

1^{ère} 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 ?	3
1.1	Ce qu'on a couvert jusqu'à présent	3
1.2	Ce dont on va parler dans ce nouveau chapitre	3
1.3	Comment on va procéder	3
2	Bases de numération & Représentation des entiers naturels	5
2.1	La base 10	5
2.2	Et en base Shadok 4, par exemple ?	6
2.3	La base 2 – ou "numération binaire"	10
2.4	La base hexadécimale, ou base 16	14
2.5	Conversions d'une base à une autre en Python	16
3	Mémoire & encodage des entiers naturels	18
3.1	Unités de mémoire	18
3.2	Les entiers naturels	18
4	Encodage des entiers relatifs	21
4.1	Utilisation d'un bit de signe	21
4.2	Le complément à 1 – puis à 2	21
5	Représentation approximative des nombres réels	27
5.1	Puissances négatives de 2	27
5.2	La notation scientifique	28
5.3	La norme IEEE 754	28
5.4	Alors d'où viennent les erreurs qu'on a vues ?	32
6	Représentation des caractères	34
6.1	Code ASCII	34
6.2	Normes ISO 8859	36
6.3	La norme UTF-8	37
6.4	Encore un détour par Python...	37
7	Circuits & logique booléenne	46
7.1	Les variables booléennes	46
7.2	Les opérateurs booléens	46
7.3	Expressions booléennes	48
7.4	Et pourquoi on apprend tout ça, au fait ?	50

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 π 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 booléenne – 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"?

Implicitement, c'est une somme de puissances de 10 :

$$2 \cdot 10^3 + 7 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$$

→ Autrement dit : comment fonctionne la base 10?

Plus généralement, c'est donc une somme de puissances de 10 lue de droite à gauche (puisque si l'on ne regarde que le premier chiffre d'un nombre on ne peut pas savoir à quelle puissance il correspond). Cela revient à dire qu'un nombre ABCDEF écrit en base 10 se lit :

$$F \cdot 10^0 + E \cdot 10^1 + D \cdot 10^2 + C \cdot 10^3 + B \cdot 10^4 + A \cdot 10^5$$

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

Tout simplement parce qu'elle permet d'écrire la totalité de tous les entiers naturels en n'utilisant que 10 symboles – les 10 chiffres arabes : 1, 2, 3, 4, 5, 6, 7, 8, 9 – et le plus important de tous : 0.

→ Pourquoi utilise-t-on la base 10?

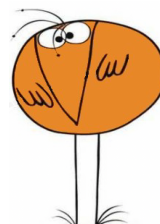
Deux remarques pour commencer : d'abord, on n'utilise pas que la base 10 – pensez par exemple aux oeufs qu'on compte encore aujourd'hui par douzaines ; ensuite, il n'y a pas de réponse certaine à cette question. La plus intuitive est que cela correspond à nos doigts – on en a 10, donc quand on les a tous "utilisés" on repart du premier, comme on fait avec les chiffres à l'écrit.

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

Ca a été dit plus haut – ce sont les chiffres, inventés en Inde, puis parvenus en Europe par le monde arabe (d'où leur nom) aux environs du X^e siècle.

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 :

- Zo
- Bu Zo
- Meu Meu
- Zo Ga
- Meu Zo Ga

Souvenez-vous : on compte, de droite à gauche, des Shadoks, puis des "poubelles"



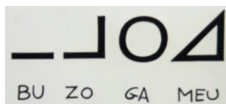
de quatre Shadoks, des "grandes poubelles" qui contiennent chacune quatre poubelles, etc...

Prenons d'entrée de jeu ici l'habitude d'aborder les nombres de droite à gauche – donc en commençant par les Shadoks, puis les poubelles, puis les grandes poubelles, etc...

- Zo = 2 – c'est écrit dans le tableau.*
- Bu Zo – on a zo, donc deux, Shadoks (chiffre de droite); et on a bu, donc une, poubelle qui contient 4 Shadoks. Donc Bu Zo = $2 \times 1 + 1 \times 4 = 6$.*
- Meu Meu – même raisonnement : meu Shadoks, donc 3, et meu poubelles, donc 3×4 ; donc Meu Meu = $3 \times 1 + 3 \times 4 = 15$.*
- Zo Ga – même chose, sauf que cette fois il y a ga, donc zéro, Shadoks à droite : Zo Ga = $0 \times 1 + 2 \times 4$*
- Meu Zo Ga – toujours la même démarche, sauf qu'on a cette fois un chiffre de plus, donc des grandes poubelles contenant chacune 4 poubelles de 4 :*

$$0 \times 1 + 2 \times 4 + 3 \times 16 = 56$$

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



Convertissez le nombre mystère en français.

Même chose que dans l'exercice précédent, sauf qu'on introduit ici un quatrième chiffre – les "super poubelles" qui contiennent 4 grandes poubelles, qui contiennent 4 poubelles, qui contiennent 4 Shadoks – donc $4 \times 4 \times 4$; il serait peut-être temps de parler en puissances de 4, non ?

$$\begin{aligned} & \text{Meu} \times 4^0 + \text{Ga} \times 4^1 + \text{Zo} \times 4^2 + \text{Bu} \times 4^3 \\ &= 3 \times 1 + 0 \times 4 + 2 \times 16 + 1 \times 64 = 99 \end{aligned}$$

Exercice 3: plus généralement...

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

On ne propose ici qu'une généralisation de ce qu'on a fait jusqu'à présent : partir du chiffre de droite, le multiplier par 4^0 , soit 1 ; passer au chiffre suivant, le multiplier par 4^1 , soit 4, et l'ajouter au précédent ; continuer ainsi vers la gauche en augmentant à chaque fois les puissances de 4 et en ajoutant au précédent jusqu'à atteindre le chiffre le plus à gauche.

