*1ère NSI — Thème 4: Représentation des données*

# Types & Valeurs de base

Lycée Fustel de Coulanges, Massy

Marc Biver, novembre 2023, *v0.2*

*Ce document reprend les notions abordées en cours et pratiquées en TP / TD; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.*

# Table des matières

# 1 Point d'étape – où est-on / où va-t-on ?

## 1.1 Ce qu'on a couvert jusqu'à présent

— Rudiments de l'architecture physique d'un ordinateur - le modèle de Von Neumann.
— Mise en jambes sur de l'écriture de code : réalisation d'une page Web en HTML.
— Et surtout : introduction à Python, et plus spécifiquement :
  — Ce qu'on appelle ses "constructions élémentaires" – variables, fonctions, conditions & embranchements, boucles...
  — Les types et valeurs de base : entiers (naturels et relatifs), flottants (réels), chaînes de caractères et booléens (qu'on n'a que brièvement abordés pour l'instant).
  — Un type dit "construit" – les listes.

## 1.2 Ce dont on va parler dans ce nouveau chapitre

On va prendre un pas de recul par rapport au chapitre sur Python – et se demander comment les données qu'on manipule sont stockés dans l'ordinateur. Après tout, on sait que celui-ci ne manipule que des 0 et des 1 ; donc comment peut-il stocker `12` ou $\pi$ ou `"Bonjour!!"` ?

C'est à ces questions qu'on va répondre en étudiant comment différents types de données sont ***représentés*** dans la mémoire de l'ordinateur

Spécifiquement, on va aborder les points suivants :

— Représentation des entiers naturels ;
— Représentation des entiers relatifs ;
— Représentation des nombres réels ("nombres flottants") ;
— Représentation et codage du texte ;
— Représentation des booléens – opérateurs booléens, logique boolénne – ce qui nous amènera à :
— Circuits logiques élémentaires.

## 1.3 Comment on va procéder

On passe ici dans un mode beaucoup plus théorique :

— On ne cherche plus à *faire faire* quelque chose à l'ordinateur (comme en programmation) ;
— On cherche plutôt à comprendre *comment l'ordinateur lui-même fonctionne.*
— On va s'intéresser à des aspects beaucoup plus mathématiques de l'informatique.

On va donc devoir travailler différemment du chapitre sur l'introduction à Python :

— **Prise de notes essentielle** ;
— Plusieurs exercices sur papier qu'on fera en classe et dont il sera très important que vous gardiez une trace ;

— **Attention :** régulièrement je vous demanderai de terminer à la maison les exercices commencés en classe – il faudra donc en avoir pris suffisamment de notes !
— Quand même, quelques applications / exercices sur machine.

# 2 Bases de numération & Représentation des entiers naturels

## 2.1 La base 10

Quelques questions "simples" pour commencer...

→*Que veut dire "2754"?*

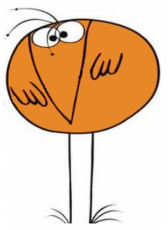
→*Autrement dit : comment fonctionne la base 10?*

→*Pourquoi la base 10 s'appelle-t-elle la base 10?*

→*Pourquoi utilise-t-on la base 10?*

→*Comment s'appellent les symboles utilisés pour écrire des nombres en base 10?*

## 2.2 Et en base ~~Shadok~~ 4, par exemple ?



Les Shadoks, c'est un dessin animé absurde français des années 60 – l'histoire d'oiseaux aux ailes ridiculement petites habitant sur une autre planète qui veulent conquérir l'univers ; pour ça ils ont besoin de la science. Donc de pouvoir compter. Leur problème ? Leur langue ne contient en tout et pour tout que quatre mots : `GA`, `BU`, `ZO`, et `MEU`....

Alors comment faire.... ?

**Allons voir la solution qu'ils ont retenue...**

Récapitulons :

| Mot | Symbole | Quantité |
|---|---|---|
| GA |  | - |
| BU |  | I |
| ZO |  | I I |
| MEU |  | I I I |

**Exercice 1: conversions d'une base à une autre**

Convertissez en base 10 les quantités exprimées en base Shadok suivantes :

a. Zo
b. Bu Zo
c. Meu Meu
d. Zo Ga
e. Meu Zo Ga

*Souvenez-vous : on compte, de droite à gauche, des Shadoks, puis des "poubelles"*  *de quatre Shadoks, des "grandes poubelles"*  *qui contiennent chacune quatre poubelles, etc...*

**Exercice 2: et maintenant sauvons le pauvre Professeur Shadoko...**

Convertissez le nombre mystère 

en français.

**Exercice 3: plus généralement...**

Proposez une méthode pour convertir un nombre Shadok dans notre système d'écriture courant.

Bon – abandonnons le vocabulaire Shadok à présent et revenons au cours : on parle évidemment ici d'une **base 4** et notre méthode générale peut s'écrire sous la forme d'une suite de puissances de 4.

Pour un nombre N noté en base 4 $a_4a_3a_2a_1a_0$, par exemple, on le convertira par la formule :

$$N = a_0 \cdot 4^0 + a_1 \cdot 4^1 + a_2 \cdot 4^2 + a_3 \cdot 4^3 + a_4 \cdot 4^4$$

Ou encore :

$$N = a_0 \cdot 1 + a_1 \cdot 4 + a_2 \cdot 16 + a_3 \cdot 64 + a_4 \cdot 256$$

Et plus généralement, pour un nombre N noté en base 4 $a_k a_{k-1}(...)a_2 a_1 a_0$, on aura :

$$N = \underbrace{a_0 \cdot 4^0 + a_1 \cdot 4^1 + a_2 \cdot 4^2 + (...) + a_{k-1} \cdot 4^{k-1} + a_k \cdot 4^k}_{\text{(k+1) éléments, de 0 à k}}$$

ce qui s'écrit aussi :

$$N = \sum_{i=0}^{k} a_i \cdot 4^i$$

Donc si on remplace Ga, Bu, Zo, et Meu par les chiffres arabes correspondants (0, 1, 2, 3), on pourra dire que le nombre 132 en base 4 vaut 46 en base 10 :

$$2 \cdot 4^0 + 3 \cdot 4^1 + 1 \cdot 4^2 = 2 \cdot 1 + 3 \cdot 4 + 2 \cdot 16 = 46$$

Pour exprimer cela simplement, on va placer la base d'expression du nombre en indice de celui-ci et écrire : $132_4 = 46_{10}$

**Exercice 4: et dans l'autre sens ?**

Convertissez en base 4 (ou en Shadok, comme vous préférez) les nombres écrits en base 10 suivants :

a. $16_{10} =$
b. $17_{10} =$
c. $64_{10} =$
d. $65_{10} =$
e. $11_{10} =$
f. $25_{10} =$
g. $57_{10} =$
h. $675_{10} =$

**Exercice 5: ça commence à devenir un poil compliqué...**

Proposez cette fois une méthode pour convertir un nombre de base 10 en base 4 (ou Shadok).

## 2.3 La base 2 – ou "numération binaire"

### Quel rapport avec l'informatique ?

Imaginons un interrupteur électrique : il peut être soit en position *ON*, soit en position *OFF*. En informatique, le "coeur" d'un ordinateur est composé de millions voire de milliards de petits "interrupteurs" appelés ***transistors***.

Un transistor permet à un courant électrique de passer (état "allumé" – que l'on va appeler "1") ou non (état "éteint" ou "0"). C'est cette simplicité qui rend les transistors fiables et efficaces pour construire des circuits électroniques complexes comme ceux que l'on trouve dans les processeurs d'ordinateur. Ainsi, la mémoire de l'ordinateur utilise des transistors pour stocker des informations sous forme de 1 et de 0 selon l'état des transistors qui la composent .

