 9 \times 10^0 + 5 \times 10^1 + 4 \times 10^2 = 9 + 50 + 400 = 459$
  — $110011001_{(2)} = 1 \times 2^0 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^7 + 1 \times 2^8 = 1 + 8 + 16 + 128 + 256 = 409$
  — $1af_{(16)} = 15 \times 16^0 + 10 \times 16^1 + 1 \times 16^2 = 15 + 160 + 256 = 431$

Méthode (Traduction décimale d'un nombre donné en base a)
La méthode dérive de la définition : on utilise la décomposition selon les puissances de la base. Le chiffre le plus à droite représente les unités, ensuite le 2^e chiffre représente les paquets de a unités (dizaines en base dix, deuzaine en base deux, vingtaines en base vingt, ...), le 3^e chiffre représente les paquets de a^2 unités (centaines en base dix, quatraine en base deux, quatre centaines en base vingt, ...), etc.
Voir les exemples ci-dessus pour lesquels les termes $a_k \times a^k$ ne sont pas écrits lorsque le chiffre a_k est nul (le but étant de simplifier l'écriture décomposée).
Remarquons que, comme les nombres $999\ldots9$ en base dix, il est facile de connaître la valeur de $bbb\ldots b$ en base $b+1$. Par exemple, comme $999_{(10)} = 10^3 - 1$, on a $666_{(7)} = 7^3 - 1 = 342$, $111111111_{(2)} = 2^9 - 1 = 511$ et $ffff_{(16)} = 16^4 - 1 = 65535$.
Remarquons une autre caractéristique du système décimal qui se transpose aisément à toutes les bases : lorsqu'on multiplie un entier décimal par 10 on ajoute un 0 à droite. Transposé à une base quelconque, cette propriété s'énonce ainsi : lorsqu'on multiplie par a un entier écrit en base a on ajoute un 0 à droite. Ainsi, on a $6660_{(7)} = 7 \times 342 = 2394$, $111111111000_{(2)} = 8 \times 511 = 4088$ (j'ai appliqué trois fois la propriété, donc il faut multiplier par $2^3 = 8$) et $ffff0_{(16)}$ vaut $16 \times 65535 = 1048560$ (dans ce dernier cas notamment, il est sans doute plus simple de procéder par soustraction $ffff0_{(16)} = fffff_{(16)} - f = 16^5 - 16 = 1048560$).
En Python, la conversion décimale s'écrit `int('n', a)` où `'n'` est le nombre écrit sous la forme d'une chaîne de caractères et `a` est la base. Par exemple `int('0b110011001',2)` renvoie 409 (comme d'ailleurs aussi `int('110011001',2)`) et `int('0x1af',16)` renvoie 431 (comme d'ailleurs aussi `int('1af',16)`).

Pour la transposition inverse, afin de déterminer le nombre de paquets de a unités, on commence par effectuer une division euclidienne par a (division conduisant à un quotient et un reste entiers) : le reste de cette division est le chiffre des unités, le quotient est le nombre de paquets de a unités qu'il convient de diviser à nouveau par a s'il atteint ou dépasse a (car le plus grand chiffre utilisable en base a a pour valeur $a-1$).

- Comment s'écrit 100 en base 16 ? On effectue la division euclidienne de 100 par 16 : $100 = 6 \times 16 + 4$. Comme $6 < 16$, on en déduit que $100_{(10)} = 64_{(16)}$.
- Comment s'écrit 1000 en base 16 ? On effectue la division euclidienne de 1000 par 16 : $1000 = 62 \times 16 + 8$. Comme $62 > 15$, on effectue la division euclidienne de 62 par 16 : $62 = 3 \times 16 + 14$. Comme $3 < 16$, on en déduit que $1000_{(10)} = 3e8_{(16)}$.

Cette démarche peut s'illustrer par une cascade de divisions euclidiennes successives par la base a dans laquelle on veut écrire notre nombre décimal. Cette cascade s'arrête lorsque le quotient obtenu est nul. Il suffit alors de remonter la suite des restes à l'envers pour obtenir l'écriture condensée en base a.



MÉTHODE (TRADUCTION EN BASE a D'UN NOMBRE DONNÉ EN BASE DIX)
On effectue la cascade des divisions euclidiennes successives par a décrite sur l'illustration ci-dessus. Cette cascade s'arrêtant lorsque le quotient obtenu est nul, il suffit de remonter la suite des restes à l'envers pour obtenir l'écriture condensée en base a.
Cette démarche algorithmique se programme aisément : on entre les nombres n et a et on obtient l'écriture condensée en base a. Une restriction s'impose : la base ne doit pas dépasser 36 (les dix chiffres arabes et les 26 lettres minuscules), mais le programme ci-dessous en Python ne teste pas la valeur de a entrée. On suppose que l'utilisateur sait ce qu'il fait et, s'il a besoin d'utiliser une base $36 \leqslant a \leqslant 72$ il peut toujours ajouter les lettres majuscules à la liste `chiffre`.

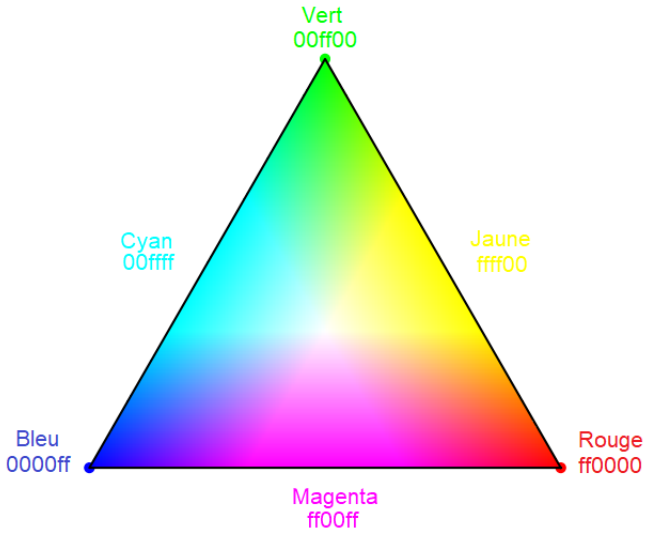
```
def conversionBase():
    reste,quotient,num,ecriture=list(),nb,nb,""
    while quotient>0:
        quotient=num//base
        reste.append(num%base)
        num=quotient
    for j in range(len(reste)): ecriture=chiffre[reste[j]]+ecriture
    return ecriture

chiffre=["0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f","g","h",\
         "i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]
nb=int(input("Entrer le nombre (en base 10) : "))
base=int(input("Entrer la base (en base 10) : "))
print("Le voici écrit dans cette base : {}".format(conversionBase()))
```

```
Entrer le nombre (en base 10) : 1000
Entrer la base (en base 10) : 16
Le voici écrit dans cette base : 3e8
```

En Python toujours, la conversion du décimal `n` au binaire s'écrit `bin(n)` et la conversion du décimal `n` à l'hexadécimal s'écrit `hex(n)`. Ce sont des chaînes de caractères qui sont renvoyées. Par exemple `bin(100)` renvoie la chaîne `'0b1100100'` et `hex(1000)` renvoie `'0x3e8'`.

Application de l'hexadécimal : Codage des couleurs
Les couleurs utilisées en informatique sont codées avec 3 nombres allant de 0 à 255 (donc 1 octet par couleur) pour les trois composantes de couleurs (rouge, vert, bleu dans le système RGB) [1]. Ainsi, (255, 255, 255) code le blanc tandis que (0, 0, 0) code le noir et les nuances de gris s'obtiennent avec le code (n, n, n) où $0 < n < 255$; la couleur pêche se code (253, 191, 183), le bleu ardoise (72, 61, 139) et l'or (255, 215, 0).



1. Il existe quantité de convertisseurs pour les couleurs sur le WEB ainsi que des palettes de toutes sortes, mais consulter cette page `http://irem.univ-reunion.fr/spip.php?article903` pour en savoir plus sur le codage des couleurs

Pour l'ordinateur, une couleur utilise 3 octets, soit 24 bits, mais écrits en hexadécimal cela tient sur 6 caractères. Pour convertir le décimal qui tient sur 1 octet en hexadécimal, on effectue une division euclidienne par 16. Par exemple, pour la composante bleue de la couleur pêche, on obtient le 1er chiffre en effectuant `253//16` en Python (quotient entier), ce qui donne 15 ; le 2e chiffre est obtenu avec le reste `253%16` en Python, ce qui donne 13. La composante est donc codée fd, et on pourrait l'obtenir directement en Python avec `hex(253)` qui donne `'0xfd'`.
Ainsi, on obtient les codes hexadécimaux du blanc 000000, du noir 111111, de la couleur pêche $fdbfb7$, du bleu ardoise $483d8b$ et de l'or $ffd700$.