En pseudo-code, on pourrait écrire :

Entrée: $\text{nbB4} \in \mathbb{N}$, exprimé en base 4

Sortie: $\text{resB10} \in \mathbb{N}$, exprimé en base 10

1: **fonction** CONVERTIRBASE4ENBASE10(nbB4)

2: $\text{resB10} \leftarrow 0$

3: $\text{puiss} \leftarrow 0$

4: **pour tout** chiffre de nbB4 **faire**

 ▷ Parcours des chiffres de droite à gauche

5: $\text{resB10} \leftarrow \text{resB10} + \text{chiffre} \times 4^{\text{puiss}}$

6: $\text{puiss} \leftarrow \text{puiss} + 1$

7: **fin pour**

8: **retourner** resB10

9: **fin fonction**

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 $a_k a_{k-1} \dots a_2 a_1 a_0$, on aura :

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

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. $675_{10} =$

On reprend à partir de maintenant les chiffres arabes pour noter nos nombres – ceux de 0 à 9 quand on est en base 10, et ceux de 0 à 3 en base 4 (qui remplacent donc Ga à Meu en base Shadok).

*On a remarqué en classe que les 4 premiers exemples de cet exercice sont des valeurs **très** proches de puissances de 4 – et ce n'est évidemment pas un hasard :*

$$\begin{aligned} 16_{10} &= (4^2)_{10} \\ &= (1 \times 4^2 + 0 \times 4^1 + 0 \times 4^0)_{10} \\ &= (100)_4 \end{aligned}$$

$$\begin{aligned} 17_{10} &= (16 + 1)_{10} \\ &= (4^2 + 1)_{10} \\ &= (1 \times 4^2 + 0 \times 4^1 + 1 \times 4^0)_{10} \\ &= (101)_4 \end{aligned}$$

Et de la même manière, on obtient pour les deux suivants :

$$64_{10} = (4^3)_{10} = 1000_4$$

$$65_{10} = (4^3 + 1)_{10} = 1001_4$$

Pour les questions e) à h), on applique rigoureusement la méthode que l'on va decrirer à l'exercice suivant et on obtient :

e. $11_{10} = 23_4$

f. $25_{10} = 121_4$

g. $57_{10} = 321_4$

h. $675_{10} = 22203_4$

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).

Pour convertir le nombre N_{10} en base 4, on va procéder de la manière qui suit :

1. On cherche d'abord la première puissance k de 4 qui soit strictement supérieure à N_{10}
2. On dresse un tableau avec les premières puissances de 4, de 4^{k-1} à 4^0 .
3. On effectue une division euclidienne de N_{10} par 4^{k-1} et on note le quotient de cette division dans la case 4^{k-1} du tableau.
4. On soustrait 4^{k-1} de N_{10} la plus grande puissance de 2 qui est inférieure au nombre à convertir.
5. On prend le reste de cette division et on répète ces étapes jusqu'à arriver à un quotient inférieur à 4.

Exemple, pour convertir le nombre 710_{10} :

$$256 \leq 710 < 1024$$

$$\text{Donc : } 4^4 \leq 710 < 4^5$$

Donc on dresse le tableau des puissances de 4 de l'exposant 4 à l'exposant 0 :

4^4	4^3	4^2	4^1	4^0
256	64	16	4	1
2	3	0	1	2

$$710/256 = 2 \quad (\text{reste} = 198) \quad (\text{on place un "2" dans la case 256})$$

$$198/64 = 3 \quad (\text{reste} = 6) \quad (\text{on place un "3" dans la case 64})$$

$$6/4 = 1 \quad (\text{reste} = 2) \quad (\text{on place un "1" dans la case 4})$$

$$2 < 4 \quad (\text{on place un "2" dans la case 1})$$

Fin

Le nombre 710_{10} se convertit donc en 23012_4 .

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 "cœur" 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 $a_k a_{k-1} \dots a_2 a_1 a_0$, on aura :

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

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.

0; 1; 10; 11; 100; 101; 110; 111; 1000; 1001; 1010; 1011; 1100; 1101; 1110; 1111; 10000; 10001

Vous saviez que l'humanité se divisait en 10 parties ? Ceux qui savent compter en binaire et ceux qui ne savent pas...

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

On va faire ici une somme des puissances de 2 successives, comme on l'avait fait avec les puissances de 4 plus haut dans le cours. Je vous soumetts ici les réponses en détaillant les calculs uniquement pour le dernier exercice :

- a. $11_2 = 3_{10}$
- b. $10110_2 = 22_{10}$

Convertissons à présent 10101100_2 en parcourant ses chiffres de droite à gauche en incrémentant à chaque étape la puissance appliquée à 2, et en incrémentant à chaque fois le résultat (que nous initialisons à 0) :

- $resultat + = 0 \times 2^0 = 0$
- $resultat + = 0 \times 2^1 = 0$
- $resultat + = 1 \times 2^2 = 4$
- $resultat + = 1 \times 2^3 = 12$
- $resultat + = 0 \times 2^4 = 0$
- $resultat + = 1 \times 2^5 = 44$
- $resultat + = 0 \times 2^6 = 0$
- $resultat + = 1 \times 2^7 = 172$

Donc $10101100_2 = 172_{10}$

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

$$25 - 16 = 9 \quad (\text{on place un "1" dans la case 16})$$

$$9 - 8 = 1 \quad (\text{on place un "1" dans la case 8})$$

$$1 - 1 = 0 \quad (\text{on place un "1" dans la case 1})$$

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.

$$25/2 = 12 \quad \text{reste } 1$$

$$12/2 = 6 \quad \text{reste } 0$$

$$6/2 = 3 \quad \text{reste } 0$$

$$3/2 = 1 \quad \text{reste } 1$$

$$1/2 = 0 \quad \text{reste } 1$$

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 :