Toute information, de quelque nature qu'elle soit, doit être convertie en 1 et en 0 pour être manipulée par un ordinateur.

Donc en base 2.

Donc en binaire.

### Conversions de et vers la base 2

C'est à présent que les Shadoks vont venir à notre secours : parce que le principe en base 2 est rigoureusement le même qu'en base 4 – sauf qu'au lieu d'utiliser 4 symboles (0, 1, 2, et 3), on n'en utilise que 2 (0 et 1) et qu'au lieu de manipuler des puissances de 4, on manipule des puissances de 2. Pour un nombre N noté en base 2 $a_k a_{k-1}(...)a_2 a_1 a_0$, on aura :

$$N = \underbrace{a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + (...) + a_{k-1} \cdot 2^{k-1} + a_k \cdot 2^k}_{\text{(k+1) éléments, de 0 à k}}$$

ce qui s'écrit aussi :

$$N = \sum_{i=0}^{k} a_i \cdot 2^i$$

**Conseil:**

Vous allez vite vous rendre compte que pour ce chapitre et pour la matière NSI en général on gagne beaucoup de temps en connaissant les premières puissances de 2 – je vous conseille donc vivement de faire l'effort de les retenir :

| | | | | | |
|---|---|---|---|---|---|
| $2^{10} = 1024$ | $2^9 = 512$ | $2^8 = 256$ | $2^7 = 128$ | $2^6 = 64$ | $2^5 = 32$ |
| $2^4 = 16$ | $2^3 = 8$ | $2^2 = 4$ | $2^1 = 2$ | $2^0 = 1$ | |

**Exercice 6: commençons par compter...**

Comptez de 1 en 1 en base 2 en partant de 0 et jusqu'à arriver au nombre qui, en base 10, est représenté par 17.
*Souvenez-vous : pour compter en base 4, on faisait Ga, Bu, Zo, Meu, Bu Ga, Bu Bu, Bu Zo... Soit 0, 1, 2, 3, 10, 11, 12... En base 2, c'est la même chose, sauf qu'on n'a que deux symboles, 0 et 1.*

**Exercice 7: et maintenant convertissons de base 2 en base 10**

En utilisant les techniques utilisées pour la base 4, convertissez les nombres binaires suivants en base 10 :

a. 11

b. 10110

c. 10101100

Passons maintenant à la conversion réciproque – de base 10 vers base 2. Il existe en fait deux méthodes possibles :

— Celle qu'on a vue en classe pour la base 4, qui s'appelle pour la base 2 "**méthode des soustractions successives**" (*parce qu'en base 2 on n'a pas besoin de diviser !*) :

1. On commence par dresser un tableau avec les premières puissances de 2.
2. On soustrait la plus grande puissance de 2 qui est inférieure au nombre à convertir.
3. On note un "1" dans la case correspondante du tableau.
4. On répète ces étapes pour le nouveau nombre à convertir jusqu'à arriver à 0.

Exemple, pour convertir le nombre $25_{10}$ :

| 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 |

$25 - 16 = 9$ (on place un "1" dans la case 16)
$9 - 8 = 1$ (on place un "1" dans la case 8)
$1 - 1 = 0$ (on place un "1" dans la case 1)
Fin

Le nombre $25_{10}$ se convertit donc en $11001_2$.

— Celle des "**divisions successives**" qui consiste à diviser le nombre donné par 2 de manière répétée, et à noter le reste à chaque étape. Le processus continue en utilisant le quotient obtenu comme nouveau nombre à diviser, jusqu'à ce que le quotient soit égal à 0.

Exemple : convertissons de nouveau, mais avec cette nouvelle méthode, $25_{10}$ en base 2.

$25/2 = 12$ reste 1
$12/2 = 6$ reste 0
$6/2 = 3$ reste 0
$3/2 = 1$ reste 1
$1/2 = 0$ reste 1

En lisant les restes à l'envers, on obtient $11001_2$.

**Exercice 8: mettons ces deux méthodes en pratique**

En utilisant les deux méthodes, convertissez en base 2 :

a. 13

b. 22

c. 128

**Astuce:**

Il est essentiel de savoir effectuer ces conversions "à la main" comme on vient de le faire – mais pour de grands nombres, ça prend du temps, et Python vous offre une solution beaucoup plus rapide :

```
# Convertir '10110 ' en base 10
nombre_decimal = int('10110 ', base =2)
print ( nombre_decimal ) # Output sera 22
```

Donc ca, c'est comment l'ordinateur stocke et manipule les nombres – c'est pratique pour lui, mais un peu complique pour nous : par exemple, la population de la France est d'environ 64,8 millions de personnes. En binaire ca nous donne : 11110111001100010100000000... Même celle de Massy, qui n'est que de 50.506 habitants, donne 1100010101001010. Vous trouvez ca lisible, vous ?

## 2.4 La base hexadécimale, ou base 16

Cette base permet un compromis entre le code binaire des machines et une base de numération pratique à utiliser pour les humains. Elle utilise 16 symboles, ou "chiffres" de base : les dix chiffres arabes et les lettres A, B, C, D, E, F. Compter de $0_{10}$ a $20_{10}$ dans différentes bases se fait de la manière suivante :

| Base 10 | Base 2 | Base 16 |
|---|---|---|
| 01 | 0001 | 1 |
| 02 | 0010 | 2 |
| 03 | 0011 | 3 |
| 04 | 0100 | 4 |
| 05 | 0101 | 5 |
| 06 | 0110 | 6 |
| 07 | 0111 | 7 |
| 08 | 1000 | 8 |
| 09 | 1001 | 9 |
| 10 | 1010 | A |
| 11 | 1011 | B |
| 12 | 1100 | C |
| 13 | 1101 | D |
| 14 | 1110 | E |
| 15 | 1111 | F |
| 16 | 10000 | 10 |
| 17 | 10001 | 11 |
| 18 | 10010 | 12 |
| 19 | 10011 | 13 |
| 20 | 10100 | 14 |

Quel sont les intérêts de cette base ?

— Compacité : contrairement à la base 2, on peut écrire de grosses quantités en utilisant peu de symboles – la population de la France, c'est 3DCC500 et celle de Massy C54A.
— La conversion de base 2 à base 16 est particulièrement simple, puisque $16 = 2^4$ – on va y revenir ci-dessous.

La conséquence de ces points est que, vous allez le constater, la base hexadécimale est omniprésente en informatique – ce qui explique pourquoi on l'étudie ici : on la retrouve dans l'adressage en mémoire, dans les adresses réseau, dans la cryptographie, dans le codage des couleurs en programmation web...

### Rapport a la base 2

Expliquons un peu plus l'intérêt qu'il y a à utiliser une base qui soit aussi une puissance de 2 : comme vous le voyez sur le tableau ci-dessus, les 16 premiers entiers correspondent exactement aux représentations sur 4 symboles : de 0000 (soit 0 en base 10) à 1111 (soit 15 en base 10). La conversion se fait donc de manière très aisée :

— De base 2 en base 16 : on découpe, en partant de la droite, le nombre en "tranches" de 4 symboles et on convertit chacune de ces tranches en son symbole hexadécimal correspondant.
— De base 16 en base 2 : on fait exactement l'inverse – on convertit un par un les symboles en leur équivalent binaire.

**Exercice 9: pratiquons ca un peu...**

Convertissez en base 2 ou 16 les quantités suivantes (utilisez le tableau ci-dessus !) :

a. $3E6_{16}$
b. $A420_{16}$
c. $110100111100_2$
d. $1101001111001_2$

### Conversion de & vers la base 10

De nouveau, les principes et méthodes sont exactement les mêmes que pour les bases 4 et 2 – sauf que l'on manipule cette fois 16 symboles et des puissances de 16.