Codage sur p bits

Les informations manipulées par un ordinateur sont des mots binaires de taille fixe, souvent multiple de 8 bits (1 octet). Or, d'après ce qui précède, sur p bits, on peut stocker des entiers positifs compris entre 0 et $2^p - 1$. Les entiers codés sur 1 octet sont donc compris entre 0 et $2^8 - 1 = 255$. Avec 2 octets on va jusqu'à 65535, sur 4 octets (32 bits) jusqu'à 4 294 967 295 et sur 8 octets (64 bits) jusqu'à 18 446 744 073 709 551 615. Ce nombre parait assez grand pour les usages habituels des nombres mais la limite est invalidante pour certaines applications (arithmétique, cryptographie) qui exigent davantage de bits.
En Python, avant la version 3, il existait deux sortes d'entiers : les `int` qui allaient jusqu'à 2 147 483 648 (environ la moitié du codage sur 4 octets, nous verrons plus loin que cela est lié à la nécessité de stocker des entiers négatifs) et les `long` au delà. Dans Python 3, la taille des entiers `int` n'est plus limitée (pour de très grands nombres, la taille de la mémoire de l'ordinateur est le seul facteur limitant).
Dans d'autres langages que Python, il subsiste différentes catégories d'entiers. En Java par exemple, il y a quatre types d'entiers : `byte` (1 octet), `short` (2 octets), `int` (4 octets) et `long` (8 octets). Le transtypage (changement de type) est automatique quand la taille de l'entier manipulé augmente, mais dans l'autre sens, comme il peut y avoir une perte d'information, elle doit être provoquée par une instruction de `cast` (forcer). Au delà du type `long` en Java, il existe encore la classe `BigInteger` qui permet de manipuler de très grands entiers.

Conversions binaire/octal/hexadécimal

Les nombres binaires manipulés par l'ordinateur étant très longs, on trouve souvent avantage à les convertir en hexadécimal (base 16) ou parfois en octal (base 8).
Les chiffres hexadécimaux se codent sur 4 bits (1 quartet) car le plus grand chiffre hexadécimal f s'écrit avec 4 bits ($f_{(16)} = 15_{(10)} = 1111_{(2)}$) et les chiffres octaux se codent sur 3 bits car $7_{(8)} = 111_{(2)}$.

> Méthode (Conversion binaire/hexadécimal)
> Pour convertir un nombre binaire en hexadécimal, il suffit de regrouper les chiffres binaires par 4, en commençant par la droite, et de remplacer chaque groupe par le chiffre hexadécimal correspondant, en complétant éventuellement le dernier groupe avec des 0 à gauche (le tableau ci-dessous peut être utile pour effectuer cette conversion). Dans le sens inverse, on convertit un nombre hexadécimal en binaire en remplaçant les chiffres hexadécimaux par leur équivalent binaire écrit sur 4 bits. Par exemple :
> $$1111000011000110010_{(2)} = 1111\ 0000\ 1110\ 0011\ 0010 = f0e32_{(16)}$$
> $$11100011000111000111_{(2)} = 0001\ 1100\ 0111\ 0001\ 1100\ 0111 = 1c71c7_{(16)}$$
> $$abcdef_{(16)} = 1010\ 1011\ 1100\ 1101\ 1110\ 1111 = 101010111100110111101111_{(2)}$$

Le principe est le même pour convertir du binaire vers l'octal, mais il faut ici regrouper les chiffres binaires par 3. Par exemple :
$$1111000011000110010_{(2)} = 011\ 110\ 000\ 111\ 000\ 110\ 010 = 3607062_{(8)}$$
$$11100011000111000111_{(2)} = 111\ 000\ 111\ 000\ 111\ 000\ 111 = 7070707_{(8)}$$
$$1234567_{(8)} = 001\ 010\ 011\ 100\ 101\ 110\ 111 = 1010011100101110111_{(2)}$$

Décimal	Binaire 4 bits	Hexadécimal	Binaire 3 bits	Octal
0	0000	0	000	0
1	0001	1	001	1
2	0010	2	010	2
3	0011	3	011	3
4	0100	4	100	4
5	0101	5	101	5
6	0110	6	110	6
7	0111	7	111	7
8	1000	8		
9	1001	9		
10	1010	a		
11	1011	b		
12	1100	c		
13	1101	d		
14	1110	e		
15	1111	f		

Cela n'a pas été mentionné précédemment, mais Python écrit les nombres octaux avec le préfixe `0o` et la conversion en octal est prévue par les fonctions `oct(n)` (du décimal vers l'octal) et `int('0on',8)` (de l'octal vers le décimal). Par exemple, `oct(8)` donne `'0o10'` et `int('0o777',8)` donne `511`.

Application de l'octal : Les droits d'accès sous Linux
Les droits d'accès définissent les actions qu'un utilisateur a le droit d'effectuer sur un fichier : `r` pour *read* (lecture), `w` pour *write* (écriture) et `x` pour *execute* (exécuter), selon qu'il est propriétaire du fichier, membre du groupe propriétaire du fichier ou ni l'un ni l'autre.
Chacun des 9 droits d'accès est stocké sur 1 bit : si le bit vaut 1, le droit correspondant est accordé et s'il vaut 0, le droit correspondant est refusé.
Par exemple : le droit d'accès `rwxr-xr--` indique que le propriétaire du fichier a tous les droits (`rwx`), un membre du groupe propriétaire n'a pas le droit d'écriture (`r-x`) et un utilisateur extérieur au groupe n'a que le droit en lecture (`r--`).
La représentation binaire de ces droits s'écrit 111 101 100 et Linux admet une notation octale qui tient sur 3 chiffres, ici 754. La commande `chmod` qui permet de modifier les droits accepte un argument octal (`chmod` et la contraction de *change mode*, changer les permissions). Ainsi la commande `chmod 754 fileName` modifie en une fois les 9 droits d'accès au fichier `fileName` avec les valeurs de notre exemple et la commande `chmod 777 fileName` en donne tous les droits à n'importe quel utilisateur.

b. Représentation des entiers relatifs

Il existe plusieurs façons de prendre en compte le signe d'un entier relatif.
Historiquement, on a utilisé la méthode du complément à 1, dite aussi du complément logique. Avec ce codage, pour changer de signe, il faut inverser la valeur de chaque bit. Par exemple, 11 s'écrivant 00001011 sur un octet binaire, pour coder -11 on écrivait 11110100. Ce nombre binaire vaut $256 - 11 = 245$, mais avec cette convention, le 1er bit (celui de gauche, dit « bit de poids fort ») indique le signe : si c'est 1, le nombre est négatif, sinon il est positif. Cette représentation avait l'inconvénient d'attribuer deux codages différents au zéro (00000000 et 11111111).
Une autre méthode possible et même plus simple aurait pu être utilisée : garder le bit de poids fort pour le signe et coder sur les bits restants la valeur absolue du nombre. Le nombre -11 s'écrit alors 10001011. L'inconvénient majeur de ce codage est encore la double représentation de zéro (00000000 et 10000000).
La représentation actuelle, utilisée dans tous les processeurs, est la méthode du complément à 2, dite aussi du complément arithmétique.

Définition 1.2 (Entiers signés en compléments à 2) Le codage des entiers relatifs se décompose en deux parties :
- un terme négatif sur un seul bit (le bit de signe)
- un terme correctif, positif (écrit sur tous les bits de poids faible)

Remarques :

- Pour un nombre binaire s'écrivant sur 1 octet $n = a_7a_6a_5a_4a_3a_2a_1a_0$, la décomposition donnant la valeur de n s'écrit $\left[a_0 + a_1 \times 2 + a_2 \times 2^2 + a_3 \times 2^3 + a_4 \times 2^4 + a_5 \times 2^5 + a_6 \times 2^6\right] - a_7 \times 2^7$. Par exemple, le nombre codé 10001100 en complément à 2 sur 1 octet vaut $\left[2^2 + 2^3\right] - 2^7 = -116$. Le même principe s'applique pour les nombres codés sur 2 ou 4 octets, ou plus généralement, sur n'importe quel nombre de bits. Par exemple le nombre codé 1001 en complément à 2 sur 4 bits vaut $\left[2^0\right] - 2^3 = -7$.
- Pour déterminer une représentation en complément à 2 d'un nombre négatif, il suffit d'ajouter 1 au complément logique (le complément à 1) de la valeur absolue de ce nombre. Par exemple, pour coder -7 sur 1 octet, on ajoute 1 au complément à 1 de $7_{(10)} = 00000111_{(2)}$, c'est-à-dire $11111000 + 1 = 11111001$. Vérification : la valeur de 11111001 se calcule, comme l'énonce la définition, $[1 + 8 + 16 + 32 + 64] - 128 = -7$.
- L'opposé d'un nombre se trouve en ajoutant 1 au complément à 1 du nombre. En effet, additionner un nombre et son complément à 1 donne toujours -1 (codé rien qu'avec des 1, quelle que soit la taille de la représentation) ; en ajoutant 1 à cette somme, on obtient bien 0. La méthode indiquée précédemment fonctionne donc dans les deux sens : pour connaître le nombre représenté en complément à 2 par le mot binaire 10111001, on ajoute 1 au complément à 1, soit $01000110 + 1 = 01000111$. Comme $01000111_{(2)} = 71_{(10)}$, on en déduit que le mot binaire 10111001 code le nombre -71.

Capacité de l'encodage en complément à deux sur p bits

Du fait de l'utilisation d'un bit de signe, il ne reste que $p - 1$ bits pour coder les valeurs.

- Les entiers codés sur 1 octet (8 bits) sont compris entre $-2^7 = -128$ et $2^7 - 1 = 127$.
- Les entiers codés sur 2 octets (16 bits) sont compris entre $-2^{15} = -32768$ et $2^{15} - 1 = 32767$
- Les entiers codés sur 4 octets (32 bits) sont compris entre $-2^{31} = -2\,147\,483\,648$ et $2^{31} - 1 = 2\,147\,483\,647$ (un peu plus de 2 milliards)
- Les entiers codés sur 8 octets (64 bits) sont compris entre $-2^{63} = -9\,223\,372\,036\,854\,775\,808$ et $2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807$ (un peu plus de 9 trillions [2]).

