Représentation des Données

I- Rappel

Nous avons vu en Seconde que toute l'information qui transite dans un ordinateur est codée avec des bits (**B**inary dig**IT**) ne prenant que les valeurs 0 et 1.
Ce type de codage logique est très facilement mis en œuvre dans un circuit électronique d'ordinateur : 0 : pas de courant, 1 : le courant passe. (Utilisation de « transistors »)

Pour coder de l'information, que ce soient des nombres, (PI=3,141592...), du texte (ce cours), des schémas, des images, des sons , les circuits électroniques d'un ordinateur classique ne peuvent utiliser que des informations en binaire.

Un élément binaire, un BIT (pour Binary Digit) peut prendre deux valeurs possibles : 0 ou 1
Un mot binaire de n bits est un ensemble de n bits. Par exemple (compléter) :

0111 est un mot de 4 bits 01111001 est un mot de ……. ? bits

1001 001 est un mot de 7bits 110 est mot de ……. ?bits

(Note : pour simplifier la lecture, on écrira les mots par bloc de 4 bits)

II- Codage , nombre de combinaisons

Mot de 1bit : 0 ou 1 . On a donc 2 combinaisons possibles
Mot de 2bits : on peut avoir 00 01 10 11 soit 4 combinaisons possibles
Mot de 3bits : on a 8 combinaisons possibles, donner les dans le tableau suivant :

C	B	A
0	0	0

D'une manière générale, un mot de n bits permet de coder 2^n combinaisons
1 bit : 2^1 = 2 combinaisons
2 bits : 2^2 = 4 combinaisons
4 bits : 2^4 = 16 combinaisons
8 bits : 2^8 = …… combinaisons
16 bits : 2^{16} = …… combinaisons

Combien de bits sont nécessaires pour coder 2048 combinaisons (justifier la réponse) :

Combien de bits sont nécessaires pour coder 27 combinaisons (justifier la réponse) :

Y a t-il alors des combinaisons inutilisées parmi toutes celles possibles, et si oui combien ?

III- Quartet, Octet, Kbit, Koctets, Moctets, Giga octets

Note : bien que ne correspondant plus à la dernière norme en vigueur , la notation ci-dessous reste encore très largement utilisée dans la littérature scientifique et en informatique.

Un quartet est mot de 4 bits . ex : 1101
Un octet est mot de 8 bits : ex : 0110 1111
Un Kbit (Kilo Bit) = 2^{10} bits = 1024 bits
1 KO (Kilo Octets) = 2^{10} octets = 1024 octets
1 MO (méga Octets) = 1KO * 1KO = 2^{20} octets (soit 1024 * 1024 octets)
1 GO (Giga Octets) = 1KO * 1KO * 1KO = 1024 Méga Octets
1 TO (Terra Octets) = 1KO * 1GO = 1024 Giga Octets

Nouvelle Normes :

KbiBit
KbiOctet
MbiOctet
GbiOctet
TbiOctet

Note : Un octet = 8bits. Donc 1KO = 2^{10} octets = 1024* 8bits = 8192 bits (2^{13})
2KO = 2 * 1KO = 2*1024 Octets = 2048 Octets
4KO = 4 * 1KO = 4*1024 Octets = 4096 Octets

8KO = KO = Octets

1MO = KO= Octets = ... Bits

Dans un mot binaire on repère deux bits importants :

Le bit de poids fort → 1011 1100 ← le bit de poids faible
MSB : Most Significant Bit **LSB :** less Significant Bit

IV- Les principaux codes numériques

L'écriture d'un nombre peut se faire dans différentes « bases » .
La base 10 ou décimale que nous connaissons tous (10 chiffres de 0 à 9)
La base 2 ou binaire (avec des 2 chiffres 0 et 1)
La base 16 ou hexadécimale (avec 16 chiffres de 0 à 9 et A à F)
La base 16 est très utilisée par exemple pour coder les couleurs dans une page web en html-CSS ou encore la représentation d'une adresse IP pour la norme V6. Elle a l'avantage d'être « compacte » à l'écriture.