**Exercice 10: puisqu'on connait la méthode allons-y directement !**

a. Convertissez $12AF_{16}$ en base 10 (en vous souvenant qu'on manipule ici des puissances de 16 : 16, 256, 4096...).
b. Convertissez $A42_{16}$ en base 10.
c. Convertissez $120_{10}$ en hexadécimal – le plus simple est d'utiliser le méthode des divisions successives, cette fois en divisant par 16 a chaque étape.
d. Convertissez $443_{10}$ en hexadécimal.

## 2.5 Conversions d'une base à une autre en Python

En Python, par défaut, le nombres affichés ou saisis le sont en base 10 ; mais Python offre différentes fonctions permettant de naviguer aisément entre les bases 2, 10, et 16. Voici quelques exemples saisis en console Python (le préfixe "> > >" étant l'invite de commande) :

```
>>> # Utilisation de la notation 0b pour
>>> # manipuler une sequence de bits
>>> 0b100
4
>>> 0b01001011
75

>>> # Dans le sens inverse, on peut utiliser la syntaxe vue plus haut,
>>> # ou bien la fonction bin()
>>> bin(4)
'0b100'
>>> bin(75)
'0b1001011'

>>> # Utilisation de la notation 0x pour manipuler
>>> # des nombres en hexadécimal
>>> 0x10
16
>>> 0xA0
160
>>> 0xABC
2748

>>> # Et enfin utilisation de la fonction hex()
>>> # pour convertir un décimal en hexadécimal
>>> hex(16)
'0x10'
>>> hex(160)
'0xa0'
>>> hex(2748)
'0xabc'
```

# 3 Mémoire & encodage des entiers naturels

## 3.1 Unités de mémoire

En informatique, **TOUT** repose sur des 0 et des 1 – ces chiffres binaires également appelés bit (qui vient de **BI**nary digi**T** en anglais). Alors justement, commençons par un peu de vocabulaire :

**Bit :** la base de la base, un transistor tout seul qui ne peut avoir que deux valeurs en tout et pour tout : 0 ou 1.

**Octet :** un groupement de 8 bits – soit $2 \times 4$ chiffres binaires que l'on peut donc très simplement écrire sous forme de deux symboles hexadécimaux. C'est pour ca que l'octet est l'unité de mémoire la plus universellement utilisée.

**Byte :** là, la langue anglaise nous tend un piège. Un byte (prononcé baille-te), c'est un octet ; ***PAS*** un bit.

**Kilooctet – ko (kb en anglais) :** $10^3 = 1.000$ octets.

**Megaoctet – Mo :** $10^6 = 1.000.000$ octets.

**Gigaoctet – Go :** $10^9 = 1.000.000.000$ octets.

**Teraoctet – To :** $10^12 = 1.000.000.000.000$ octets.

## 3.2 Les entiers naturels

### Encodage / Représentation

On rappelle que l'ensemble des entiers naturels, $\mathbb{N}$, est l'ensemble des nombres entiers positifs, donc compris entre 0 et $+\infty$. Ce sont donc des entiers **non signés**.

Si on veut représenter un tel nombre sur une plage définie de bits, on sera limités dans la plage de valeurs qu'on peut stocker. Par exemple, sur 1 bit on ne peut aller que de 0 a 1 ; sur 4 bits, de 0000 a 1111, soit en décimal de 0 a 15. C'est évidemment peu pratique – et donc en général, on code les entiers plutôt sur 4 *octets*, soit $4 \times 8 = 32$ bits.

**Exercice 11: entier maximal sur 1 ou 4 octets**

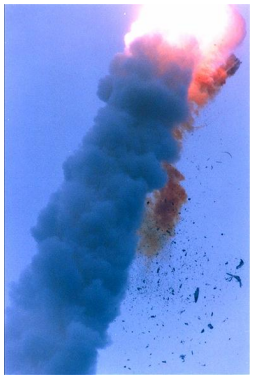
Quel est l'entier maximal que l'on peut coder sur 1 octet (8 bits) ? Et sur 4 octets (32 bits) ?

**Exercice 12: et à l'inverse...**

Combien de bits sont nécessaires pour coder l'entier 1 ? L'entier 7 ? L'entier 200 ?

Pourquoi est-ce important, ce genre de considération ? Demandez donc ça aux scientifiques qui travaillaient sur le premier vol de la fusée Ariane 5 à Kourou, en Guyane, le 4 juin 1996, et ont vécu ces 36 secondes tragiques... (Pour en savoir plus sur "la ligne de code la plus chère de l'Histoire" : page Wikipedia)


ARIANE 501 - 4 Juin 1996



# 4 Encodage des entiers relatifs

## 4.1 Utilisation d'un bit de signe

On sait donc maintenant coder en binaire des entiers positifs – mais comment faire pour des entiers négatifs (appartenant à l'ensemble des entiers relatifs, noté $\mathbb{Z}$) ?

Une première idée consisterait à utiliser ce qu'on appelle un "bit de signe" : décider que lorsqu'on utilise une représentation sur 8 bits pour coder un entier signé, le bit le plus à gauche est dédié au signe du nombre : un "0" indique un nombre positif et un "1" un nombre négatif. De ce fait, seuls les 7 bits restants sont utilisés pour représenter la valeur absolue de l'entier. Ainsi, dans ce système, la plus grande valeur positive que l'on peut représenter n'est pas "11111111" (qui serait interprété comme un nombre négatif à cause du bit de signe à "1"). Au lieu de cela, la plus grande valeur positive possible est "01111111", ce qui correspond à 127 en notation décimale.

*Vocabulaire : on appelle le bit le plus à gauche d'une séquence le "**bit de poids fort**" et celui le plus à droite le "**bit de poids faible**" – correspondant aux puissances de deux plus élevées à gauche qu'à droite.*

Donc, par exemple, 4 sera codé 00000100 et -4 10000100, et si on fait la somme des deux :

$$\begin{array}{r} 0\,0\,0\,0\,0\,1\,0\,0 \\ 1\,0\,0\,0\,0\,1\,0\,0 \\ \hline 1\,0\,0\,0\,1\,0\,0\,0 \end{array}$$

Ce qui, exprimé en décimal, donnerait : 4 + (-4) = -8 – ce qui est un peu un problème... Face à cette limitation, d'autres méthodes de représentation des nombres négatifs ont été développées pour permettre desopérations arithmétiques plus simples et plus précises.

## 4.2 Le complément à 1 – puis à 2

### Complément à 1

Le complément à 1 d'un nombre binaire est le nombre qui change tous les 0 en 1 et tous les 1 en 0. Si $n$ est un nombre binaire, le complément à 1 de $n$ est noté $\overline{n}$. Par exemple, le complément à 1 de 0011 est 1100.

Donc, par exemple, si fait la somme de 4 (00000100) et de $\overline{4}$, toujours sur 8 bits, on obtient :

$$\begin{array}{r} 0\,0\,0\,0\,0\,1\,0\,0 \\ 1\,1\,1\,1\,1\,0\,1\,1 \\ \hline 1\,1\,1\,1\,1\,1\,1\,1 \end{array}$$

Ce n'est évidemment pas ce qu'on cherche – mais on sent que c'est un début de piste...

**Exercice 13: faisons une petite pause Python...**

Ecrivez, dans IDLE, une fonction CompUn(Lst) qui prend en entrée une liste de bits (par exemple `[1,0,1,1,0,1]`) et retourne en sortie une liste contenant le complément à 1 (donc "inverse" tous les bits) de la liste en entrée (donc dans le cas précédent : `[0,1,0,0,1,0]`). Testez votre fonction sur les exemples suivants (la flèche représentant évidemment ce que doit retourner votre fonction) :

a. `CompUn([0])` →`[1]`

b. `CompUn([1])` →`[0]`

c. `CompUn([0, 1])` →`[1, 0]`

d. `CompUn([0,0,1,1,1])` →`[1,1,0,0,0]`

### Complément à 2

Ce que l'on va utiliser pour encoder les nombres relatifs est la méthode du **complément à 2** que l'on peut énoncer de la manière suivante :

- Le bit de poids fort est toujours utilisé pour représenter le signe ;
- La représentation des nombres positifs est inchangée ;
- Les nombres négatifs sont le complément à 2 de leur valeur positive :
  - On effectue le complément à 1 du nombre ;
  - On ajoute 1.