On remarque que ces intervalles ne sont pas centrés sur 0 : alors que -128 est représentable sur 1 octet, son opposé 128 ne l'est pas. C'est un défaut mineur à côté de l'unicité de la représentation du zéro que ce codage fait gagner : 0 se code rien qu'avec des 0 et -1 rien qu'avec des 1.

Extension de la capacité d'encodage

Il est possible d'étendre la capacité de représentation d'un codage en ajoutant vers la gauche autant de fois que l'on souhaite le bit de poids fort (le bit de signe).
Par exemple, les entiers codés sur 1 octet sont étendus sur 2 octets en écrivant 8 fois le bit de signe à gauche de la représentation sur 1 octet :

- l'entier positif 01100111 est naturellement étendu sur 2 octets en écrivant 0000000001100111.
- l'entier négatif 11100111 est étendu sur 2 octets en écrivant 1111111111100111. Comment se convaincre qu'il s'agit du même nombre ? L'opposé de 11100111 est $00011000 + 1 = 00011001$ (on ajoute 1 au complément à 1), celui de 1111111111100111 est $0000000000011000 + 1 = 00011001$, ces nombres ayant les mêmes opposés sont donc égaux.

En n'ajoutant rien que des 1 à gauche d'un nombre négatif (son bit de signe est 1), l'opposé n'est pas modifié donc le nombre ne l'est pas non plus.

Addition/soustraction d'entiers relatifs

L'addition sur des entiers relatifs binaires en compléments à 2 suit les mêmes algorithmes que ceux définis pour les entiers naturels : on additionne les bits entre eux en propageant la retenue éventuelle.

2. le trillion vaut un million puissance 3, selon l'échelle longue inventée par Nicolas Chuquet (1450-1488) au XVe siècle et non remise en cause depuis, même si elle est peu utilisée, du fait notamment de la confusion avec l'échelle courte où le trillion vaut seulement un million puissance 2

La somme $1+1=10$ par exemple, conduit à une retenue de 1. Pour effectuer une soustraction, on la transforme en une addition de l'opposé (on ajoute alors le complément à deux). Par exemple pour effectuer la soustraction $12-15$, on effectue l'addition $12+(-15)$. Le complément à 2 sur 1 octet de $15_{(10)}=1111_{(2)}$ étant 11110001, comme $12_{(10)}=1100_{(2)}$, il faut effectuer l'addition 11110001+00001100. Il n'y a aucune retenue dans cette somme, effectuons-la en ligne : on trouve 11111101, soit l'opposé de $00000011_{(2)}=3_{(10)}$, c'est-à-dire -3.
Bien sûr, en procédant sur des mots binaires de taille fixée, ces méthodes peuvent entraîner des dépassements de capacité que l'on peut détecter en examinant le bit de signe : en ajoutant deux nombres positifs, si le bit de signe est 1 cela indique une retenue sortant du dernier rang non nulle et donc un dépassement de capacité. De même, si on ajoute deux nombres négatifs et qu'on obtient un bit de signe nul.
Ci-dessous, quelques exemples d'additions et de soustractions (transformées en additions de l'opposé) binaires sur 1 octet, celles de droite présentant un dépassement de capacité. Les retenues entrantes ont été représentées en rouge, au-dessus des additions, comme il est d'usage de le faire dans les additions décimales. Le bit de signe a été mis en gras pour le test de débordement évoqué plus haut concernant les sommes d'entiers de même signe. Il y a parfois un bit de dépassement qui a été mis entre parenthèses : on n'y prête pas attention, cela ne fausse pas le résultat. Regardez la soustraction $45-27$: pour trouver un résultat positif, le bit de signe du nombre négatif (-27) doit bien être neutralisé par une retenue entrante. Cela provoque une retenue sortante qui n'a aucune signification. Ce n'est pas de la magie, ni de la prestidigitation : c'est de la mathématique !

Additions				
significations	45+27=72	-83+(-37)=-120	-83+(-101)=-184 dépassement	45+91=136 dépassement
retenues	01111110	11111110	01111110	11111110
	00101101	10101101	10101101	00101101
	+00011011	+11011011	+10011011	+01011011
	01001000	(1)**1**0001000	(1)**0**1001000	**1**0001000
Soustractions				
significations	45-27=18	-83-(-37)=-46	101-(-83)=184	5-(-125)=130
transformations	45+(-27)=18	-83+37=-46	101+83=184 dépassement	5+125=130 dépassement
retenues	11011010	01011010	10001110	11111010
	00101101	10101101	01100101	00000101
	+11100101	+00100101	+01010011	+01111101
	(1)**0**0010010	**1**1010010	**1**0111000	**1**0000010

Les débordements de capacité sur les additions ou les soustractions ne prêtent pas à conséquence en Python3, puisque nous avons vu que la capacité d'encodage est agrandie par extension lorsque la nécessité l'exige. Attention, cette facilité n'est pas présente dans tous les langages...

Multiplication d'entiers relatifs

La multiplication sur des entiers positifs binaires suit le même algorithme que celui défini pour les entiers naturels. Ce qui est vraiment commode en binaire : les tables sont extrêmement limitées. Il n'y a qu'une seule table de multiplication (comme il n'y a qu'une seule table d'addition) ! Fini les apprentissages douloureux, le tableau ci-dessous résume tout ce qu'il faut savoir par cœur pour effectuer des multiplications (et des additions).

×	0	1
0	0	0
1	0	1

+	0	1
0	0	1
1	1	10

Les débordements de capacité sur les multiplications arrivent rapidement : en codant les entiers avec 1 octet en complément à 2, les opérandes sont susceptibles d'aller jusqu'à 127 mais pour qu'un débordement intervienne, il suffit que le plus petit des deux opérandes dépasse $\sqrt{127}\approx 11,3$. Ainsi, si l'on effectue 12×13 par exemple, il y aura dépassement de la capacité de représentation $(12\times 13=156>127)$.
Ci-dessous quatre exemples de multiplications sur des entiers codés sur 4 bits. Avec ce type de codage, un débordement survient dès que le plus petit des opérandes dépasse $\sqrt{7}\approx 2,6$ soit dès qu'il dépasse 2. Alors que les opérandes sont susceptibles d'aller jusqu'à 7, on ne peut pas aller au-delà de

2×3 sans déborder. La 2e multiplication concerne un nombre négatif : le résultat est correct, même s'il y a 2 bits de dépassement à ne pas prendre en compte (entre parenthèses).

Multiplications	0011 ×0010 = 0110	0011 ×1110 = (10)1010	0101 ×0011 = 1111	0111 ×0101 = (10)0011
significations	3×2=6	3×(-2) = -6	5×3=15 dépassement	7×5=35 dépassement

2. Représentation des nombres réels

En informatique, il y a deux sortes de nombres : les entiers et les flottants. Dans la vraie vie, c'est plus complexe que cela : pour simplifier, il y a les entiers, les décimaux (dont la partie décimale est finie), les rationnels (quotients de 2 entiers), les réels et les complexes (une partie imaginaire, l'autre réelle). Les flottants apportent une solution pour encoder les nombres réels, mais cette solution a quelques défauts que nous allons envisager.
Un nombre réel est un nombre dont la partie décimale peut être infinie.
Le nombre $\pi = 3,14159265358979323846264338327950288\ldots$ par exemple a une infinité de décimales (je n'en ai écrit que les premières). Ce nombre est typiquement impossible à écrire sous forme décimale ; mais ce n'est pas nécessaire d'aller chercher très loin : le quotient $1 \div 3 = 0,3333\ldots$ (ce nombre est un réel rationnel) n'a pas non plus une écriture décimale finie. Or, les ordinateurs ne peuvent écrire que des nombres dont l'écriture décimale est finie. Tous ces nombres seront donc approchés par la solution flottante ; ils ne seront jamais écrits de façon exacte.
L'écriture s'approchant le plus des nombres flottants est l'écriture scientifique où un nombre est codé sous la forme $sm \times 10^e$, s étant le signe ($+$ ou $-$), m étant la mantisse (un nombre décimal compris entre 1 inclus et 10 exclu) et e étant l'exposant (un entier). Ainsi par exemple, on écrit $12345,6789$ sous la forme $1,23456789 \times 10^4$ et $0,00000987654321$ sous la forme $9,87654321 \times 10^{-6}$.
La 1re différence : les flottants utilisent la base 2 alors que la notation scientifique est en base 10. La 2e différence est la précision qui est forcément limitée en informatique alors qu'elle est illimitée (en principe) pour la notation scientifique.

a. Écriture binaire d'un nombre décimal

Pour comprendre la partie « décimale » d'un nombre non entier en base 2, il faut suivre la même logique que pour la partie entière : le 1er chiffre après la virgule représente le nombre de demis ($2^{-1} = \frac{1}{2} = 0,5$), le 2e chiffre représente le nombre de quarts ($2^{-2} = \frac{1}{2^2} = \frac{1}{4} = 0,25$), le 3e chiffre représente le nombre de huitièmes ($2^{-3} = \frac{1}{2^3} = \frac{1}{8} = 0,125$), etc. Par exemple,
$1,01_{(2)} = \left(1 + 0 \times \frac{1}{2} + 1 \times \frac{1}{4}\right)_{(10)} = 1,25_{(10)}$.