- 13
- 22
- 128

1^{ère} méthode (soustractions successives) :

a. 13 :

16	8	4	2	1
0	1	1	0	1

$$13 - 8 = 5 \quad (\text{on place un "1" dans la case 8})$$

$$5 - 4 = 1 \quad (\text{on place un "1" dans la case 4})$$

$$1 - 1 = 0 \quad (\text{on place un "1" dans la case 1})$$

Fin

Le nombre 13_{10} se convertit donc en 1101_2 .

b. 22 :

32	16	8	4	2	1
0	1	0	1	1	0

$22 - 16 = 6$ (on place un "1" dans la case 16)
 $6 - 4 = 2$ (on place un "1" dans la case 4)
 $2 - 2 = 0$ (on place un "1" dans la case 2)
 Fin

Le nombre 22_{10} se convertit donc en 10110_2 .

c. 128 :

256	128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	0	0

$128 - 128 = 0$ (on place un "1" dans la case 128)
 Fin

Le nombre 128_{10} se convertit donc en 1000000_2 .

2nde méthode (divisions successives) :

a. 13 :

$13/2 = 6$ reste 1
 $6/2 = 3$ reste 0
 $3/2 = 1$ reste 1
 $1/2 = 0$ reste 1

En lisant les restes à l'envers, on obtient bien 1101_2 .

b. 22 :

$22/2 = 11$ reste 0
 $11/2 = 5$ reste 1
 $5/2 = 2$ reste 1
 $2/2 = 1$ reste 0
 $1/2 = 0$ reste 1

En lisant les restes à l'envers, on obtient bien 10110_2 .

c. 128 :

$128/2 = 64$ reste 0
 $64/2 = 32$ reste 0
 $32/2 = 16$ reste 0
 $16/2 = 8$ reste 0
 $8/2 = 4$ reste 0
 $4/2 = 2$ reste 0
 $2/2 = 1$ reste 0
 $1/2 = 0$ reste 1

En lisant les restes à l'envers, on obtient bien 1000000_2 .



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 :

```

1 # Convertir '10110 ' en base 10
2 nombre_decimal = int('10110 ', base =2)
3 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 compliqué 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} à 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 à 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 ça 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

- a. $3E6_{16}$: on s'appuie sur le tableau précédent et on traduit symbole par symbole en binaire : $3_{16} = 0011_2$; $E_{16} = 1110_2$; $6_{16} = 0110_2$. Donc en éliminant les 0 sur la gauche cela donne : $3E6_{16} = 1111100110_2$
- b. $A420_{16} = 1010010000100000_2$
- c. 110100111100_2 : toujours en s'appuyant sur le tableau précédent, on effectue l'opération inverse en découpant le nombre binaire en "tranches" de 4 chiffres, en partant de la droite :

$$\underbrace{1101}_{13} \underbrace{0011}_{3} \underbrace{1100}_{4}$$

Puis on convertit un à un ces nombres pour obtenir : $D3C_{16}$

- d. 1101001111001_2 – même méthode :

$$\underbrace{1}_{1} \underbrace{1010}_{10} \underbrace{0111}_{7} \underbrace{1001}_{5}$$

Et on obtient : $1A79_{16}$

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 connaît 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 la méthode des divisions successives, cette fois en divisant par 16 à chaque étape.
- d. Convertissez 443_{10} en hexadécimal.

$$a. 12AF_{16} = (15 \times 16^0 + 10 \times 16^1 + 2 \times 16^2 + 1 \times 16^3)_{10} = 4783_{10}$$

$$b. A42_{16} = (2 \times 16^0 + 4 \times 16^1 + 10 \times 16^2)_{10} = 2626_{10}$$

$$c. 120_{10} :$$

$$120/16 = 7 \quad \text{reste } 8$$

$$7/16 = 0 \quad \text{reste } 7$$

En lisant les restes à l'envers, on obtient 78_{16} .

$$d. 443_{10} :$$

$$443/16 = 27 \quad \text{reste } 11$$

$$27/16 = 1 \quad \text{reste } 11$$

$$1/16 = 0 \quad \text{reste } 1$$

En lisant les restes à l'envers (et en convertissant le nombre décimal 11 en son équivalent hexadécimal soit le chiffre B), on obtient $1BB_{16}$.

Exercice 11: à faire à la maison pour le prochain cours

a. $ABCD_{16}$ en base 10 ;

b. 1896_{16} en base 10 ;

c. 4097_{10} en hexadécimal ;

d. 2023_{10} en hexadécimal.

On applique les mêmes méthodes que précédemment et on obtient :

$$a. ABCD_{16} = 43981_{10}$$

$$b. 1896_{16} = 6294_{10}$$

$$c. 4097_{10} = 1001_{16}$$

$$d. 2023_{10} = 7E7_{16}$$

2.5 Conversions d'une base à une autre en Python

En Python, par défaut, les 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 2×4 chiffres binaires que l'on peut donc très simplement écrire sous forme de deux symboles hexadécimaux. C'est pour ça 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.

Kiloctet – 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 à 1 ; sur 4 bits, de 0000 à 1111, soit en décimal de 0 à 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 12: 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) ?

Sur 8 bits, le nombre maximal que l'on peut coder est $(11111111)_2$ soit, en décimal : $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255$. Cette somme est assez fastidieuse à faire – une manière plus rapide consiste à observer que, en binaire : $11111111 + 1 = 100000000$ (pour vous en convaincre, allez voir la section suivante !), et que $100000000_2 = (2^8)_{10} = 256_{10}$

Pour 32 bits, utilisons directement la manière rapide ci-dessus : $2^{32} = 4294967296$; donc le nombre maximal que l'on peut coder sur 4 octets est 4294967295.

Exercice 13: et à l'inverse...

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

- L'entier 1 se code sur 1 bit ;
- L'entier 7 sur 3 bits (puisque 3 bits, comme on l'a vu à la question précé-

- dente, peuvent justement coder jusqu'à $2^3 - 1 = 8 - 1 = 7$);
- L'entier 200 sur 8 bits (qui ont, on l'a vu, un maximum de 255);
 - Plus généralement, l'entier N peut se coder sur k bits, où k est tel que : $2^{k-1} \leq N < 2^k$.

Bienvenue au CP ! Commençons par les additions

Il est essentiel pour la suite de bien comprendre comment fonctionnent les additions en binaire :

- $0 + 0 = 0$ (sans retenue)
- $1 + 0 = 1$ (sans retenue)
- $1 + 1 = 0$ avec une retenue de 1
- (et attention :) s'il y avait une retenue préalable, $1 + 1 = 1$ avec une retenue de 1

Ainsi considérons l'addition des deux nombres binaires 01011011 et 00100101 :

$$\begin{array}{r} 01011011 \\ 00100101 \\ \hline 10000000 \end{array}$$

Exercice 14: additionnons donc !

Effectuez l'addition binaire des nombres 10101010 et 00010111. Ensuite, additionnez 10101010 et 11101010. Combien de bits sont nécessaires pour stocker les résultats de ces additions ?

$$\begin{array}{r} 10101010 \\ 00010111 \\ \hline 11000001 \end{array}$$

$$\begin{array}{r} 10101010 \\ 11101010 \\ \hline 110010100 \end{array}$$

Tous ces nombres peuvent être codés sur 8 bits (donc 1 octet) à l'exception du dernier où la somme de deux nombres codés sur 1 octet dépasse la valeur maximale possible (255 en décimal) et son codage dépasse donc la capacité d'un octet.

Dépassement de capacité en Python

Exercice 15: bonne nouvelle ! On passe (brièvement...) sur ordinateur !

Créez un script IDLE dans lequel vous mettez le code suivant et exécutez-le :