Exemples :

- Codage de -3 sur 4 bits :
  - $3_{10} = 011_2$ (on laisse le bit de poids fort de côté puisqu'il est utilisé pour le signe)
  - Complément à 1 : 100
  - Complément à 2 : 100 + 1 = 101
  - Donc le codage de -3 est, avec le bit de signe, est le "mot binaire" : 1101
- Codage de -4 :
  - $4_{10} = 100_2$ (on laisse le bit de poids fort de côté puisqu'il est utilisé pour le signe)
  - Complément à 1 : 011
  - Complément à 2 : 011 + 1 = 100
  - Donc le codage de -4 est, avec le bit de signe : 1100
- Codage de -1 :
  - $1_{10} = 001_2$
  - Complément à 1 : 110
  - Complément à 2 : 110 + 1 = 111
  - Donc le codage de -1 est, avec le bit de signe : 1111

Quel est l'intérêt ? Essayons de faire quelques sommes de chiffres opposés ***en ignorant la retenue finale*** :

$$\begin{array}{r} 0\,0\,1\,1 \\ 1\,1\,0\,1 \\ \hline 0\,0\,0\,0 \end{array}$$

$$\begin{array}{r} 0\,1\,0\,0 \\ 1\,1\,0\,0 \\ \hline 0\,0\,0\,0 \end{array}$$

**Exercice 14: additions d'entiers relatifs**

En utilisant le complément à deux et toujours sur 4 bits, effectuez les additions binaires suivantes

a. $-1_{10} + 1_{10}$

b. $2_{10} + -2_{10}$

c. $1_{10} + -4_{10}$

d. $-2_{10} + -4_{10}$

e. $2_{10} + 3_{10}$

On constate qu'avec cet encodage, pour tous mots binaires $m_1$ et $m_2$, on peut les additionner en effectuant simplement une addition binaire sans se soucier de leur signe et on obtiendra bien le résultat voulu si on ignore une éventuelle retenue finale.

En appliquant la méthode du complément à 2 encodons sur 4 bits le plus possible d'entiers relatifs – on obtient :

| **Mot Binaire** | | | | **Entier Relatif** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |

On constate qu'on a représenté ici, sur 4 bits, les entiers allant de -8 à +7 ; soit les entiers allant de $-(2^3)$ à $(2^3-1)$. On peut facilement se convaincre que cette formule est généralisable en disant que sur $k$ bits, on peut représenter des valeurs d'entiers relatifs allant de $-(2^{k-1})$ à $(2^{k-1}-1)$ — le complément à 2 de l'entier 0 servant à représenter l'entier $-(2^{k-1})$.

### Réciproque : passer d'une représentation binaire à l'entier relatif

Pour convertir un nombre binaire stocké sur un octet en complément à deux vers sa représentation en entier relatif, suivez les étapes suivantes :

1. Si le bit le plus à gauche (bit de signe) est 0, le nombre est positif – il suffit de convertir les 7 bits restants directement de base 2 à base 10.

2. Sinon, le nombre est négatif :
   (a) Inversez tous les bits.
   (b) Ajoutez 1 au résultat.
   (c) Convertissez les 7 bits restants en base 10 et ajoutez un signe négatif devant le résultat.

**Exercice 15: conversions en décimal**

Convertissez en décimal les nombres représentés en utilisant la méthode du complément à deux suivant

a. 00001000
b. 11111101
c. 11111000

# 5 Représentation approximative des nombres réels

**ATTENTION:**

Ce chapitre est assez complexe, notamment au niveau calculatoire. Il est important que vous en compreniez les concepts et les modes de calcul, mais **il ne vous sera pas demandé de connaître précisément la norme IEEE 754 – c'est hors programme**. C'est une des raisons pour lesquelles ce chapitre contient relativement moins d'exercices que les autres.

**Exercice 16: commençons par essayer de voir le problème**

Dans une console Python, effectuez les calculs suivants :

a. `0.5 - 0.2 - 0.2 - 0.1`

b. `9007199254740992.0 + 2.0`

c. `9007199254740992.0 + 1.0 + 1.0`

d. `1.0 + 1.0 + 9007199254740992.0`

La raison de ces erreurs est qu'il n'est pas possible de représenter la totalité des nombres réels (de l'ensemble mathématique $\mathbb{R}$) avec exactitude en binaire – ce qui en première analyse se comprend intuitivement très bien en constatant que le nombre de réels compris entre 0 et 1 est infini, tandis que le nombre de combinaisons de bits sur une taille mémoire donnée est nécessairement finie. On doit donc recourir à des approximations – et l'objet de ce chapitre est de comprendre comment ces approximations fonctionnent.

## 5.1 Puissances négatives de 2

Jusqu'à présent, nous avons travaillé avec des puissances successives de deux toujours positives – 1, 2, 4, 8, 16, 32, etc... – et en avons considéré la somme pour convertir des nombres de la base 2 à la base 10 – par exemple :

$$\begin{aligned} 10_{10} &= (8+2)_{10} \\ &= (1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0)_{10} \\ &= (1010)_2 \end{aligned}$$

Nous allons ici étendre cette notion aux puissances *négatives* de 2 pour représenter des nombres compris dans l'intervalle $[0, 1[$ :

| $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | ... |
|---|---|---|---|---|---|
| 0.5 | 0.25 | 0.125 | 0.0625 | 0.03125 | ... |

Ainsi, en utilisant cette notation, le mot binaire 1010 codé sur 4 bits aura pour valeur :

| $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 |

Que l'on va calculer ainsi :

$$\begin{aligned} 1010_2 &= (1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} + 0\times 2^{-4})_{10} \\ &= (0,5 + 0,125)_{10} \\ &= (0,625)_2 \end{aligned}$$

## 5.2 La notation scientifique

On utilise beaucoup, en mathématiques et en sciences, ce qu'on appelle la *notation scientifique* des nombres décimaux. Celle-ci se compose d'un signe (+ ou -) ; d'une *mantisse*, notée $m$ et comprise dans l'intervalle $[1,\ 10[$ ; et d'un *exposant*, nombre entier relatif noté $n$. L'écriture de cette valeur est alors $\pm m \times 10^n$. Ainsi, spécifiquement :

$$\begin{array}{rll} 2\ 156 & \text{s'écrit} & +2,156 \times 10^3 \\ -398\ 879,62 & \text{s'écrit} & -3,9887962 \times 10^5 \\ 0,000142 & \text{s'écrit} & +1,42 \times 10^{-4} \\ 1,34 & \text{s'écrit} & +1,34 \times 10^0 \end{array}$$

## 5.3 La norme IEEE 754

Cette norme, développée dans les années 1980 aux USA, existe en deux versions, l'une, dite "codage simple précision", pour une représentation sur 32 bits (4 octets), et l'autre, dite "codage double précision", pour une représentation sur 64 bits (8 octets). Dans les deux cas elle s'appuie sur le même principe que la notation scientifique – à s'avoir une décomposition du nombre en une somme de puissances, utilise les puissances négatives de 2 pour décrire le plus précisément possible la mantisse, et s'écrit sous la forme suivante :

$$(-1)^s \cdot (1+m) \times 2^{(n-d)}$$

- $s$ est le bit de signe, valant 0 ou 1 ;
- $m$ est la mantisse, comprise (puisqu'on raisonne ici non plus en base 10 mais en base 2) dans l'intervalle $[0,\ 1[$ ;
- $n$ est toujours l'exposant, mais il est cette fois "*décalé*" (on peut également dire "*biaisé*") d'une valeur d qui dépend du format choisi (32 ou 64 bits).