Déterminons l'écriture binaire d'un nombre décimal `nb` à l'aide de l'algorithme suivant :

1. Chercher n la plus grande puissance de 2 supérieure ou égale à 0 que l'on peut enlever à `nb`
2. Répéter tant qu'on le souhaite les instructions suivantes :
   - Si on peut : enlever 2^n au nombre et écrire le chiffre '1', sinon écrire '0'
   - Écrire la virgule si n vaut 0, puis enlever 1 à n

```
nb=float(input("Entrer un nombre décimal "))
n=0
while 2**(n+1) <= nb :
    n += 1
for i in range(52):
    if nb >= 2**n :
        print("1", end="")
        nb -= 2**n
    else :
        print("0", end="")
    if n==0 :
        print(",", end="")
    n -= 1
```

```
Entrer un nombre décimal 1.25
1,01000000000000000000000000000000000000000000000000

Entrer un nombre décimal 5.1
101,00011001100110011001100110011001100110011001100110011
```

Le programme Python ci-dessus implémente cet algorithme : il écrit 52 chiffres de l'écriture binaire du nombre entré. On constate par exemple que $5,1$ n'a pas une écriture binaire finie (la suite de chiffres 0011 se répète) alors que son écriture décimale l'est.

b. Les nombres flottants

DÉFINITION 1.3 (NOMBRE FLOTTANT) Un nombre flottant est un nombre qui s'écrit sous la forme $sm \times 2^e$ qui signifie, dans un encodage sur 64 bits (8 octets) :

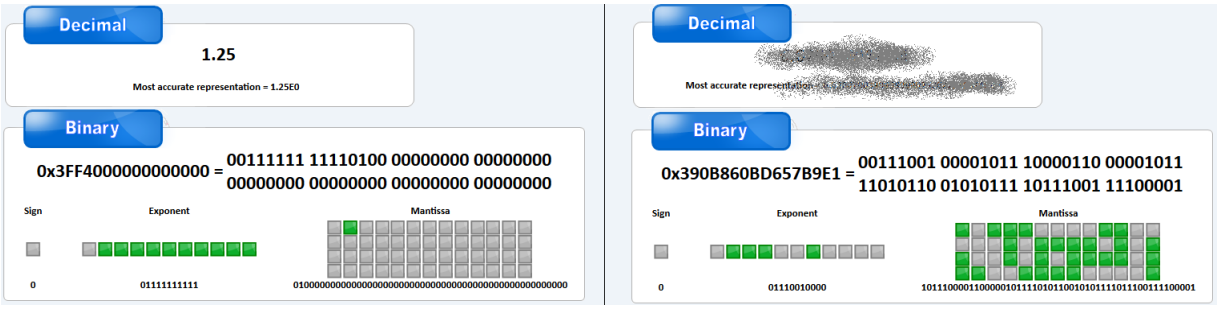
- s est le signe, codé sur 1 bit par 0 pour $+$ et 1 pour $-$
- m est la mantisse, codée sur 52 bits donnant les 52 premières « décimales » binaires qui suivent une partie entière de 1
- e est l'exposant, codé sur 11 bits par la valeur binaire de $e + 1023$

Les bits codant le signe, l'exposant décalé et la mantisse tronquée forment, dans cet ordre, la représentation du nombre.

Remarques :

- L'encodage choisi pour l'exposant n'est pas le complément à 2 mais un décalage de 1023. L'exposant 0 est codé 1023, soit 01111111111 en base 2. Les valeurs possibles pour l'exposant vont de -1022 (codé 1, soit 00000000001) à 1023 (codé 2046, soit 11111111110). Les exposants -1023 et 1024 (codés respectivement 00000000000 et 11111111111) sont réservés pour des usages spéciaux comme coder le 0, les `nan` (*not a number*), $+\infty$ et $-\infty$. Si ce qui code l'exposant est noté e', alors le véritable exposant est $e = e' - 1023$. Ainsi, la valeur du nombre est $sm \times 2^{(e'-1023)}$.
- La mantisse étant toujours supérieure à 1, on ne code pas ce 1. Les chiffres donnés sont ceux de la décomposition en base 2 de la partie « décimale » (celle qui suit la virgule). Par exemple, pour le nombre 1,25 qui s'écrit 1,01 en base 2, la mantisse contient seulement 01 suivis de 50 zéros. Comme l'exposant de 1,25 est égal à 0, ce nombre sera donc codé
  $0 - 01111111111 - 0100000000000000000000000000000000000000000000000000$
  (les tirets pour séparer les trois parties du codage n'ont pas d'existence dans la machine qui sait exactement ce que représente chaque bit).
- L'encodage présenté ici est le plus répandu actuellement. Il s'agit de la norme IEEE 754 qui date de 1985, et plus particulièrement du format appelé « double précision ». Pour coder le 0, l'exposant décalé e' et la mantisse m sont tous les deux nuls. Pour l'infini, l'exposant décalé vaut 1024 et la mantisse m est nulle. Le plus petit nombre positif différent de zéro et le plus grand nombre négatif différent de zéro sont représentés par un exposant décalé $e' = 1$ et une mantisse $m = 0$. Ces nombres valent donc $\pm 2^{-1022} \approx \pm 2,2250738585072014 \times 10^{-308}$.

EXEMPLE 1 – Pour coder ou décoder un nombre décimal sous la forme flottante IEEE 754 double précision exposée ici, il existe de nombreuses applications[3] sur le WEB. L'image ci-dessous montre le résultat pour 1,25 (à gauche) et pour un autre nombre (à droite) que l'on doit déterminer.



Dans cet exemple, le nombre flottant est codé
$0 - 01110010000 - 1011100001100000101111010110010101111011100111100001$.

3. celle-ci est située à l'adresse `http://www.binaryconvert.com/result_double.html`

Il s'agit d'un nombre positif car le bit de signe vaut 0.
L'exposant décalé est $e' = 1110010000_{(2)} = (16 + 128 + 256 + 512)_{(10)} = 912_{(10)}$ donc le véritable exposant est $e = 912 - 1023 = -111$.
La mantisse est codée 10111000011000001011110101100101011110111001111000 01. En ajoutant 1, devant, la véritable mantisse est donc $1,10111000011000001011110101100101011110111001111000 01$, un nombre qui vaut $1 + 2^{-1} + 2^{-3} + 2^{-4} + 2^{-5} + 2^{-10} + 2^{-11} + 2^{-17} + \ldots$ soit, en ne gardant que ces 17 premières décimales, $\frac{2^{17}+2^{16}+2^{14}+2^{13}+2^{12}+2^{7}+2^{6}+1}{2^{17}} = \frac{225473}{131072} \approx 1,720222$
La valeur du nombre est donc environ de $1,7220222 \times 2^{-111} \approx 6,626056 \times 10^{-34}$.
Pas mal, le véritable nombre entré était $6.62607004e-34$, soit $6,62607004 \times 10^{-34}$.
Il s'agit d'une évaluation de la constante de Planck qui joue un rôle central dans la mécanique quantique. Ce nombre est une constante physique exprimée en Js et nommée d'après le physicien Max Planck (1858-1947). Elle est, depuis le 20 mai 2019, à la base de la définition du kilogramme.

c. Erreurs liées aux flottants

L'approximation effectuée par cette représentation des réels est souvent négligeable (heureusement !), mais elle est à l'origine de certains dysfonctionnements qu'il convient de détecter afin d'en limiter les effets. Dans le but d'illustrer ce propos, voici trois exemples sous forme de petits programmes en Python et leur résultat d'exécution.

Exemple 2 – Le 1er programme double la valeur d'une variable x et donne à y la valeur de l'entier suivant x. Logiquement, il y a toujours une distance de 1 entre y et x qui devrait conduire à une belle boucle infinie. Or, il n'en est rien : le programme s'arrête lorsque $x = 9\,007\,199\,254\,740\,992$ (9 trillions, tiens donc !), soit après la 53e étape. Le nombre y est alors égal au nombre x.

```
x = 1.0
y = x+1
i = 0
while y-x == 1 :
    x *= 2
    y = x+1
    i += 1
print("Fini au bout de {} étapes".format(i))
print("x={}, y={}".format(x,y))
```

```
Fini au bout de 53 étapes
x=9007199254740992.0, y=9007199254740992.0
```

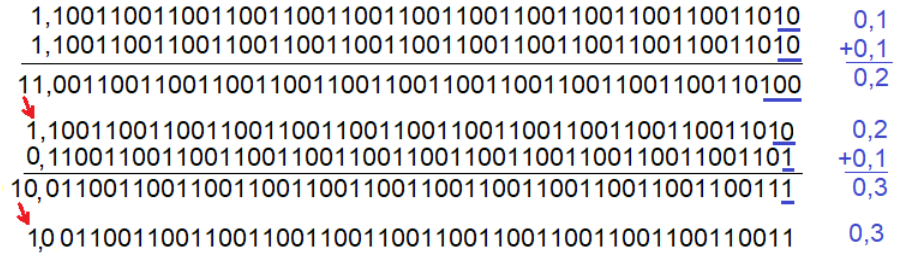
En utilisant l'application mentionnée plus haut, j'obtiens le code de ce nombre flottant (et oui, c'est un flottant du fait de l'écriture 1.0 au départ) :
0 – 10000110100 – 0000000000000000000000000000000000000000000000000000
La mantisse est nulle ! Normal, puisqu'il s'agit d'une puissance de 2. L'exposant de ce nombre est $e = 1076 - 1023 = 53$ et, par conséquent $y = 2^5 3 + 1 = 2^5 3 \times (1 + 2^{-53})$. Il y aurait 1 bit de plus dans la mantisse et cela fonctionnerait encore une étape.
Ce dépassement de capacité de la représentation serait ici corrigé par l'utilisation exclusive d'entiers. En remplaçant le `x=1.0` du début par `x=1`, Python ne voit plus de flottants, rien que des entiers, et le programme ne s'arrête plus : après la 100e étape par exemple, il affiche :
`x=126765060022822940149670320537 6, y=12676506002282294014967032053 77`
Retenons que Python, comme beaucoup d'autres langages, privilégie le type flottant quand il y a un mélange de types. Par exemple `y=x+1` conduit à `y=2.0` lorsque `x=1.0`.