```
1  # on importe d'abord une bibliotheque celebre en python,  
2  # tres utilisee pour la rapidite des calculs  
3  import numpy  
4  
5  # on utilise sa fonction uint8 pour coder l'entier 250 sur 8 bits  
6  a = numpy.uint8 (250)  
7  print (bin (a))  
8  
9  for i in range (1, 10):  
10     print (f" Iteration {i}, valeur de a : {a}")  
11     print (f"en binaire : {bin (a)}")  
12     a += numpy.uint8 (1)
```

Que remarquez-vous après avoir exécuté ce code ? Pouvez-vous expliquer pourquoi cela se produit en utilisant les concepts abordés dans ce cours ?

La sortie de ce programme commence normalement :

```
Iteration 1, valeur de a : 250  
en binaire : 0b11111010
```

Puis cela se poursuit jusqu'à obtenir une erreur ressemblant à ceci :

```
RuntimeWarning: overflow encountered in ubyte_scalars  
a += numpy.uint8 (1)
```

*Ce qui se passe ici est un dépassement de capacité : on a demandé à l'ordinateur de stocker la valeur d'une variable (**a**) sur un espace mémoire limité (8 bits, donc un octet), puis on a fait augmenter la valeur de **a** jusqu'à dépasser le nombre maximal que l'on peut coder sur 8 bits et donc générer cette erreur.*

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](#))



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 "1111111" (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 "0111111", 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 des opérations 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 \bar{n} . Par exemple, le complément à 1 de 0011 est 1100.

Donc, par exemple, si fait la somme de 4 (00000100) et de $\bar{4}$, 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 15: 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])` \rightarrow `[1]`
- b. `CompUn([1])` \rightarrow `[0]`
- c. `CompUn([0, 1])` \rightarrow `[1, 0]`
- d. `CompUn([0,0,1,1,1])` \rightarrow `[1,1,0,0,0]`

```
1  def CompUn(Lst):
2      '''Fonction qui va renvoyer le complement a 1 du nombre binaire
3      passe en entree sous forme de liste'''
4
5      # On initialise le resultat: une liste vide
6      resultat = []
7
8      # On boucle sur la totalite de la liste passee en argument
9      for i in range(len(Lst)):
10         # Pour chaque element de la liste, on ajoute son contraire dans la
11         # liste resultat
12         if Lst[i] == 0:
13             resultat.append(1)
14         else:
15             resultat.append(0)
16
17     # Et pour finir on renvoie la liste resultat ainsi completee
18     return resultat
```

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 16: 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. $2_{10} + 3_{10}$

a. $-1_{10} + 1_{10}$: On vient de voir l'encodage de -1 en complément à 2 donc on peut directement poser l'opération (toujours sur 4 bits) :

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

b. $-2_{10} + -2_{10}$: Codage de -2 :

- $2_{10} = 010_2$
- Complément à 1 : 101
- Complément à 2 : $101 + 1 = 110$
- Donc le codage de -2 est, avec le bit de signe : 1110

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

c. $1_{10} + -4_{10}$: On a déjà l'encodage de -4 en complément à 2 donc on peut

directement poser l'opération :

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

Ce qui, on le verra dans le tableau plus bas, est bien le mot binaire correspondant à la valeur décimale -3 .

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

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

Ce qui, toujours en vérifiant dans le tableau plus bas, correspond bien à la valeur décimale -6 .

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

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

(ce qui correspond bien à $+5$ – juste pour confirmer que les additions d'entiers positifs fonctionnent toujours bien)



Important:

On notera bien que dans la totalité des exemples précédents où on terminait l'addition avec une retenue (donc dans les cas où la somme "réelle" des deux binaires donnait un résultat $> 1111_2$) on ignorait cette retenue – en d'autres termes on ne considérait que les 4 bits les plus à droite (donc de poids le plus faible) pour obtenir le résultat. Ces exemples étaient le a), le b), et le d). On peut aisément vérifier qu'il s'agit des exemples où soit on additionne deux valeurs négatives, soit on additionne une valeur positive à une négative de valeur absolue égale ou inférieure.

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 17: conversions en décimal

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

- a. 00001000
- b. 11111101
- c. 11111000

a. 00001000 : le bit de poids fort (le plus à gauche) est 0, donc c'est un entier positif, donc on convertit directement de binaire à décimal : en l'occurrence on voit que ce nombre est égal à 2^3 soit 8.

b. 11111101 : cette fois, le nombre est négatif :

- Inversion des bits / complément à 1 : 00000010
- Ajout de 1 : 00000011

- Conversion en base 10 : $11_2 = 3_{10}$
- Ajout du signe moins – le résultat est : -3 .

Note : pour vous en convaincre, vous pouvez poser l'addition $11111101 + 000000011$ et vérifier que vous retombez bien sur 0 (en ignorant bien sûr la retenue finale).

c. 11111000 : même démarche :

- Inversion des bits / complément à 1 : 00000111
- Ajout de 1 : 00001000
- Conversion en base 10 : $1000_2 = 8_{10}$
- Ajout du signe moins – le résultat est : -8 .

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 18: 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. $1.0 + 1.0 + 9007199254740992.0$

L’exécution des commandes ci-dessus en console Python donnent le résultat suivant :