Ecrite de la sorte en mémoire et encodée sur 32 bits, la valeur d'un nombre flottant sera structurée comme suit :

$$\underbrace{\overbrace{\text{Signe}}^{(1\ bit)}}_{s}\ \underbrace{\overbrace{\text{Exposant}}^{(8\ bits)}}_{n}\ \underbrace{\overbrace{\text{Mantisse}}^{(23\ bits)}}_{m}$$

Voyons un exemple concret, le mot $M$ de 32 bits suivant :

$$11000011010101101100000000000000$$

Commençons par le décomposer en signe (bit de poids fort), exposant (8 bits suivants), et mantisse (23 bits restants) :

$$\overbrace{1}^{signe}\ \overbrace{10000110}^{exposant}\ \overbrace{10101101100000000000000}^{mantisse}$$

Calculons alors sa valeur décimale :

$$\begin{aligned}
\text{signe} &= -1^1 \\
&= -1 \\
\\
\text{exposant} &= 2^7 + 2^2 + 2^1 \\
&= 128 + 4 + 2 \\
&= 134 \\
\\
\text{mantisse} &= 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-8} + 2^{-9} \\
&= 0,677734375
\end{aligned}$$

Appliquons à présent la formule générale décrite plus haut ($(-1)^s \cdot (1+m) \times 2^{(n-d)}$) en utilisant un décalage $d$ de 127 puisque nous étudions un encodage sur 32 bits :

$$\begin{aligned}
M &= -1 \times (1 + 0,677734375) \times 2^{(134-127)} \\
&= -1,677734375 \times 2^7 \\
&= -214,75
\end{aligned}$$

**Exercice 17: Entrainons-nous...**

Convertissons quelques réels codés sur 32 bits pour s'entrainer...

a.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

b.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Exercice 18: C'est un peu fastidieux... et si on codait ça ?**

a. Sur papier, écrire en pseudo-code l'algorithme d'une fonction qui prend en entrée une liste de 32 éléments représentant les 32 bits encodant un nombre flottant et qui calcule puis renvoie la valeur décimal du réel correspondant.

b. Dans IDLE, écrire cette fonction en Python. Si vous avez terminé, rendez cette fonction robuste – qu'elle sache gérer des listes n'ayant pas le bon nombre d'éléments, ou ne contenant pas que des 0 ou des 1 (utilisez la fonction `assert` notamment)...

## 5.4 Alors d'où viennent les erreurs qu'on a vues ?

Le problème est qu'il n'est en général pas possible de coder *exactement* un nombre réel avec ce système – sauf si, par chance, il peut se décomposer exactement de cette manière. L'ordinateur en est donc réduit à stocker des approximations des valeurs qu'on lui donne.

Par exemple : le nombre flottant le plus proche de 1,2 est

0 – 01111111 – 00110011001100110011001

La valeur exacte de ce nombre, si on le développe comme on l'a fait plus tôt, est de 1.2000000476837158 – ce qui est évidemment une différence négligeable pour la représentation de valeurs à l'échelle humaine (qu'il s'agisse, de distances, de pourcentages dans des contextes divers, de montants d'argent, de mesures physiques telles

que le poids ou la vitesse d'un objet...). Cependant, il faut bien avoir conscience de l'existence de ces approximations.

Deux remarques :

— Il est évident ici que l'ordinateur utilise une méthode d'arrondis pour passer d'un mot binaire à sa valeur décimale et réciproquement. Cette méthode est hors programme.
— Si cependant c'est un sujet qui vous intéresse n'hésitez pas à consulter ce site pour voir plus en détails comment différentes valeurs réelles sont effectivement codées au moyen de la norme IEEE 754.

En tout état de cause, l'objet de ce chapitre est principalement de vous alerter sur deux points fondamentaux :

— Le fait **qu'il ne faut jamais** effectuer des tests d'égalité entre nombres flottants dans un programme (on l'a vu en début de chapitre, et on en avait vu également des exemples dans le cours sur Python) ;
— Le fait que dans des contextes où les besoins de précision sont hors du commun (dans certaines disciplines scientifiques par exemple), il sera nécessaire de trouver des moyens de contourner ces difficultés pour éviter de générer de réelles erreurs.

# 6 Représentation des caractères

**ATTENTION:**

Comme le chapitre précédent, celui-ci a pour vocation d'être une introduction au codage des caractères – vous devez en comprendre les principes et les mécanismes, mais **il ne vous sera pas demandé de connaître précisément le normes d'encodage (ASCII et Unicode) – c'est hors programme**.

## 6.1 Code ASCII

Les ordinateurs ne stockent que des 0 et des 1 ; on sait assez simplement établir une correspondance entre un nombre décimal et sa traduction en binaire. Partant de ces deux constats, on a initialement établi une correspondance unique entre chaque caractère et un entier naturel – on appelle cette démarche le *charset*.

Ceci a donné naissance en 1963 au code ASCII (*American Standard Code for Information Interchange*), toujours en usage, et permettant d'encoder 128 caractères : 33 caractères dits de contrôle (codes 0 à 32 et code 127 – ils incluent des caractères "blancs" comme les espaces ou les tabulations, des suppressions, des retours à la ligne, des marqueurs de début et de fin de texte...), 94 caractères standards (codes 33 à 126) comportant les 26 lettres minuscules, les 26 lettres majuscules, les 10 chiffres, et 32 symboles (ponctuation, symboles arithmétiques...).

Le tableau ci-dessous vous présente tout ceci exhaustivement (pour votre culture générale – il est évident **que ce n'est pas à connaitre par cœur !**). La colonne code contient deux valeurs - d'abord en décimal, puis en hexadécimal.

| Code | Car. | Code | Car. | Code | Car. | Code | Car. |
|---|---|---|---|---|---|---|---|
| 0, 0 | NUL | 32, 20 | Espace | 64, 40 | @ | 96, 60 | ‘ |
| 1, 1 | SOH | 33, 21 | ! | 65, 41 | A | 97, 61 | a |
| 2, 2 | STX | 34, 22 | " | 66, 42 | B | 98, 62 | b |
| 3, 3 | ETX | 35, 23 | # | 67, 43 | C | 99, 63 | c |
| 4, 4 | EOT | 36, 24 | $ | 68, 44 | D | 100, 64 | d |
| 5, 5 | ENQ | 37, 25 | % | 69, 45 | E | 101, 65 | e |
| 6, 6 | ACK | 38, 26 | & | 70, 46 | F | 102, 66 | f |
| 7, 7 | BEL | 39, 27 | ’ | 71, 47 | G | 103, 67 | g |
| 8, 8 | BS | 40, 28 | ( | 72, 48 | H | 104, 68 | h |
| 9, 9 | HT | 41, 29 | ) | 73, 49 | I | 105, 69 | i |
| 10, A | LF | 42, 2A | * | 74, 4A | J | 106, 6A | j |
| 11, B | VT | 43, 2B | + | 75, 4B | K | 107, 6B | k |
| 12, C | FF | 44, 2C | , | 76, 4C | L | 108, 6C | l |
| 13, D | CR | 45, 2D | - | 77, 4D | M | 109, 6D | m |
| 14, E | SO | 46, 2E | . | 78, 4E | N | 110, 6E | n |
| 15, F | SI | 47, 2F | / | 79, 4F | O | 111, 6F | o |
| 16, 10 | DLE | 48, 30 | 0 | 80, 50 | P | 112, 70 | p |
| 17, 11 | DC1 | 49, 31 | 1 | 81, 51 | Q | 113, 71 | q |
| 18, 12 | DC2 | 50, 32 | 2 | 82, 52 | R | 114, 72 | r |
| 19, 13 | DC3 | 51, 33 | 3 | 83, 53 | S | 115, 73 | s |
| 20, 14 | DC4 | 52, 34 | 4 | 84, 54 | T | 116, 74 | t |
| 21, 15 | NAK | 53, 35 | 5 | 85, 55 | U | 117, 75 | u |
| 22, 16 | SYN | 54, 36 | 6 | 86, 56 | V | 118, 76 | v |
| 23, 17 | ETB | 55, 37 | 7 | 87, 57 | W | 119, 77 | w |
| 24, 18 | CAN | 56, 38 | 8 | 88, 58 | X | 120, 78 | x |
| 25, 19 | EM | 57, 39 | 9 | 89, 59 | Y | 121, 79 | y |
| 26, 1A | SUB | 58, 3A | : | 90, 5A | Z | 122, 7A | z |
| 27, 1B | ESC | 59, 3B | ; | 91, 5B | [ | 123, 7B | { |
| 28, 1C | FS | 60, 3C | < | 92, 5C | \ | 124, 7C | \| |
| 29, 1D | GS | 61, 3D | = | 93, 5D | ] | 125, 7D | } |
| 30, 1E | RS | 62, 3E | > | 94, 5E | ^ | 126, 7E | ~ |
| 31, 1F | US | 63, 3F | ? | 95, 5F | _ | 127, 7F | DEL |