Exemple 3 – Le 2e programme se contente d'ajouter le flottant 0.1 à la valeur de a qui vaut initialement 0. Logiquement, on devrait voir s'afficher les valeurs 0.1, 0.2, 0.3, etc. Cela fonctionne correctement presque partout sauf pour 0.3 qui s'affiche avec un 4 en excès sur la 17e décimale et ensuite 0.8 s'affiche avec un 1 en défaut sur la 16e.

```
a = 0
for n in range(1,9) :
    a += 0.1
    print("a={} au bout de l'étape n°{}".format(a,n))
```

```
a=0.1 au bout de l'étape n°1
a=0.2 au bout de l'étape n°2
a=0.30000000000000004 au bout de l'étape n°3
a=0.4 au bout de l'étape n°4
a=0.5 au bout de l'étape n°5
a=0.6 au bout de l'étape n°6
a=0.7 au bout de l'étape n°7
a=0.7999999999999999 au bout de l'étape n°8
```

Je rentre successivement 0.1, 0.2, 0.3 et 0.30000000000000004 dans l'application qui m'en donne les codes flottants :
$0,1_{(10)} = 0 - 01111111011 - 1001100110011001100110011001100110011001100110011010$
$0,2_{(10)} = 0 - 01111111100 - 1001100110011001100110011001100110011001100110011010$
$0,3_{(10)} = 0 - 01111111101 - 0011001100110011001100110011001100110011001100110011$
$0,30000000000000004_{(10)} = \ 0011001100110011001100110011001100110011001100110100$
Le nombre désigné par la représentation de $0,1$ est en réalité environ égal à $0,1000000000000000055511115123126$ (j'ai utilisé la même application de conversion dans le sens inverse). Il y a un arrondi qui a été fait sur la mantisse où l'on reconnaît la séquence 0011 qui se répète, en principe jusqu'à l'infini : les 3 derniers bits devraient être 001, suivis de 100110011 etc. mais, du fait de l'arrondi, ces bits sont mis à 010. Cette erreur d'arrondi se propage à $0,2$ sans occasionner d'erreur mais, à l'étape suivante, on en obtient une.
J'ai essayé de transcrire les additions qui se font successivement sur l'image ci-dessous. Les flèches rouges indiquent le glissement occasionné par la division par 2, l'exposant augmentant de 1 à chaque addition. J'ai indiqué l'erreur d'arrondi par un soulignement en bleu. Il semblerait, d'après mes calculs, que l'erreur ne provienne pas de l'arrondi si le nombre obtenu pour $0,3$ était simplement tronqué. Les mécanismes d'arrondis du format IEEE 754 ne se limitent pas à une troncature. Pour mieux éclaircir ce défaut, il conviendrait de se reporter aux spécifications de cette norme.



Retenons que les nombres flottants peuvent avoir de légères différences avec les résultats attendus. Ces différences portant sur les dernières décimales (15e ou 16e décimale en notation scientifique décimale), un arrondi des résultats à 12 chiffres nous assurerait de leur exactitude dans ce cas précis. On ne peut donc pas tester l'égalité entre ce type de résultat flottant et une valeur décimale exacte, par exemple en écrivant la boucle `while a!=0.3` car cette valeur qui, logiquement devrait arriver, peut ne jamais arrive en réalité du fait de la notation flottante.

Exemple 4 – Le 3e programme calcule la somme des inverses des entiers jusqu'à un certain point. Cette somme $S_n = \sum_{i=1}^{n} \frac{1}{i}$ augmente progressivement quand n augmente, mais de plus en plus lentement. En mathématiques, on prouve que lorsque n prend des valeurs suffisamment grandes, S_n finit par dépasser n'importe quelle valeur fixée arbitrairement (on note cela $\lim_{n\to+\infty} S_n = +\infty$).

```
n=int(input("Calculer la somme jusqu'à quel nombre ? "))
somme = 0
somme_inv = 0
for i in range(1,n+1) :
    somme += 1/i
    somme_inv += 1/(n+1-i)
print("S({})={}".format(n,somme))
print("Z({})={}".format(n,somme_inv))
```

```
Calculer la somme jusqu'à quel nombre ? 1000000
S(1000000)=14.392726722864989
Z(1000000)=14.392726722865772
```

n	S programme	S réelle	Z programme
1	1.0	1.0000000000000000000	1.0
10	2.928968253968253 8	2.928968253968253 96825	2.928968253968253 8
100	5.187377517639621	5.187377517639620 26080	5.187377517639621 5
1000	7.485470860550343	7.485470860550344 91265	7.485470860550341
10000	9.787606036044348	9.787606036044382 26417	9.787606036044386
100000	12.090146129863335	12.090146129863427 94736	12.090146129863408
1000000	14.392726722864989	14.392726722865723 63138	14.392726722865772

Ceci dit, on s'aperçoit que les valeurs de S_n s'écartent de plus en plus de la valeur exacte que le tableau ci-dessus présente, tronquée à 16 chiffres significatifs (les chiffres qui diffèrent sont colorés en magenta). On peut comprendre là que les erreurs d'arrondis finissent par se ressentir. Cela ne parait

pas gênant outre mesure sauf que si on continue (on ne va pas le faire car c'est très long, il faut aller très loin en double précision), les valeurs de S_n vont se stabiliser sur une certaine valeur, suffisamment grande pour que l'infime quantité ajoutée soit tout simplement éliminée par le manque de chiffres de la mantisse.
Ce phénomène est tout de même très fâcheux et il faudra bien se garder d'en tirer la conclusion que la suite des valeurs de S_n finit par devenir constante...
Cet exemple montre également une autre particularité de l'addition flottante : elle ne respecte pas les règles d'associativité et de commutativité de l'addition. En effet, en calculant la somme à l'envers, soit en calculant $Z_n = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} + \frac{1}{1}$ alors que $S_n = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n-1} + \frac{1}{n}$, on ne trouve pas le même résultat (comparer les colonnes 2 et 4).

Ce dernier phénomène apparait dans des situations plus simples encore :
Si on calcule avec des flottants $2^{60} - 2^{60} + 1$ (en Python `2**60-2**60+1.0`), on trouve `1.0`, ce qui est correct. Mais si on calcule $2^{60} + 1 - 2^{60}$ (en Python `2**60+1.0-2**60`), on trouve `0.0`, ce qui est faux. Il s'agit d'un dépassement de capacité par le bas (*underflow*) où les bits de moindre poids sont sacrifiés pour maintenir ceux de poids fort.
Les nombres flottants génèrent toutes sortes d'autres incongruités de ce genre.
Vous pouvez consulter cette page [4] de la documentation Python pour en découvrir quelques autres.

3. Représentation des textes

Aperçu historique :
Dans différents domaines (télécommunications, cryptographie, informatique) on a eu besoin d'associer les caractères d'un alphabet a d'autres symboles, généralement des nombres.

Le code Morse [5] par exemple est un système permettant de coder un texte par des impulsions courtes et longues, qu'elles soient produites par des signes, une lumière, un son ou un geste. Ce précurseur des codes numériques de télécommunication est encore utilisé aujourd'hui. Qui ne connait le fameux signal de détresse tititaaataaataaatititi (à utiliser de préférence après un naufrage) ou le fameux tititaaa du début de la 5e de Beethoven, diffusée sur Radio Londres en juin 1944 pour annoncer le débarquement (un 'V' comme victoire) ?
Le code Baudot [6] datant de 1874, est un des premiers codages de textes en binaire : chaque caractère est codé par une série de 5 bits, ce qui permet $2^5 = 32$ combinaisons. Il utilise deux jeux de caractères appelés Lettres (*Lower Case*) et Chiffres (*Upper Case*). Ce système a, par la suite, évolué en passant progressivement à 8 bits, pour représenter jusqu'à 256 symboles.

En informatique, avant la standardisation des années 60, de nombreux codages de caractères incompatibles entre eux existaient. En 1963, la première version publiée de l'ASCII (*American Standard Code for Information Interchange*) apparaît. Au début, cette norme était en concurrence avec les standards incompatibles, mais progressivement elle s'impose partout jusqu'à devenir la norme de codage de caractères la plus influente à ce jour. IBM, par exemple, qui utilisait initialement ses propres codages ne commença à l'utiliser sur ses matériels qu'avec l'IBM PC, en 1981.