Base 10	Binaire pur	BCD	Hexadécimal
0	0000	0000 0000	0
1	0001	0000 0001	1
2	0010	0000 0010	2
3	0011	0000 0011	3
4	0100	0000 0100	4
5	0101	0000 0101	5
6	0110	0000 0110	6
7	0111	0000 0111	7
8	1000	0000 1000	8
9	1001	0000 1001	9
10	1010	0001 0000	A
11	1011	0001 0001	B
12	1100	0001 0010	C
13	1101	0001 0011	D
14	1110	0001 0100	E
15	1111	0001 0101	F
16	10000	0001 0110	10
17	10001	0001 0111	11
18	10010	0001 1000	12
19	10011	0001 1001	13
20	10100	0010 0000	14

En base 10, les chiffres vont de 0 à 9
En base 2 , les chiffres vont de 0 à 1
En base 16, les chiffres vont de 0 à F (après 9, on utilise des lettres A,B,C,D,E,F)

Conversion Binaire → Décimal

Un nombre en binaire sur 8 bits peut s'écrire :

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
b7	**b6**	**b5**	**b4**	**b3**	**b2**	**b1**	**b0**

Poids décimal de chaque bit

Nombre binaire

On peut écrire la conversion :

$Y_{(10)} = b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_{0\,(2)} = b_7 * 2^7 + b_6 * 2^{6\ +} b_5 * 2^5 + \ldots\ldots + b_0 * 2^0$

Exemple :

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

$0010\ 1001_{(2)} = 1 + 8 + 32 = 41_{(10)}$

Exercice : Donner les valeurs décimal des nombres binaires suivants :

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

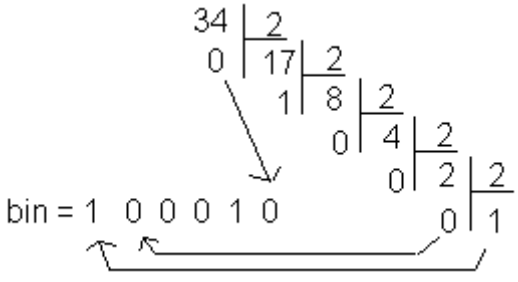
128	64	32	16	8	4	2	1
0	1	0	1	0	0	1	1

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

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

Conversion décimal → Binaire (base 2)

a- Méthode de la division par la base :



Dans cette méthode, on divise successivement le nombre à convertir par 2 (car base 2) .
Au reste des divisions correspond le nombre binaire. Attention au sens de lecture (du bas (MSB) vers le haut (LSB))

Exercice :
Convertir 56 et 107 en base 2 par la méthode de la division par la base
vérifier que vous trouvez bien : 0011 1000 (pour 56) et 01101011 (pour 107)

b- Méthode « manuelle »

Le principe est d'additionner les poids (en commençant par le plus fort possible) afin d'obtenir le nombre décimal voulu :
Exemple : pour convertir 67 :

128	64	32	16	8	4	2	1

128 est trop grand
64 : ok je mets « 1 » sous le poids 64
64 + 32 = 96 → trop grand
64 + 4 = 68 → trop grand
64 + 2 = 66 → ok, je mets « 1 » sous le poids 2
66 + 1 = 67 → ok, je mets « 1 » sous le poids 1

J'obtiens donc en binaire : 0100 0011

Exercice :
S'entraîner à convertir par cette méthode :

25, 36, 155, 128, 200, 840

Eléments de corrections :

25 : 0001 1001
36 : 0010 0100

…

Conversion Binaire → Hexadécimal

Pour convertir du binaire vers l'hexadécimal, on va faire des blocs de 4 bits (quartets) et positionner les poids en fonction de ces blocs :
Exemple :

8	4	2	1	8	4	2	1
1	0	0	1	1	1	0	0
8+1=9				8+4=12 = C en hexa			

La conversion donne donc : 9C

S'entraîner :

8	4	2	1	8	4	2	1
	0	0	1	0	1	1	1

8	4	2	1	8	4	2	1
1	0	1	1	1	1	0	1

8	4	2	1	8	4	2	1
1	1	1	1	1	1	1	1

Conversion Hexadécimal → Binaire

Exemple A9 : **A** = 10 = 8 + 2 et **9** = 8 + 1 , donc on obtient :

8	4	2	1	8	4	2	1
A = 10 = 8 + 2				9 = 8 + 1			
1	0	1	0	1	0	0	1

S'entraîner : convertir en binaire les nombres hexa suivants : 5B , 31, F7, 11, FFA, D4

8	4	2	1	8	4	2	1

8	4	2	1	8	4	2	1

V- Représentation des nombres "signés" et décimaux . Problèmes et limites.