Un texte codé en ASCII est simplement une suite d'octets (on va voir pourquoi tout de suite) correspondant à la séquence de caractères et utilisant les valeurs du tableau ci-dessus. Par exemple la phrase `Bonjour le monde!` correspond à la séquence d'octets :

| B | o | n | j | o | u | r | | l | e | | m | o | n | d | e | ! |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 42 | 6F | 6E | 6A | 6F | 75 | 72 | 20 | 6C | 65 | 20 | 6D | 6F | 6E | 64 | 65 | 21 |

→*Ce tableau nous permet donc d'encoder des textes simples, mais il est loin d'être parfait. A première vue, qu'est-ce qui ne va pas? Quel(s) problème(s) va-t-on rapidement rencontrer?*?

→*Sur combien de bits va-t-on encoder ces caractères ASCII?*?

## 6.2 Normes ISO 8859

Partant du constat de ces faiblesses, on a progressivement amélioré les capacités et méthodes de codage de texte pour les enrichir et les rendre compatibles avec de plus en plus de langues et de caractères spéciaux:

— On a commencé par utiliser ce fameux $8^{\text{ème}}$ bit de l'octet pour *doubler* le nombre de caractères qu'on pouvait coder (norme ISO 8859) et arriver à 256, les 128 premiers demeurant les caractères ASCII. On a appelé cette nouvelle norme "ANSI".

— C'était bien mieux, mais encore insuffisant – pensez encore une fois aux Russes, Arabes, et Chinois pour ne citer qu'eux. On a donc multiplié ces codages en créant des *pages* (ou tables de correspondances) de codes pour différents contextes, numérotées et nommées "8859-$n$". Pour chaque page, on conserve les caractères ASCII sur les 128 premiers codes et on utilise les 128 restants pour coder des caractères spécifiques à une zone géographique. Ainsi (quelques exemples):

  — La page 8859-1 est spécifique aux caractères de l'Europe occidentale;

  — La page 8859-5 est spécifique aux caractères cyriliques (donc, entre autres, au Russe);

  — La page 8859-6 est spécifique à l'Arabe;

  — La page 8859-9 est spécifique au Turc et au Kurde;

  — etc...

Tout ceci constitue évidemment une amélioration (16 tables ont été développées en tout, avec donc une capacité d'encodage totale 17 fois supérieure à ce que permettait la seule norme ASCII) mais:

— Ca reste largement insuffisant: 10 de ces 16 pages sont consacrées aux seules langues latines!

— Le fonctionnement impose de spécifier avant tout texte quelle page d'encodage on utilise (puisqu'un même caractère peut être codé différemment suivant la norme utilisée) et de s'y tenir – ce qui peut entraîner beaucoup de confusion. Avez-vous par exemple déjà croisé un mail ou une page web ressemblant à ceci?

*Prenons lâ€™exemple typique de la lumiÃ¨re Ã@mise par un phare maritime : elle est dâ€™abord indivisible, son coÃ¿¿t de production Ã@tant alors indÃ@pendant du nombre dâ€™utilisateurs ; elle possÃ¨de une propriÃ@tÃ@ de non-rivalitÃ@ (elle ne se dÃ@truit pas dans lâ€™usage et peut donc Ãªtre adoptÃ@e par un nombre illimitÃ@ dâ€™utilisateurs) ; elle est Ã@galement non excluable car il est impossible dâ€™exclure de lâ€™usage un utilisateur, mÃªme si ce dernier ne contribue pas Ã son financement.*

## 6.3 La norme UTF-8

Dans un monde de plus en plus globalisé, le besoin d'un système de codage capable de gérer un vaste éventail de caractères est devenu impératif. C'est dans ce contexte que l'UTF-8 (8-bit Unicode Transformation Format) est devenu la norme dominante: elle s'appuie toujours, pour les premiers caractères, sur la norme ASCII mais permet d'utiliser jusqu'à 24 bits pour coder un caractère. Créé en 1992, UTF-8 peut représenter plus d'un million de caractères différents, ce qui le rend idéal pour les applications internationales, y compris les pages Web. En effet, si vous consultez le code source d'une page Web moderne, il est fort probable que "UTF-8" soit spécifié comme l'encodage de caractères utilisé.

Nous ne couvrirons pas ici les principes de codage en UTF-8 – il vous est suffisant de savoir d'où cette norme vient, dans quelle logique d'évolution elle s'inscrit. Si en revanche vous voulez en savoir plus à ce sujet, je vous invite à consulter cette section de la page Wikipedia consacrée au codage UTF-8.

## 6.4 Encore un détour par Python...

### Fonctionnement de base des chaînes de caractères

Faisons de nouveau un petit détour par Python, et plus spécifiquement:

- Quelques éléments de manipulation de chaines de caractères que nous n'avons pas encore vus jusqu'à présent;
- Quelques fonctions spécifiques que propose le langage pour aller d'un caractère à son code et réciproquement.

Jusqu'à présent on n'a pas vraiment manipulé de chaines de caractères – on a affiché des messages à l'écran qui en utilisaient ("`print('Le nombre est pair')`" par exemple), ou, par le biais de la fonction `input`, on a pris du texte saisi par l'utilisateur en entrée. Mais on ne s'est pas vraiment demandé *ce qu'était* une variable de type "str".

**Définition:**

Une **chaîne de caractères** est une collection ordonnée (ou séquence) de caractères délimitée par des apostrophes (') ou des guillemets (").

Le choix des délimitateurs est le plus souvent imposé par le contenu de la chaîne de caractères. Par exemple la chaîne 'Bonjour !' ne fonctionne très bien délimitée par des apostrophes, tandis que la chaîne "Aujourd'hui" contenant une apostrophe nécessite des guillemets comme délimitateur.