4. https ://docs.python.org/fr/3.7/tutorial/floatingpoint.html
5. Samuel Finley Breese Morse (1791-1872) utilise ce code pour son télégraphe dont il dépose le brevet en 1840
6. Jean Maurice Émile Baudot (1845-1903) est l'inventeur du code portant son nom, utilisé par les téléscripteurs du réseau Télex

a. Code ASCII

L'ASCII définit 128 codes à 7 bits, comprenant 95 caractères imprimables : les chiffres arabes de 0 à 9, les lettres minuscules et capitales de A à Z, et des symboles mathématiques et de ponctuation. Les ordinateurs travaillant presque tous sur un multiple de 8 bits (octet) depuis les années 1970, chaque caractère d'un texte en ASCII est souvent stocké dans un octet dont le bit de poids fort est 0.

Les caractères de numéro 0 à 31 et le 127 ne sont pas affichables. Ils correspondent à des commandes de contrôle, comme le saut de ligne (le retour charriot, noté CR pour *Charriot Return* dans la table ci-contre qui date de 1972) qui porte le numéro 13 (00001101 en binaire) ou le bip sonore (BEL).
Pour connaître le numéro d'un caractère de cette table : multiplier par 16 la colonne et additionner la ligne. Par exemple, la touche d'espace (noté SP pour *Space*) porte le numéro $2 \times 16 + 0 = 32$ (00100000 en binaire), l'esperluette qui est dans la même colonne porte le numéro $2 \times 16 + 6 = 38$ (00100110 en binaire).

USASCII code chart

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

Les codes 48 à 57 représentent les chiffres de 0 à 9 (48 code le 0 et 57 le 9).
Les codes 65 à 90 représentent les majuscules de A à Z (65 code le A et 90 le Z).
Les codes 97 à 122 représentent les minuscules de a à z (97 code le A et 122 le Z).

En Python, pour connaître le code ASCII d'un caractère `c`, il suffit de taper `ord(c)`.
Par exemple `ord('A')` renvoie `65` tandis que `ord('9')` renvoie `57`.
Inversement, pour connaître le caractère qui correspond à un code ASCII `n`, il suffit de taper `chr(n)`.
Par exemple `chr(35)` renvoie `'#'` tandis que `chr(67)` renvoie `'C'`.

Exemple 5 – Que signifie ce « texte »
0100001001101111011011100110101001101111011101010111001000100000000100001, codé en ASCII ?
Il faut découper ce texte en octets et retrouver les caractères avec la table.

octet	colonne		ligne		caractère
01000010	100	4	0010	2	B
01101111	110	6	1111	15	o
01101110	110	6	1110	14	n
01101010	110	6	1010	10	j
01101111	110	6	1111	15	o
01110101	111	7	0101	5	u
01110010	111	7	0010	2	r
00100000	010	2	0000	0	SP
00100001	010	2	0001	1	!

Remarque : cet exercice élémentaire et fastidieux gagnerait à être effectué par un programme.

Dans le sens contraire, comment est codé en ASCII la phrase « La nuit porte conseil » ?
Le petit programme ci-dessous répond, pour cette phrase comme pour toutes les autres.

```
def inBaseDeux(n):
    octet = ""
    for i in range(8):
        if n >= 2**(7-i) :
            octet += "1"
            n -= 2**(7-i)
        else :
            octet += "0"
    return octet

texte="La nuit porte conseil."
for lettre in texte:
    print(inBaseDeux(ord(lettre)),end="")
```

```
01001100011000010010000001101110011101010110100101110100001000000111000001101111011100100111010001100101001000000110001101101111011100111001101100101011010010110110000010111 0
```

b. Code ISO-8859-1

Le codage ASCII, initialement prévu pour écrire l'anglais, ne comporte aucun caractères accentués. Le bit de poids fort étant disponible pour l'étendre de 128 à 256 caractères maximum, quelques normes incluant des accents et autres signes diacritiques (la cédille, le tilde, etc.) sont apparues dont l'ISO-8859-1 qui est aussi appelée *latin-1*. Ce standard convient à la plupart des langues latines puisqu'il ajoute aux caractères ASCII imprimables les caractères qui lui manquaient. En français cependant, il manque le e dans l'o, les caractères œ et Œ, et aussi le symbole de l'euro € qui n'existait pas lors de la normalisation de ce système.
L'extrait de table qui suit montre les ajouts apportés par cet encodage. La norme ISO/CEI-8859-1 présentée ici ne comporte aucun caractère de contrôle, contrairement à l'ISO-8859-1 qui inclus ceux de l'ASCII en y ajoutant une nouvelle série (pour les codes commençant en hexadécimal par 8 et 9).

ISO/CEI 8859-1																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

La position A0 en hexadécimal (160 en décimal) est assignée au caractère de l'espace insécable (noté NBSP, pour *no-break space*).
La position AD en hexadécimal (173 en décimal) est assignée au caractère de formatage indiquant la position d'une possible coupure de mot, normalement invisible, n'apparaissant que si un saut de ligne est réalisé à cette position (noté parfois SHY pour *soft hyphen*).
Les symboles mathématiques sont encore assez pauvres, mais on voit apparaître les signes de la multiplication (D7 en hexadécimal) et de la division (F7 en hexadécimal) ainsi que le plus ou moins (B1 en hexadécimal).

En Python, pour obtenir le code ISO-8859-1 d'un caractère `c`, on peut taper `ord('c')` ou `list('c'.encode('latin1'))` (`ord` donne le code unicode qui coïncide au *latin* − 1 jusqu'à 255). Par exemple `list('A'.encode('latin1'))` renvoie `[65]`. Le code du caractère est renvoyé dans une liste car Python utilise Unicode qui nécessite parfois plus d'un octet pour représenter un seul caractère. Remarquons encore que les codes ASCII et ISO-8859-1 sont identiques pour les lettres non accentuées et les chiffres.
Un exemple de lettre accentuée : `list('è'.encode('latin1'))` renvoie `[232]`. Pour notre œ, par contre, on obtient le message d'erreur suivant :
`'latin-1' codec can't encode character '\u0153' in position 0: ordinal not in range(256)`.

Variantes de l'ISO-8859

La norme ISO-8859 décline d'autres jeux de caractères dont l'ISO-8859-15 appelé *latin-9*. Ce jeu est presque identique au *latin-1* car il ne diffère de celui-ci que sur 8 caractères. Il ajoute les caractères qui faisaient défaut en français : €, œ et Œ. En Python, la commande `list('œ'.encode('latin9'))` renvoie `[189]`, soit BD en hexadécimal. Ce caractère prend donc la place du caractère ½ en *latin-1* .

Position	0xA4	0xA6	0xA8	0xB4	0xB8	0xBC	0xBD	0xBE
8859-1	¤	¦	¨	´	¸	¼	½	¾
8859-15	€	Š	š	Ž	ž	Œ	œ	Ÿ

L'ISO-8859-2 appelé *latin-2* convient à la plupart des langues d'Europe centrale ou de l'Est basées sur l'alphabet romain (bosnien, croate, polonais, tchèque, slovaque,slovène et hongrois).

L'ISO-8859-5 couvre la plupart des langues slaves utilisant un alphabet cyrillique (biélorusse, bulgare, macédonien, russe, serbe et ukrainien).
L'ISO-8859-6 couvre les caractères les plus courants de l'arabe. Les caractères grecs apparaissent dans l'ISO-8859-7 et ceux de l'hébreu dans l'ISO-8859-8.

c. Unicode

Malgré cette multiplication des normes, certaines langues comme le chinois qui nécessite plusieurs milliers de caractères, ne peuvent se satisfaire d'un encodage sur 1 seul octet. Le principe de la norme Unicode est de réunir tous les langages dans une seule table.
Sa 1re édition remonte à 1991. Il constitue un répertoire de près de 140 000 caractères aujourd'hui, couvrant une centaine d'écritures. Toutes ne sont pas encore présentes, mais les écritures les plus utilisées dans le monde sont représentées.
Chaque caractère de ce gigantesque jeu tient sur 21 bits maximum. Un caractère Unicode (point de code) est noté U+xxxx[xx] où xxxx[xx] est un nombre hexadécimal de 4 à 6 chiffres. Les points de code utilisant 4 chiffres hexadécimaux appartiennent au 1er plan. Ce plan, nommé plan de base multilingue, contient à lui seul la majeure partie des caractères utilisés dans toutes les langues du monde mais aussi des symboles. Il y a 17 plans, qui permettent chacun d'attribuer 65 536 points de codes, soit environ 1,1 million de valeurs possibles.
Les points de codes dans l'intervalle U+0000 à U+007F sont identiques à ceux de l'ASCII et ceux dans l'intervalle U+0000 à U+00FF (les 256 premiers) sont identiques à ceux de l'ISO-8859-1 (Latin-1). Grâce à cette compatibilité, les systèmes d'encodage ASCII et ISO-8859-1 sont tous les deux capables de transformer un sous-ensemble restreint de points de codes Unicode en octets. Le défaut d'Unicode est qu'il est plus lent et prend plus de place que d'autres représentations du même texte. Dans la pratique, il est utilisé indirectement dans le format UTF qui s'écrit principalement sur 1 octet pour sa version UTF-8.