```
In [41]: 0.5 - 0.2 - 0.2 - 0.1
Out[41]: -2.7755575615628914e-17
In [42]: 9007199254740992.0 + 2.0
Out[42]: 9007199254740994.0
In [43]: 9007199254740992.0 + 1.0 + 1.0
Out[43]: 9007199254740992.0
In [44]: 1.0 + 1.0 + 9007199254740992
Out[44]: 9007199254740994.0
```

Je pense que les erreurs et incohérences sautent aux yeux...

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}	\dots
0.5	0.25	0.125	0.0625	0.03125	\dots

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 :

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

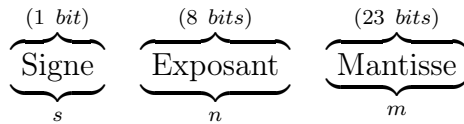
5.3 La norme IEEE 754

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 (puisque l'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).

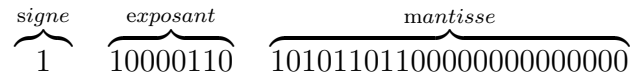
Ecrite de la sorte en mémoire et encodée sur 32 bits, la valeur d'un nombre flottant sera structurée comme suit :



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) :



Calculons alors sa valeur décimale :

$$\begin{aligned} \text{signe} &= -1^1 \\ &= -1 \end{aligned}$$

$$\begin{aligned} \text{exposant} &= 2^7 + 2^2 + 2^1 \\ &= 128 + 4 + 2 \\ &= 134 \end{aligned}$$

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

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 19: Entraînons-nous...

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

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	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	0	0

a.	0	1	0	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	0

Isolons tout d'abord les différentes composantes de ce mot binaire :

- *Signe : 0*
- *Exposant : 10000001*
- *Mantisse : 101010000000000000000000*

Comme ci-dessus, calculons à présent les valeurs de ces différentes com-

posantes :

$$\begin{aligned} \text{signe} &= -1^0 \\ &= +1 \end{aligned}$$

$$\begin{aligned} \text{exposant} &= 2^7 + 2^0 \\ &= 128 + 1 \\ &= 129 \end{aligned}$$

$$\begin{aligned} \text{mantisse} &= 2^{-1} + 2^{-3} + 2^{-5} \\ &= 0,65625 \end{aligned}$$

Appliquons enfin 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 :

$$\begin{aligned} M &= 1 \times (1 + 0,65625) \times 2^{(129-127)} \\ &= 1,65625 \times 2^2 \\ &= 6,625 \end{aligned}$$

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

Isolons tout d'abord les différentes composantes de ce mot binaire :

- Signe : 1
- Exposant : 01111101
- Mantisse : 1111000000000000000000

Comme ci-dessus, calculons à présent les valeurs de ces différentes composantes :

$$\begin{aligned} \text{signe} &= -1^1 \\ &= -1 \end{aligned}$$

$$\begin{aligned} \text{exposant} &= 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0 \\ &= 64 + 32 + 16 + 8 + 4 + 1 \\ &= 125 \end{aligned}$$

$$\begin{aligned} \text{mantisse} &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} \\ &= 0,9375 \end{aligned}$$

Appliquons enfin 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 :

$$\begin{aligned} M &= -1 \times (1 + 0,9375) \times 2^{(125-127)} \\ &= -1,9375 \times 2^{-2} \\ &= -1,9375 \times 0,25 \\ &= -0,484375 \end{aligned}$$

Exercice 20: C'est un peu fastidieux... et si on codait ça ?

- 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 décimale du réel correspondant.
- 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)...

Entrée: *Lst*, liste de 32 bits ($\in \{0,1\}$)

Sortie: *res* $\in \mathbb{R}$