Une chaîne de caractères fonctionne à bien des égards comme une liste:

- `len(chaine)` renvoie la longueur (en nombre de caractères) de la chaîne `chaine`.
- On peut définir une chaîne de caractères de différentes manières:
  - En partant d'une chaîne vide:

    ```
    chaine1 = ''
    chaine2 = ""
    ```

  - Par extension, en indiquant directement les éléments entre délimitateurs:

    ```
    chaine3 = 'Bonjour !'
    ```

  - Par *transtypage*, en utilisant la fonction `str()`:

    ```
    chaine4 = str(123) # renverra la chaîne '123'
    ```

- Comme avec les listes on peut effectuer des opérations de **concaténation** et de **duplication** au moyen des opérateurs `+` et `*`. Si a ="Bonjour " et b = "Monsieur" alors:
  - L'opération `c = a + b` affectera à la variable c la valeur "Bonjour Monsieur".
  - L'opération `d = a * 3` affectera à la variable d la valeur "Bonjour Bonjour Bonjour ".

— Comme c'est le cas pour les listes, on peut effectuer divers tests sur les chaînes de caractères:
  — Test d'égalité: `"Abc" == 'Abc'` renverra `True` puisque les deux listes sont égales (attention en revanche au fait que les majuscules et les minuscules sont bien des caractères différents: `"Abc" == 'abc'` renverra `False`).
  — Test d'inégalité: `"abc" != "acb"` renverra `True` puisque les chaînes ne sont pas égales entre elles.
  — Test d'appartenance au moyen du mot clé `"in"`: `"jour" in "Bonjour !"` renverra donc `True`.
  — Et à l'inverse, test de "non appartenance" en utilisant l'opérateur logique `not` que l'on a déjà vu: `"nuit" not in "Bonjour !"` renverra `True`.
  — Test d'ordre, basé sur l'encodage des caractères - et notamment sur le code ASCII des lettres élémentaires non accentuées. Ainsi:
    — `"a" < "n"` renverra `True` (codes ASCII respectifs: 61 et 6E)
    — `"N" < "a"` renverra `True` égaleement (codes ASCII respectifs: 4E et 61)
— Les éléments (caractères) d'une chaîne sont ordonnés et indexés – le plus à gauche a l'index 0, puis on incrémente de un en un, comme pour une liste. On peut accéder à ces éléments avec la même syntaxe que pour une liste - et avec les mêmes limites. Si on définit `"chaine = 'Bonjour !'"` alors:
  — `chaine[0]` vaut `"B"`;
  — `chaine[3]` vaut `"j"`;
  — `chaine[8]` vaut `"!"` (puisque le point d'exclamation est précédé d'un espace);
  — Et `chaine[9]` renverra une erreur de type `IndexError: string index out of range`.
— Et enfin, comme pour une liste, on peut parcourir une chaîne de caractères à l'aide d'une boucle, soit par indice soit par caractère (ou élément):

```
1  chaine = "Bonjour !"
2
3  # Boucle 1: par indice
4  for i in range(len(chaine)):
5      print(chaine[i])
6
7  # Boucle 2: par caractère / élément
8  for car in chaine:
9      print(car)
```

Les deux boucles ci-dessus donneront le même affichage à l'écran:

```
B
o
n
j
o
u
r

!
```

**ATTENTION:**

A l'inverse d'une liste, le type "str" est *non modifiable* en Python. Ceci signifie en particulier qu'on ne peut pas modifier un caractère spécifique à l'intérieur d'une chaîne. Les commandes suivantes:

```
chaine = 'BonJour !'
chaine[3] = "j"
```

renverront une erreur de type `TypeError: 'str' object does not support item assignment`.

**Exercice 19: manipulons quelques chaînes de caractères...**

Rédigez d'abord en pseudo-code puis écrivez le code de:

a. Une fonction qui prend deux chaînes de caractères en entrée, et renvoie la concaténation des deux avec un espace ajouté au centre.

b. La même fonction qui cette fois vérifie qu'il n'y a qu'un seul espace entre les deux chaînes – qui donc n'en ajoutera un que si la première ne se termine pas par un espace et la seconde ne commence pas par un espace.

c. Une fonction qui prend une chaîne de caractères en entrée, et renvoie, si sa longueur est impaire, le caractère situé au centre (donc pour "ABCDE" elle renverrait "C"), et si elle est paire:
   - Version simple (corrigée ci-dessous): elle renvoie la chaîne de caractères "XX" à chaque fois (donc pour "ABCDEF" en entrée, elle renvoie "XX").
   - Version plus sophistiquée pour ceux qui veulent essayer: les deux caractères du milieu (pour "ABCDEF" en entrée, elle renvoie "CD").

d. Affectation d'élèves par groupe alphabétique: une fonction qui prend en entrée un nom (chaîne de caractères commençant par une majuscule) et retourne un nombre situé entre 1 et 3 – 1 si l'initiale est comprise entre A et H; 2 si elle est entre I et Q; et 3 si elle est entre R et Z.

e. Séparation des caractères par des virgules: une fonction qui prend en entrée une chaîne de caractères et renvoie la même chaîne avec des virgules insérées entre tous les caractères (par exemple l'appel à cette fonction avec "NSI" renverra "N,S,I").

### Fonctions avancées

Une chaîne de caractères n'étant pas modifiable, il est relativement difficile de la manipuler (par exemple, si on veut modifier certains de ses caractères). Pour contourner ce problème, une méthode consiste à transformer la chaîne en liste, à effectuer diverses manipulations sur celle-ci, puis à la retransformer en chaîne par concaténation.

La première de ces méthodes s'appuie sur la fonction de transtypage `list()` qui renvoie une liste dont les éléments correspondent aux caractères de la chaîne:

```
list("Bonjour !")
```

renverra

```
["B","o","n","j","o","u","r"," ","!"]
```

L'inconvénient de cette approche est évidemment que cela découpe toutes les chaînes

en caractères individuels - ce qui n'est pas nécessairement ce qu'on recherche. On peut vouloir par exemple isoler des mots (en repérant les espaces) ou des éléments d'une liste (en repérant des virgules ou point virgules). Pour cela on utilise la méthode `chaine.split(separateur)` dans laquelle on spécifie un séparateur:

```
"Thomas, Sirine, Aboubacar".split(", ")
"Thomas, Sirine, Aboubacar".split() # Pas de séparateur spécifié
```

renverront

```
['Thomas', 'Sirine', 'Aboubacar']
['Thomas,', 'Sirine,', 'Aboubacar'] # sep par défaut: espace
```

(Notez que le séparateur est éliminé de la chaîne par la méthode split).

La méthode split a sa réciproque, qui est la méthode join:

```
mots = ["Rouge","Vert","Bleu"]
" ".join(mots)
" et ".join(mots)
"".join(mots)
```

renverront

```
"Rouge Vert Bleu"
"Rouge et Vert et Bleu"
"RougeVertBleu"
```

**Exercice 20: quelques unes des nombreuses autres méthodes...**

Dans une console IDLE, testez avec différentes valeurs les commandes suivantes et déterminez ce que font ces différentes méthodes:

a. `chaine1.count(chaine2)`
b. `chaine1.find(chaine2)`
c. `chaine1.replace(chaine2, chaine3)`
d. `chaine1.capitalize()`
e. `chaine1.isupper()`
f. `chaine1.upper()`
g. `chaine1.islower()`
h. `chaine1.lower()`
i. `chaine1.swapcase()`

**Exercice 21: quelques méthodes permettant de coder / décoder des caractères...**

Dans une console IDLE, testez les commandes suivantes et déterminez ce qu'elles font:

a. `ord('a')`
b. `ord('Z')`
c. `chr(122)`
d. `chr(0x4D)`

**Exercice 22: un hack de mot de passe!**

Dans cet exercice, nous allons simuler un simple crack de mot de passe. Choisissez un mot de passe composé de quatre caractères (lettres majuscules, minuscules sans accents, ou chiffres ), par exemple "Abc1" ou "ZZZZ". L'objectif est de créer un programme Python qui va tenter de deviner ce mot de passe.