Pour obtenir un nombre négatif, on va partir du nombre positif et effectuer ce qu'on appelle le "complément à 2".
On doit imposer le format (par exemple 8 bits ou 16 bits ...)
Exemple sur 8 bits :

On veut représenter -3 :
3 en binaire = 0 0 0 0 0 0 1 1
Complément à 1 : 1 1 1 1 1 1 0 0
+ 1
Complément à 2 : **1 1 1 1 1 1 0 1** = -3

On veut faire 8 - 3 :
1 0 0 0
+ 1 1 0 1
= 0 1 0 1 = 5 en décimal

On veut faire 3 - 8 :
8 : 0000 1000
1111 0111
+ 1
-8 : 1111 1000

0 0 0 0 0 0 1 1 3
+ 1 1 1 1 1 0 0 0 - 8
= 1 1 1 1 1 0 1 1 = -5

Sur 8 bits, on pourra donc représenter les nombres "signés" de :
-128 à +127

-128 = 1000 0000

+127= 0111 1111

Sur 16 bits, on pourra donc représenter les nombres "signés" de :
-32768 à +32767 soit de

1000 0000 0000 0000 à 0111 1111 1111 1111

Exercice :
Représenter les nombre signés sur 8 bits suivants en binaire :
-1 -64 -31 -125

Conversion réel décimal en base 2

Exemple : conversion de 12,6875 en binaire

Conversion de 12 : donne (1100)2
Conversion de 0,6875
0,6875 x 2 = 1,375 = 1 + 0,375
0,375 x 2 = 0,75 = **0** + 0,75
0,75 x 2 = 1,5 = **1** + 0,5
0,5 x 2 = 1 = **1** + 0

D'où $(12,6875)_{10} = (1100,1011)_2$

Conversion réel décimal en base 16 (hexa)

Exemple : conversion de 171,3046875 en hexadécimal :

Conversion de 171 : donne $(AB)_{16}$
Conversion de 0,3046875 :

0,3046875 x 16 = 4,875 = 4 + 0,875
0,875 x 16 = 14,0 = 14 + 0
$(171,3046875)10 = (AB,4E)_{16}$

Note : La représentation de nombres signés à virgules ou de façon plus générale de nombres avec réels avec exposants sort du cadre du programme et ne sera pas abordé.

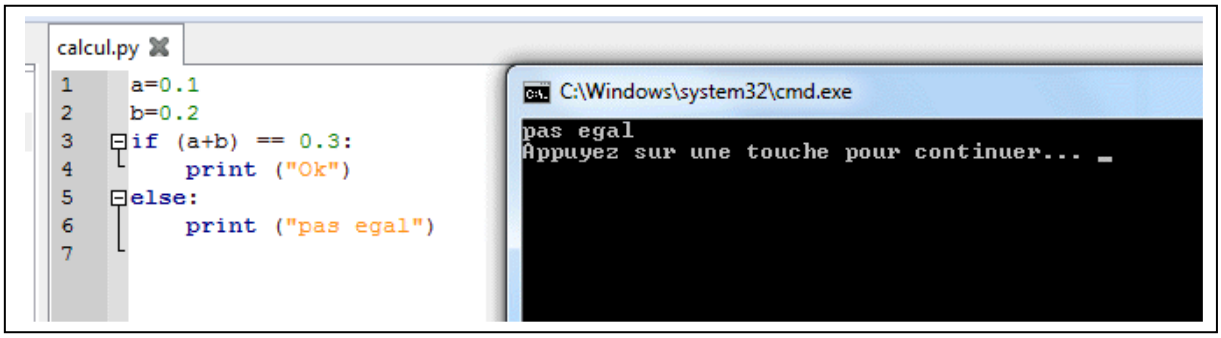
Pour s'entrainer :
Représenter les nombres entiers suivants en binaire (8 bits) :
-25 -15 -112 -1 -13

Effectuer les opérations en binaire:
23 - 15 =
130 - 55 =

Représenter les nombres décimaux suivants en binaire (sur 8bits après la virgule) :
0.1 0.3 12,4 0,25 32,06

Les nombres à virgule flottante sont représentés, au niveau matériel, en fractions de nombres binaires (base 2). La conséquence est que lorsque vous faites par exemple :
A = 0.1 (en décimal)
En interne , en binaire :
A = 0.0001100110011001100110011001100110011001100110011001100110011010

Ce qui correspond en réalité en décimal à :
0.1000000000000000055511151231257827021181583404541015625



En conséquence il ne faudra jamais faire de test d'égalité avec des nombres décimaux au risque de se retrouver avec des erreurs inhérentes au stockage en interne en binaire. Ceci n'est pas une spécificité de python mais de tous les langages.

Les types en python (python 2.7 et antérieures):
Le type int sera de 32 bits, bit de signe inclus (c-a-d 31 bits pour le nombre)
Soit +/- 2 147 483 647 en décimal
Le type "long" si le nombre est supérieur à 31 bits
Le type "float" pour les nombres à virgule
Le type "complex" pour les nombres complexes.

Depuis la version 3 de python, le type int est étendu (long integer) et il n'y plus besoin du type long puisque le type int accepte maintenant des nombres non limités en taille

Pour connaître le type d'une variable , on utilisera l'instruction "type(valeur)"
Exécutons le code ci-dessous (en python 2.7) :