```
1: fonction CONVBINENREEL(Lst)
2:   exposant  $\leftarrow 0$ 
3:   mantisse  $\leftarrow 0.0$  ▷ C'est un nb réel  $\in [0, 1[$ 
4:
5:   signe  $\leftarrow -1^{Lst[0]}$  ▷ Calcul signe sur le 1er elt de la liste d'entrée
6:
7:   pour i = 1 à 8 faire ▷ Calcul exposant sur les 8 elts suivants
8:     puiss  $\leftarrow 8 - i$  ▷ indice 1  $\leftrightarrow 2^7$  ; i = 2  $\leftrightarrow 2^6$  ; ... ; i = 8  $\leftrightarrow 2^0$ 
9:     si Lst[i] = 1 alors
10:       exposant  $\leftarrow \text{exposant} + 2^{\text{puiss}}$ 
11:     fin si
12:   fin pour
13:   exposant  $\leftarrow \text{exposant} - 127$  ▷ Application du décalage sur 32 bits
14:
15:   pour i = 9 à 31 faire ▷ Calcul mantisse sur les 23 derniers elts
16:     puiss  $\leftarrow 8 - i$  ▷ indice 9  $\leftrightarrow 2^{-1}$  ; i = 10  $\leftrightarrow 2^{-2}$  ; ... ; i = 31  $\leftrightarrow 2^{-23}$ 
17:     si Lst[i] = 1 alors
18:       mantisse  $\leftarrow \text{mantisse} + 2^{\text{puiss}}$ 
19:     fin si
20:   fin pour
21:   mantisse  $\leftarrow 1 + \text{mantisse}$  ▷ Application du "(1+m)" de la formule
22:
23:   resultat  $\leftarrow \text{signe} \times \text{mantisse} \times 2^{\text{exposant}}$ 
24:
25:   retourner resultat
26: fin fonction
```

```

1  def Conv2Float(Lst):
2      '''Fonction qui va convertir une liste de 32 bits en nombre reel
3      ↪ (flottant) en s'appuyant sur la norme IEE-754 en simple precision'''
4
5      # On initialise les variables de travail
6      exposant = 0
7      mantisse = 0.0 #(c'est un nombre reel compris entre 0 et 1)
8
9      # On fixe tout d'abord le signe - bit de poids fort, le plus a
10     ↪ gauche, donc element 0 de la liste
11     signe = Lst[0]
12
13     # On effectue ensuite les calculs de l'exposant. On sait qu'ils sont
14     ↪ codes sur 8 bits, du 2eme au 9eme, donc des indices 1 a 8.
15     ↪ Correspondance indice <-> puissance: indice 1 <-> puissance 7 de 2;
16     ↪ 2 <-> 6; (...); 8 <-> 0.
17     for i in range(1,9):
18         exposant += Lst[i] * 2 ** (8 - i)
19
20     # On applique le decalage sur 32 bits, qui est de 127
21     exposant -= 127 # (cette syntaxe est equivalente a "exposant = exposant
22     ↪ - 127")
23
24     # On calcule enfin la mantisse, sur les bits restants, donc d'indice 9 a
25     ↪ 31. Correspondance indice <-> puissance: indice 9 <-> puissance (-1)
26     ↪ de 2; 10 <-> (-2); (...); 31 <-> (-23).
27     for i in range(9, 32):
28         mantisse += Lst[i] * 2 ** (8 - i)
29
30     # Ne reste qu'a calculer le resultat en utilisant la formule vue en
31     ↪ cours: [(-1) ** signe] * [1 + mantisse] * [2 ** exposant]
32     resultat = ((-1) ** signe) * (1 + mantisse) * (2 ** exposant)
33
34     return resultat
35
36 # Exemples de test:
37
38 # Exercice 19.a - resultat = 6.625
39 liste = [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]
40 print(Conv2Float(liste))
41
42 # Exercice 19.b - resultat = -0,484375
43 liste = [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]
44 print(Conv2Float(liste))
45
46 # Et un dernier exemple de test que vous devriez reconnaitre...
47 liste = [0,1,0,0,0,0,0,0,0,1,0,0,1,0,0,1,0,0,0,1,1,1,1,1,1,0,1,1,0,1,1,1]
48 print(Conv2Float(liste))

```

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 – 0011001100110011001

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 connaître 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??

Il manque énormément de caractères – et d'autant plus si on n'est pas anglophone! Déjà, pour les francophones, on note qu'il n'y a aucun accent. Mais que devaient penser les Arabes, les Chinois, les Russes...?

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

On compte 128 caractères soit (et ce n'est évidemment pas un hasard) 2^7 . Il serait donc possible de n'utiliser que 7 bits pour coder les caractères ASCII.

En pratique, on a toujours codé ces caractères sur un octet (donc 8 bits) en utilisant à l'origine le bit de poids fort comme un mécanisme de contrôle d'erreur (qui étaient fréquentes avec les premières mémoires électroniques). Le principe était d'en faire un bit de parité: il était positionné à 0 ou à 1 de manière à ce que tous les caractères aient toujours un nombre pair de bits à 1; et si ce n'était pas le cas, une erreur était détectée.

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^{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 cyrilliques (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:

- Ça 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 (puisque'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 l'exemple typique de la lumière mise par un phare maritime : elle est d'abord indivisible, son coût de production est alors indépendant du nombre d'utilisateurs ; elle possède une propriété de non-rivalité (elle ne se détruit pas dans l'usage et peut donc être adoptée par un nombre illimité d'utilisateurs) ; elle est également non excluable car il est impossible d'exclure de l'usage un utilisateur, même si ce dernier ne contribue pas à 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 chaînes 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 chaînes 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 délimiteurs 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 délimiteur.

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 délimiteurs:

```
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` également (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 21: manipulons quelques chaînes de caractères...