Il est évident que vous allez utiliser plusieurs des fonctions que l'on vient de découvrir... Je vous laisse deviner lesquelles. **Pensez bien à commencer par rédiger sur papier votre pseudo-code!!**.

Une fois le mot de passe trouvé, le programme doit afficher un message pour vous narguer.

# 7 Circuits & logique booléenne

## 7.1 Les variables booléennes

On les a déjà vues sans formellement les présenter – alors corrigeons cela:

**Définition:**

Les *valeurs booléennes* sont les valeurs 0 et 1 correspondant à l'état d'un bit: elles sont interprétées comme signifiant `False` (faux) et `True` (vrai) respectivement.

Une *variable booléenne* est donc une variable ne pouvant prendre que l'une ou l'autre de ces deux valeurs.

Dans ce cours on les a déjà vues à de multiples reprises quand on effectue des tests dans une console Python: `2 == 3` renvoie `False` et `3 in [1, 2, 3, 4]` renvoie `True`.

Le codage des variables booléennes est bien entendu évident – puisqu'elles ne peuvent prendre que deux valeurs (qui en plus sont 0 et 1) il va de soi qu'il suffit d'un bit pour les représenter en mémoire.

## 7.2 Les opérateurs booléens

Eux aussi on a déjà commencé à les voir sans formellement les présenter – et comme par ailleurs ils sont assez difficiles à définir (et qu'en plus c'est un peu futile d'essayer de le faire) le mieux est de directement présenter les principaux d'entre eux...

### L'opérateur "non"

Il transforme 1 en 0 et réciproquement:

```
non(1) = 0
non(0) = 1
```

Un opérateur booléen peut s'exprimer en "table de vérité" à deux ou plus colonnes – les colonnes de gauche représentent les entrées, la colonne de droite représente la "sortie", soit le résultat de l'application de l'opérateur. Les lignes, elles, doivent contenir l'ensemble des combinaisons d'entrées possibles – comme on va le voir, dans le cas d'un opérateur aussi simple que non(), il n'y en a que deux.

**Exercice 23: table de vérité de l'opérateur "non"**

Compléter la table de vérité suivante:

| $a$ | $non(a)$ |
|---|---|
| 0 | |
| 1 | |

En Python, l'opérateur "non" est `not`.

### L'opérateur "ou"

Soient $a$ et $b$ deux variables booléennes; l'opérateur `ou` appliqué à $a$ et $b$ renvoie 1 si l'une des deux variables au moins vaut 1, 0 sinon.

**Exercice 24: table de vérité de l'opérateur "ou"**

Compléter la table de vérité suivante:

| $a$ | $b$ | $a$ ou $b$ |
|---|---|---|
| 1 | 1 | |
| 1 | 0 | |
| 0 | 1 | |
| 0 | 0 | |

En Python, l'opérateur "ou" est `or`.

### L'opérateur "et"

Soient $a$ et $b$ deux variables booléennes; l'opérateur `et` appliqué à $a$ et $b$ renvoie 1 si les deux variables valent 1, 0 sinon.

**Exercice 25: table de vérité de l'opérateur "et"**

Compléter la table de vérité suivante:

| $a$ | $b$ | $a$ et $b$ |
|---|---|---|
| 1 | 1 | |
| 1 | 0 | |
| 0 | 1 | |
| 0 | 0 | |

En Python, l'opérateur "et" est `and`.

### L'opérateur "xor" (également appelé "ou exclusif")

Soient $a$ et $b$ deux variables booléennes; l'opérateur `xor` appliqué à $a$ et $b$ renvoie 1 si et seulement si exactement une des deux variables vaut 1, 0 sinon.

**Exercice 26: table de vérité de l'opérateur "xor"**

Compléter la table de vérité suivante:

| $a$ | $b$ | $a$ xor $b$ |
|---|---|---|
| 1 | 1 | |
| 1 | 0 | |
| 0 | 1 | |
| 0 | 0 | |

En Python, l'opérateur "xor" est `^`.

## 7.3 Expressions booléennes

**Définition:**

Une *expression booléenne* est une expression qui a pour valeur 0 ou 1; elle est construite à l'aide d'une combinaison de variables booléennes et d'opérateurs booléens.

Par exemple `(a et b) ou non(c)` est une expression booléenne – j'imagine que vous ne serez pas surpris que maintenant je vous demande...

**Exercice 27: table de vérité de l'expression "(a et b) ou non(c)"**

Pour se faciliter la tâche, lorsqu'une expression booléenne devient compliquée, on ajoute des colonnes intermédiaires représentant les résultats des sous expressions – et l'on combine ensuite ces résultats intermédiaires pour obtenir le résultat final.

Compléter la table de vérité suivante:

| $a$ | $b$ | $c$ | $a$ et $b$ | non($c$) | ($a$ et $b$) ou non($c$) |
|---|---|---|---|---|---|
| 1 | 1 | 1 | | | |
| 1 | 1 | 0 | | | |
| 1 | 0 | 1 | | | |
| 1 | 0 | 0 | | | |
| 0 | 1 | 1 | | | |
| 0 | 1 | 0 | | | |
| 0 | 0 | 1 | | | |
| 0 | 0 | 0 | | | |

**Exercice 28: vous avez compris le principe? Parfait!**

Vous n'aurez donc aucun mal à dresser les tables de vérité des expressions booléennes suivantes (où, bien entendu, `a`, `b`, et `c` sont des variables booléennes):

a. `f(a, b, c) = non(a et non(b)) ou c`

b. `g(a, b, c) = (a xor non(b)) et c`

c. `h(a, b, c) = non(a et b) et (b ou c)`

## 7.4 Et pourquoi on apprend tout ça, au fait?

Cette section n'est évidemment pas à apprendre en vue de contrôles; on verra, mais je ne pense même pas passer trop de temps dessus en cours. En revanche, il me semble que c'est clairement une question légitime que vous pourriez vous poser. Alors si c'est le cas et / ou que ça vous intéresse – continuez à lire!

Il y a plusieurs raisons pour s'intéresser à la logique booléenne, pour en acquérir tôt les fondamentaux comme vous le faites actuellement. Le problème est que ces raisons semblent lointaines et abstraites quand on est en 1ère – et on a un peu l'impression qu'on est en train de nous obliger à apprendre des trucs qui ne servent à rien.

Je ne pense pas pouvoir vous passionner ici en quelques points – mais si vous réfléchissez un peu aux raisons que je vous propose ci-dessous, peut-être pourrez-vous vous persuader de l'intérêt du sujet à long terme, pour vous.

— Bases de l'informatique: les ordinateurs fonctionnent grâce à des circuits logiques qui traitent les informations en utilisant des opérations booléennes. Comprendre cette logique est essentiel pour saisir comment fonctionnent les ordinateurs au niveau le plus élémentaire.

— Fondements du codage et des algorithmes: la plupart des langages de programmation et algorithmes utilisent la logique booléenne pour la prise de décisions

(structures conditionnelles comme if/else) et les boucles (while, for). Une compréhension solide de la logique booléenne est donc essentielle pour écrire et comprendre des codes et des algorithmes efficaces.

— Plus largement – résolution de problèmes et pensée critique: apprendre la logique booléenne vous aide à décomposer les problèmes complexes en parties plus petites et plus gérables, une compétence précieuse dans de nombreux domaines, pas seulement en informatique. C'est ce qu'on appelle une approche cartésienne – quelque chose que vous apprendrez en cours de philosophie l'an prochain je crois.

— Plus généralement encore: la logique booléenne n'est pas uniquement utilisée en informatique. Elle trouve des applications dans des domaines tels que les mathématiques, l'électronique, la philosophie, et même dans la vie quotidienne pour la prise de décision logique. Elle est une manière de raisonner qui vous sera utile dans vos études supérieures, quelles qu'elles soient.