UTF-8

Créé en 1992, l'UTF-8 est l'un des encodages les plus couramment utilisés.
Python 3 l'utilise pour représenter ses chaînes de caractères et pour écrire des fichiers. La plupart des sites WEB font de même aujourd'hui. La bonne pratique est d'utiliser UTF-8 dès que c'est possible, afin de garantir une bonne interopérabilité.
UTF signifie *Unicode Transformation Format* (format de transformation Unicode) et 8 signifie que des nombres codés sur 8 bits (donc inférieurs à 256) sont utilisés dans l'encodage. Il existe également les codages UTF-16 et UTF-32, mais ils sont moins souvent utilisés que UTF-8 : avec l'UTF-32, chaque caractère est toujours encodé sur 4 octets ce qui fait beaucoup de gâchis.
UTF-8 utilise les règles suivantes :

- Si le code est inférieur à 128, il est représenté par la valeur de l'octet correspondant.
- Si le point de code est supérieur ou égal à 128, il est transformé en une séquence de deux, trois ou quatre octets, où chaque octet de la séquence est compris entre 128 et 255.

Une chaîne de caractères ASCII est un texte UTF-8 valide. Les caractères appartenant à l'ASCII ont les mêmes codes en UTF-8, en ASCII et en latin-1 : le code de 'A' est 65 dans ces trois encodages. Tapez `list('A'.encode('utf8'))`, `list('A'.encode('latin1'))` ou `list('A'.encode('ascii'))` dans une console Python, vous obtiendrez toujours `[65]`.
Par contre, les caractères latin-1 qui ne sont pas dans l'ASCII n'ont pas les mêmes codes en UTF-8 qu'en latin-1 : 'é' a le code 233 en latin-1 (soit E9 en hexadécimal) mais en UTF-8, selon la règle, ce caractère est codé sur deux octets : tapez `list('é'.encode('utf8'))` dans une console Python, vous obtiendrez `[195, 169]` ce qui correspond en hexadécimal aux nombres C3 et A8. On peut ainsi obtenir ce caractère en tapant `'\xE9'` (codage du point de code en hexadécimal) ou bien `'\u00E9'` (codage du point de code selon la norme UTF-8).

d. Fichiers textes

Un fichier est une séance d'octets. Le nom du fichier, son extension, le dossier auquel il appartient sont des caractéristiques du système de fichier utilisé. Celui-ci dépend du système d'exploitation. Un fichier est dans un certain format lorsqu'il suit une spécification particulière indiquant dans quel ordre doivent se succéder les différents octets du fichier, et quelle est leur signification. Un format ouvert est complètement documenté, contrairement à un format fermé. Attention de ne pas confondre un fichier texte (indépendamment de son encodage) et un fichier issu d'un logiciel de traitement de texte : il n'y a absolument rien en commun dans le contenu de ces deux types de fichier !
Le format texte encodé en latin-1 ou UTF-8 est un des plus simples formats de fichier. Il contient seulement les codes des caractères composant le texte. On peut y enregistrer des textes ou des nombres mais il ne s'agit finalement que de suites de caractères.
Sur le disque dur de l'ordinateur, un fichier est une suite de blocs, chaque bloc ayant l'adresse du bloc suivant. L'espace de stockage a été fragmenté afin de pouvoir ajouter ou supprimer des fichiers de tailles différentes ou aussi de compléter des fichiers sans avoir à tout re-écrire (sans écraser le fichier préexistant). Il est impossible d'insérer des éléments dans un fichier : on ne peut les ajouter qu'à la fin. La lecture d'un fichier s'effectue bloc par bloc. La tête de lecture voyage dans le disque, permettant un accès progressif au fichier. La lecture/écriture sur un disque est une opération coûteuse en temps pour laquelle il est nécessaire d'avoir une mémoire tampon, ce qui rend asynchrone la lecture et l'écriture.

4. Représentation booléennes

Le type de données envisagé ici est le type booléen. Nommé d'après l'inventeur de l'algèbre qui porte son nom [7], ce type de donnée ne prend que 2 valeurs : oui ou non, vrai ou faux, True ou False pour les anglophones, 1 ou 0 pour les ordinateurs. Une donnée booléenne ne tient donc que sur un seul bit. Ce qui nous importe ici est de comprendre les fonctions booléennes : qui associent un booléen à un ou plusieurs booléens. Une fonction booléenne se définit d'une façon symbolique (comme les fonctions numériques qui utilisent des symboles opératoires) mais aussi à l'aide d'une table, dite table de vérité. Nous allons en donner quelques exemples fondamentaux.

a. Fonctions booléennes de base

La plus simple des fonctions booléennes est la fonction *non*.
Elle associe vrai à faux et faux à vrai. On pourrait l'appeler fonction *contraire* ou *complment* ou encore *not* (c'est d'ailleurs son nom en Python). Ainsi, $non(0) = 1$ et $non(1) = 0$.
En Python `not(True)` donne `False` et `not(False)` donne `True`.
Comme le test de l'égalité est noté `==` (2 symboles d'égalité), l'instruction `a%2==0` donnera `True` si et seulement si le contenu de `a` est pair (`a%2` désigne le résultat de la division euclidienne de `a` par 2).
La table de vérité de cette fonction est donnée ci-dessous (à gauche).

NON	
x	non(x)
0	1
1	0

ET		
x	y	x et y
0	0	0
0	1	0
1	0	0
1	1	1

OU		
x	y	x ou y
0	0	0
0	1	1
1	0	1
1	1	1

La fonction booléenne *et* agit sur deux booléens, noté x et y.
Pour que $et(x, y) = 1$, il faut que $x = 1$ **et** $y = 1$. Dans tous les autres cas $x\ et\ y = 0$.
On note plutôt $et(x, y)$ sous la forme $x\ et\ y$, exactement comme on note $x + y$ plutôt que $+(x, y)$.
En Python, cette fonction est notée `and` et on a, par exemple, `True and True` qui donne `True`.
Pour être plus concret, `a<0 and b==1` donnera `True` si et seulement si le contenu de `a` est négatif **et** le contenu de `b` vaut 1. La table de vérité de la fonction *et* est donnée ci-dessus (au centre).

7. George Boole (1815-1864) est un logicien, mathématicien et philosophe qui modernisa la logique en la fondant sur une structure algébrique.

La fonction booléenne *ou*, comme la fonction *et*, agit sur deux booléens.
Pour que $ou(x, y) = 1$, on note plutôt $x \ ou \ y = 1$, il faut que l'un au moins des deux booléens soit vrai. C'est plus simple de définir cette fonction en disant que, pour que $ou(x, y) = 0$, plutôt noté $x \ ou \ y = 0$, il faut que $x = 0$ **et** $y = 0$.
En Python, cette fonction est notée `or` et on a, par exemple, `True or True` qui donne `True` et, plus concrètement, `a<0 or a>1` donnera `False` si et seulement si le contenu de `a` est contenu entre 0 inclus et 1 inclus. La table de vérité de la fonction *ou* est donnée ci-dessus (à droite).

PROPRIÉTÉ 1.1 (LOIS DE MORGAN) Pour tout couple (x, y) de bits, on a $non(x \ et \ y) = non(x) \ ou \ non(y)$ et $non(x \ ou \ y) = non(x) \ et \ non(y)$. Autrement dit :

1. l'expression $x \ et \ y$ peut être remplacée par l'expression $non(non(x) \ ou \ non(y))$
2. l'expression $x \ ou \ y$ peut être remplacée par l'expression $non(non(x) \ et \ non(y))$

DÉMONSTRATION Pour montrer ces propriétés, il faut dresser les tables de vérité de $non(non(x) \ ou \ non(y))$ et $non(non(x) \ et \ non(y))$ et les comparer avec celles de $x \ et \ y$ et $x \ ou \ y$.
Voici la table de $non(non(x) \ ou \ non(y))$, dressée progressivement en utilisant d'abord la fonction *non*, puis la fonction *ou* et enfin la fonction *non* à nouveau.

NON(NON(x) OU NON(y))					
x	y	non(x)	non(y)	non(x) ou non(y)	non(non(x) ou non(y))
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

On constate que pour chaque couple (x, y) possible, les valeurs de vérité coïncident dans cette table et dans celle de $x \ et \ y$ (donnée plus haut) : ces deux expressions booléennes sont donc équivalentes.
Je laisse la 2e démonstration à titre d'exercice.

Conséquence : Il est possible de se passer de la fonction *et* puisque chaque fois que cette fonction est nécessaire, on peut la remplacer par une autre qui ne contient que *non* et *ou*.
Si on conserve la fonction *et*, par contre, on pourra se passer de la fonction *ou*.

La fonction *ou* est un *ou* inclusif qui apparait dans l'expression « je serai là, qu'il pleuve ou qu'il vente » (je serai là aussi s'il pleut et vente, simultanément). Le *ou* exclusif de la langue, qui apparait sur un menu de restaurant dans l'expression « dessert ou café » (sous-entendu, ce n'est pas prévu de commander les deux), est une fonction qui peut se noter *oux* ou *xor* en anglais/informatique.
La table de vérité du *ou* exclusif est donnée ci-dessous.

OUX (XOR)		
x	y	x et y
0	0	0
0	1	1
1	0	1
1	1	0

Quelle est l'expression du *oux* à l'aide des fonctions de base ?
D'après la table, pour avoir $x \ oux \ y = 1$, il faut et il suffit d'avoir $non(x) \ et \ y$ ou $x \ et \ non(y)$, d'où l'équivalence $x \ oux \ y = (non(x) \ et \ y) \ ou \ (x \ et \ non(y))$
On peut s'amuser à traduire cette expression sans le *et* :
$x \ oux \ y = [non(non(non(x)) \ ou \ non(y))] \ ou \ [non(non(x) \ ou \ non(non(y)))]$
Pour un humain, ce n'est pas très lisible, il faut bien l'avouer.
Mais pour un ordinateur c'est clair comme de l'eau de roche.

b. Algèbre de Boole

On appelle *addition* la fonction *ou* et *multiplication* la fonction *et*.