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

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

a. En pseudo-code:

Entrée: *a* et *b*, chaînes de caractères

Sortie: *resultat*, chaîne de caractères

1: **fonction** `CONCAT1(a, b)`

```

2:   resultat ← a           ▷ On commence par la première chaîne
3:   resultat ← resultat + " "           ▷ On ajoute l'espace
4:   resultat ← resultat + b           ▷ On ajoute la seconde chaîne
5:   retourner resultat       ▷ C'est fini - on renvoie le résultat
6: fin fonction

```

Ce qui donne en Python:

```

1  def concat1(a, b):
2      # Utilisation de l'opérateur "+" pour concaténer des chaînes de
      ↪ caractères
3      resultat = a + " " + b
4      return resultat

```

- b. En pseudo-code – presque la même chose, évidemment, sauf que l'on vérifie au préalable la fin de la chaîne a et le début de la chaîne b (on ne couvre pas ici le cas où a finit par un espace ET b commence par un espace)

Entrée: a et b, chaînes de caractères

Sortie: resultat, chaîne de caractères

```

1: fonction CONCAT2(a, b)
2:   LongA ← len(a) ▷ On récupère la longueur de la première chaîne
   de caractères
3:   DernA ← a[LongA]           ▷ On isole le dernier caractère de a
4:   PremB ← b[0]               ▷ On isole le premier caractère de b
5:   resultat ← a               ▷ On commence par la première chaîne
6:   si DernA ≠ " " ET PremB ≠ " " alors           ▷ PAS d'espace
7:     resultat ← resultat + " "           ▷ On en ajoute donc un
8:   fin si
9:   resultat ← resultat + b ▷ On ajoute la seconde chaîne dans tous
   les cas
10:  retourner resultat           ▷ C'est fini - on renvoie le résultat
11: fin fonction

```

Ce qui donne en Python:

```

1  def concat2(a, b):
2      # Dernier caractère de a
3      DernA = a[len(a) - 1]
4      # Premier caractère de b
5      PremB = b[0]
6
7      resultat = a
8
9      if DernA != " " and PremB != " ":
10         resultat += " " # on ajoute l'espace s'il n'y en a pas
11
12         resultat += b
13     return resultat

```

- c. En pseudo-code:

Entrée: chn, une chaîne de caractères

Sortie: resultat, chaîne de caractères


```

1: fonction MILIEU(chn)
2:   LongChn  $\leftarrow$  len(chn)  $\triangleright$  Longueur de la chaîne
3:   si LongChn est paire alors
4:     resultat  $\leftarrow$  "XX"
5:   sinon
6:     centre  $\leftarrow$  LongChn // 2
7:     resultat  $\leftarrow$  chn[centre]
8:   fin si
9:   retourner resultat
10: fin fonction

```

Vérifions que le calcul ci-dessus fonctionne sur un exemple: imaginons qu'en entrée on ait *chn* = [ABCDE]. Alors:

- *LongChn* = 5 (donc la longueur est bien impaire)
- *centre* = 5 // 2 = 2
- Et donc *chn*[*centre*] = *chn*[2] = "C" (puisque, comme pour les listes, les indices commencent à 0) – ce qui est bien ce qu'on voulait.

Ce qui donne en Python:

```

1  def milieu(chn):
2      # Longueur de la chaîne
3      LongChn = len(chn)
4      if LongChn % 2 == 0:
5          resultat = "XX"
6      else:
7          centre = LongChn // 2
8          resultat = chn[centre]
9
10     return resultat

```

d. En pseudo-code:

Entrée: *nom*: chaîne de caractères débutant par une majuscule

Sortie: *resultat*, valant 1, 2, ou 3

```

1: fonction AFFECTATION(nom)
2:   Initiale  $\leftarrow$  nom[0]  $\triangleright$  On ne vérifie pas que c'est une majuscule
3:   si Initiale  $\leq$  H alors
4:     resultat  $\leftarrow$  1
5:   sinon si Initiale  $\leq$  Q alors
6:     resultat  $\leftarrow$  2
7:   sinon
8:     resultat  $\leftarrow$  3
9:   fin si
10:  retourner resultat
11: fin fonction

```

On s'appuie ici sur le fait que le code ASCII des lettres majuscules est séquentiel, commençant (en hexadécimal) à 41 pour A et allant jusqu'à 5A pour Z (ce qui fait que les comparaisons numériques implicites utilisant le symbole \leq fonctionnent bien).

Ce qui donne en Python:

```
1  def affectation(nom):
2      # Récupération de l'initiale
3      Initiale = nom[0]
4      if Initiale <= "H":
5          resultat = 1
6      elif Initiale <= "Q":
7          resultat = 2
8      else:
9          resultat = 3
10
11     return resultat
```

- e. Cet exercice reprend l'ensemble de ce qu'on a vu dans les précédents et y ajoute le parcours d'une chaîne de caractères au moyen d'une boucle comme on sait déjà le faire pour une liste – il est donc très complet, et je ne saurais trop vous conseiller de bien le comprendre / l'assimiler en vue des contrôles à venir...

En pseudo-code:

Entrée: chn, chaîne de caractères

Sortie: resultat, chaîne de caractères

```
1: fonction DECOUPAGE(chn)
2:   LongChn ← len(chn)                                ▷ Nombre de caractères
3:   resultat ← ""                                       ▷ Initialisation du résultat: chaîne vide
4:   pour tout caractere de chaine faire
5:     si resultat ≠ "" alors                            ▷ Pas de virgule avant 1er caractère
6:       resultat ← resultat + ","
7:     fin si
8:     resultat ← resultat + caractere                  ▷ Ajout caractère suivant
9:   fin pour
10:  retourner resultat
11: fin fonction
```

On voit bien dans le pseudo-code ci-dessus et dans le code Python ci-dessous qu'on parcourt ici la chaîne de caractères exactement de la même manière qu'on parcourt habituellement les éléments d'une liste.

Ce qui donne en Python:

```

1  def decoupage(chn):
2      # Récupération de la longueur de la chaîne
3      LongChn = len(chn)
4
5      # Initialisation du résultat - chaîne vide
6      resultat = ""
7
8      # Parcours de tous les éléments (caractères) de la chaîne
9      for i in range(LongChn):
10         # On n'ajoute de virgule que s'il y a déjà des caractères dans
            ↳ le résultat
11         if resultat != "":
12             resultat += ","
13
14         # On ajoute le caractère en cours
15         resultat += chn[i]
16
17     return resultat

```

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 22: 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()`

- a. Renvoie le nombre de fois que l'on trouve `chaine2` dans `chaine1`. Donc `"ABACAD".count("A")` renvoie 3.
- b. Renvoie la position de la première occurrence (dans le sens de la lecture – donc en partant de la gauche) de `chaine2` dans `chaine1`. Donc `"ABCD".find("C")` renvoie 2 (puisque les indices commencent à 0)
- c. Renvoie `chaine1` dans laquelle toutes les instances de `chaine2` auront été remplacées par `chaine3`. Donc `"ABACD".replace("A", "99")` renvoie `"99B99CD"`.
- d. Renvoie `chaine1` avec le premier caractère en majuscule et tous les autres en minuscules. Donc `"aBBgD".capitalize()` renvoie `"Abbgd"`.
- e. Renvoie `True` si tous les caractères de `chaine1` sont des majuscules, `False` sinon.
- f. Renvoie `chaine1` avec tous ses caractères passés en majuscules. Donc `"AbCd".upper()` renverra `"ABCD"`.
- g. Renvoie `True` si tous les caractères de `chaine1` sont des minuscules, `False` sinon.
- h. Renvoie `chaine1` avec tous ses caractères passés en minuscules. Donc `"AbCd".lower()` renverra `"abcd"`.
- i. Renvoie `chaine1` avec toutes ses majuscules passées en minuscules et réciproquement. Donc `"AbCd".swapcase()` renverra `"aBcD"`.

Exercice 23: 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)`

- a. Renvoie 97, ce qui est (en base décimale) le **code ASCII** de la lettre "a".
- b. Renvoie 90, ce qui est (en base décimale) le **code ASCII** de la lettre "Z".
- c. Renvoie 'z', dont le **code ASCII** (en base décimale) est 122.
- d. Renvoie 'M', dont le **code ASCII** (en base hexadécimale) est 4D (on rappelle que l'opérateur "0x" permet d'entrer des valeurs hexadécimales dans Python).

Exercice 24: 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.

Celui-là, je ne vous en donne pas la solution – je le laisse comme défi à ceux qui veulent s'y essayer. Envoyez-moi vos solutions si vous voulez!

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 25: table de vérité de l'opérateur "non"

Compléter la table de vérité suivante:

a	$\text{non}(a)$
0	
1	

a	$\text{non}(a)$
0	1
1	0

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 26: 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	

a	b	a ou b
1	1	1
1	0	1
0	1	1
0	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 27: 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	

a	b	a et b
1	1	1
1	0	0
0	1	0
0	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 28: table de vérité de l'opérateur "xor"

Compléter la table de vérité suivante:

a	b	$a \text{ xor } b$
1	1	
1	0	
0	1	
0	0	

a	b	$a \text{ xor } b$
1	1	0
1	0	1
0	1	1
0	0	0

En Python, l'opérateur "xor" est \wedge .

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 $(a \text{ et } b)$ ou $\text{non}(c)$ est une expression booléenne – j'imagine que vous ne serez pas surpris que maintenant je vous demande...

Exercice 29: 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	$a \text{ et } b$	$\text{non}(c)$	$(a \text{ et } b) \text{ ou } \text{non}(c)$
1	1	1			
1	1	0			
1	0	1			
1	0	0			
0	1	1			
0	1	0			
0	0	1			
0	0	0			

a	b	c	$a \text{ et } b$	$\text{non}(c)$	$(a \text{ et } b) \text{ ou } \text{non}(c)$
1	1	1	1	0	1
1	1	0	1	1	1
1	0	1	0	0	0
1	0	0	0	1	1
0	1	1	0	0	0
0	1	0	0	1	1
0	0	1	0	0	0
0	0	0	0	1	1

Exercice 30: 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. $f(a, b, c) = \text{non}(a \text{ et } \text{non}(b)) \text{ ou } c$
- b. $g(a, b, c) = (a \text{ xor } \text{non}(b)) \text{ et } c$
- c. $h(a, b, c) = \text{non}(a \text{ et } b) \text{ et } (b \text{ ou } c)$

a.

a	b	c	$\text{non}(b)$	$a \text{ et } \text{non}(b)$	$\text{non}(a \text{ et } \text{non}(b))$	$f(a, b, c)$
1	1	1	0	0	1	1
1	1	0	0	0	1	1
1	0	1	1	1	0	1
1	0	0	1	1	0	0
0	1	1	0	0	1	1
0	1	0	0	0	1	1
0	0	1	1	0	1	1
0	0	0	1	0	1	1

b.

a	b	c	$\text{non}(b)$	$a \text{ xor } \text{non}(b)$	$g(a, b, c)$
1	1	1	0	1	1
1	1	0	0	1	0
1	0	1	1	0	0
1	0	0	1	0	0
0	1	1	0	0	0
0	1	0	0	0	0
0	0	1	1	1	1
0	0	0	1	1	0

c.

a	b	c	$a \text{ et } b$	$\text{non}(a \text{ et } b)$	$b \text{ ou } c$	$h(a, b, c)$
1	1	1	1	0	1	0
1	1	0	1	0	1	0
1	0	1	0	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	1
0	1	0	0	1	1	1
0	0	1	0	1	1	1
0	0	0	0	1	0	0

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 1^{ère} – 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.