On définit l'ordre $0 < 1$ sur les éléments booléens.

Cet ordre permet de redéfinir
- le *ou* $(+)$: $x \ ou \ y = x \ + \ y = \max(x, y)$
- le *et* $(\times)$: $x \ et \ y = x \ \times \ y = \min(x, y)$

Les propriétés, connues pour la plupart dans les ensembles de nombres, de l'addition et de la multiplication se retrouvent ici, ce qui justifie l'emploi de ces symboles :

❊ Commutativité :
$x + y = y + x$, le *ou* $(+)$ est commutatif.
$x \times y = y \times x$, le *et* $(\times)$ est commutatif.

❊ Associativité :
$(x + y) + z = x + (y + z)$, le *ou* $(+)$ est associatif.
$(x \times y) \times z = x \times (y \times z)$, le *et* $(\times)$ est associatif.
Notez bien que tout cela se démontre à l'aide de tables de vérité, sans aucune difficulté.

❊ Élément neutre :
On appelle « élément neutre » d'une opération, un élément particulier de l'ensemble sur lequel porte l'opération qui ne modifie pas l'autre élément.
Comme $x + 0 = 0 + x = x$, 0 est l'élément neutre de *ou* $(+)$.
Comme $x \times 1 = 1 \times x = x$, 1 est l'élément neutre de *et* $(\times)$.

❊ Élément absorbant :
On appelle « élément absorbant » d'une opération, un élément particulier de l'ensemble sur lequel porte l'opération qui ne modifie pas l'autre élément.
Comme $x + 1 = 1 + x = 1$, 1 est l'élément absorbant de *ou* $(+)$.
Comme $x \times 0 = 0 \times x = 0$, 0 est l'élément absorbant de *et* $(\times)$.

❊ Distributivité :
Il y a deux sortes de distributivité (pour les nombres, il n'y en a qu'une, c'est $\times$ qui est distributive par rapport à $+$ comme dans l'exemple de cette égalité $3 \times (5 + 6) = 3 \times 5 + 3 \times 6$) :
$x \times (y + z) = (x \times y) + (x \times z)$, $\times$ est distributive par rapport à $+$
$x + (y \times z) = (x + y) \times (x + z)$, $+$ est distributive par rapport à $\times$

❊ Complément :
En notant $\bar{x}$ le contraire/complément de x, on constate que $x + \bar{x} = 1$ et $x \times \bar{x} = 0$.
le complément est involutif car $\bar{\bar{x}} = x$.

❊ Idempotence :
L'addition et la multiplication sont idempotentes car $x + x = x$ et $x \times x = x$.
Cette propriété permet des simplifications comme par exemple $x \times (x + y) = x + x \times y$.

❊ Priorité :
Pour faciliter leur compréhension, on a décidé que ces opérations seraient soumises aux mêmes règles que les opérations sur les nombres, la fonction *et*, la multiplication logique, est ainsi prioritaire par rapport à la fonction *ou*, la somme logique. On peut, bien sûr, pour s'aider, continuer à placer des parenthèses dans les opérations même quand celles-ci ne sont pas nécessaires.
Le programme ci-dessous expérimente cette question avec Python puisqu'il permet de comparer les tables de vérité des trois expressions : $x + y \times z$, $x + (y \times z)$ et $(x + y) \times z$

```
for i in [0,1]:
    for j in [0,1]:
        for k in [0,1]:
            print("{} or {} and {} = {}".format(i,j,k,i or j and k),end='')
            print(" - {} or ({} and {}) = {}".format(i,j,k,i or (j and k)),end='')
            print(" - ({} or {}) and {} = {}".format(i,j,k,(i or j) and k))
```

```
0 or 0 and 0 = 0 - 0 or (0 and 0) = 0 - (0 or 0) and 0 = 0
0 or 0 and 1 = 0 - 0 or (0 and 1) = 0 - (0 or 0) and 1 = 0
0 or 1 and 0 = 0 - 0 or (1 and 0) = 0 - (0 or 1) and 0 = 0
0 or 1 and 1 = 1 - 0 or (1 and 1) = 1 - (0 or 1) and 1 = 1
1 or 0 and 0 = 1 - 1 or (0 and 0) = 1 - (1 or 0) and 0 = 0
1 or 0 and 1 = 1 - 1 or (0 and 1) = 1 - (1 or 0) and 1 = 1
1 or 1 and 0 = 1 - 1 or (1 and 0) = 1 - (1 or 1) and 0 = 0
1 or 1 and 1 = 1 - 1 or (1 and 1) = 1 - (1 or 1) and 1 = 1
```

On constate que $x + y \times z = x + (y \times z)$ alors que $x + y \times z \neq (x + y) \times z$ ce qui confirme que les opérateurs `and` et `or` de Python suivent bien la règle de priorité donnée plus haut.

c. Multiplexeur

La fonction de multiplexage mux agit sur trois booléens, notés x, y et z.
Cette fonction est définie par les deux conditions : $mux(0, y, z) = y$ et $mux(1, y, z) = z$.
Cette définition conduit à dresser la table de cette fonction. On peut alors dresser une 2e table, celle de l'expression $(non(x)\ \ et\ \ y)\ \ ou\ \ (x\ \ et\ \ z)$ qui peut aussi s'écrire $\bar{x} \times y + x \times z$ (parenthèses inutiles du fait de la priorité de $\times$ sur $+$).

MUX							
x	y	z	mux(x,y,z)	non(x)	non(x) et y	x et z	(non(x) et y) ou (x et z)
0	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	1	1	0	1
0	1	1	1	1	1	0	1
1	0	0	0	0	0	0	0
1	0	1	1	0	0	1	1
1	1	0	0	0	0	0	0
1	1	1	1	0	0	1	1

On constate que ces deux tables sont égales, ce qui signifie que $mux(x, y, z) = \bar{x} \times y + x \times z$.

Cette fonction permet, au niveau du processeur, de définir une commande qui conduit, en sortie, la valeur de l'entrée y (si la commande $x = 0$) ou la valeur de l'entrée z (si la commande $x = 1$).
Nous aurons l'occasion de revenir sur ce point dans le chapitre 5 (Architecture d'un ordinateur).

Une autre application, théorique cette fois, de cette fonction : elle permet de montrer que toutes les fonctions booléennes peuvent s'exprimer avec seulement les fonctions *non*, *et* et *ou*. Pour montrer cela, on procède par récurrence sur le nombre de booléens en entrée :

- S'il n'y a qu'un seul booléen, il n'y a que 4 fonctions possibles : la fonction *non* ; la fonction identité *id* ($id(0) = 0$ et $id(1) = 1$) ; la fonction constante y ($y(0) = 1$ et $y(1) = 1$) ; la fonction constante n ($n(0) = 0$ et $n(1) = 0$). Toutes ces fonctions s'expriment avec *non*, *et* et *ou* : pour *non*, c'est trivial ; *id* n'utilise aucune fonction car $id(x) = x$; pour n, on a $n(x) = x\ \ et\ \ non(x)$; pour y, on a $y(x) = x\ \ ou\ \ non(x)$.
- En supposant que toutes les fonctions booléennes ayant N booléens en entrée peuvent s'exprimer avec seulement les fonctions *non*, *et* et *ou*. Pour exprimer une fonction f ayant $N + 1$ booléens en entrée, il suffit de trouver des fonction g_1 et g_2 ayant N booléens en entrée, telles que :
  — $g_1(x_1, x_2, \ldots, x_N) = f(x_1, x_2, \ldots, x_N, 0)$
  — $g_2(x_1, x_2, \ldots, x_N) = f(x_1, x_2, \ldots, x_N, 1)$
  Comme on sait par hypothèse exprimer g_1 et g_2 avec seulement *non*, *et* et *ou*, on peut aussi le faire pour f puisqu'on sait exprimer *mux* avec seulement *non*, *et* et *ou* et que $f(x_1, x_2, \ldots, x_N, x_{N+1}) = mux(x_{N+1}, g_1(x_1, x_2, \ldots, x_N), g_2(x_1, x_2, \ldots, x_N)))$.

Conclusion : comme on sait le faire pour $N = 1$ alors on sait le faire pour $N = 1 + 1 = 2$, et de proche en proche, on sait le faire pour tout entier N.

Cette démonstration fournit également une méthode pour trouver l'expression symbolique d'une expression connue par sa table de vérité :

La fonction *oux* par exemple est connue, au départ, par sa table de vérité car $x \ oux \ y = 1$ si $x = 1$ ou $y = 1$ mais pas les deux en même temps.
On dérive de cette fonction les fonctions $g_1(x) = oux(x, 0)$ et $g_2(x) = oux(x, 1)$ qui sont simples à trouver : il s'agit de *id* pour g_1 et de *non* pour g_2 (le vérifier).
On en déduit que

$$\begin{aligned} x \ oux \ y &= mux(y, g_1(x), g_2(x)) \\ &= mux(y, x, non(x)) \\ &= (non(y) \ et \ x) \ ou \ (y \ et \ non(x)) \end{aligned}$$

Ce résultat est bien celui qui a été obtenu précédemment par une autre méthode.