

# Représentation des nombres

# Binaire / Octet

Les nombres sont représentés en binaires:

ex. pour des entiers

2	10
6	110
156	10011100
86751	10101001011011111

un '1' ou un '0'  
est un "bit"

Un **octet** est un groupe de 8 bits

10011100 00101101  
1<sup>er</sup> octet    2<sup>eme</sup> octet

# Binaire / Octet / Hexadécimale

Il est pratique de représenter les nombres binaires en hexadécimale

1 octet (8 bits) se représente par 2 caractères hexadécimaux

ex.

$\underbrace{1001}_{9} \underbrace{1100}_{C} \Rightarrow 9C \text{ en hexadécimale}$

On a bien: 156 en base 10, 10011100 en base 2, 9C en base 16

ex2.

Le nombre 8A4F en hexadécimale vaut:

1000101001001111 en base 2

35407 en base 10

Ce nombre peut se représenter sur 2 octets

# Utilisation mémoire

Si on considère un nombre stocké sur 4 octets, on peut compter de

00 00 00 00 à FF FF FF FF

0 à 4294967295 en base 10

*Rem. 4294967295 octets ~ 4Go*

*= limite maximum de RAM sur les anciens PC  
(car adresses stockés sur 4 octets)*

Mais, uniquement nombre positifs !!

Comment représenter -5 ?

# Nombres signés

On utilise un bit pour savoir si le nombre est positif ou négatif

ex.

**0**100

est un nombre positif

**1**100

est un nombre négatif

Attention: les nombres négatifs sont évalués avec le complément à 2+1

ex. un nombre de 2 octets 1100 0100 1000 1101

**1**100 0100 1000 1101 = C4 8D (en hexa)

Ce nombre est négatif

complément à 1  
011 1011 0111 0010

+ 1  
011 1011 0111 0011

**-15219** (en base 10)

# Exemples nombres entiers

Ex. Pour des nombres entiers stockés sur 4 octets

Binaire	hexa	Valeur en base 10
0110 0010 0000 1111 0010 1000 1111 0011	62 0F 28 F3	1645160691
1110 0010 0000 1111 0010 1000 1111 0011	E2 0F 28 F3	-502322957
0000 0000 0000 0000 0000 0000 0000 0001	00 00 00 01	1
1000 0000 0000 0000 0000 0000 0000 0000	80 00 00 00	-2147483648
1111 1111 1111 1111 1111 1111 1111 1111	FF FF FF FF	-1
0111 1111 1111 1111 1111 1111 1111 1111	7F FF FF FF	2147483647

# Lecture processeur

Un nombre en hexadécimale peut être lue par (au moins) 2 manière:

ex: 62 0F 28 F3

Peut être lu:

62  
0F  
28  
F3

F3  
28  
0F  
62

Les octets de poids forts  
sont lus en premier

= **Big Endian**

Les octets de poids faibles  
sont lus en premier

= **Little Endian**

Les PC standards sont en Little Endian  
(Anciens Mac étaient en Big Endian)

# Little Endian

On stocke en mémoire (RAM) les données avec l'octet de poids faible en premier

ex.        1568 en base 10

= 00 00 06 20 en hexadécimale (sur 4 octets)

est stocké en RAM en tant que:

**20 06 00 00**

Soit en binaire:

00100000 00000110 00000000 00000000

*(Ce stockage dépend des processeurs:*

*20 06 00 00 ne correspond pas au même nombre*

*sur un pentium (little endian) et sur un ARM en big endian)*



# Stockage nombre entiers

Cas d'un processeur en Little Endian

Valeur en base 10	hexadécimale en mémoire	Binaire en mémoire
1645160691	F3 28 0F 62	0011 1111 1000 0010 1111 0000 0010 0110
-502322957	F3 28 0F E2	0011 1111 1000 0010 1111 0000 0010 1110
1	01 00 00 00	0001 0000 0000 0000 0000 0000 0000 0000
-2147483648	00 00 00 80	0000 0000 0000 0000 0000 0000 0000 1000
-1	FF FF FF FF	1111 1111 1111 1111 1111 1111 1111 1111
2147483647	FF FF FF 7F	1111 1111 1111 1111 1111 1111 1111 0111

# La taille des données

L'encodage dépend du nombre d'octet alloué pour chaque nombre

On considère généralement les types suivants (en C par exemple):

- **short**: nombre entier sur 2 octets
- **int**: nombre entier sur 4 octets
- **long**: nombre entier sur 8 octets
- **long long**: nombre entier sur 16 octets (parfois 8 octets)

*Attention: le type "int" en Python n'a pas une taille fixe  
l'encodage est masqué/transparent pour l'utilisateur (contrairement au C)  
Il s'agit d'un type différent que le type "int" standard du C*

# Exemple de type d'entiers

ex.

Le nombre 4876 en base 10

4876 base 10 => 13 0C en hexa

Soit 00010011 00001100 en binaire

Sur une architecture Big Endian:

Type	Stockage en binaire	Stockage en hexa
short	00001100 00010011	13 0C
int	00001100 00010011 00000000 00000000	13 0C 00 00
long	00001100 00010011 00000000 00000000 00000000 00000000 00000000 00000000	13 0C 00 00 00 00 00 00
long long	00001100 00010011 00000000 00000000 . . .	13 0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00

# Exemple de type d'entiers (2)

ex.

Le nombre -4876 en base 10

Sur une architecture Big Endian:

Type	Stockage en binaire	Stockage en hexa
short	11110100 11101100	F4 EC
int	11110100 11101100 11111111 11111111	F4 EC FF FF
long	11110100 11101100 11111111 11111111 11111111 11111111 11111111 11111111	F4 EC FF FF FF FF FF FF
long long	11110100 11101100 11111111 11111111 . . .	F4 EC FF FF FF FF FF FF FF FF FF FF FF FF FF FF

# Visualiser stockage en Python

Attention: La commande hex() Python affiche la valeur hexadécimale indépendante de l'encodage machine

ex. hex(-145)

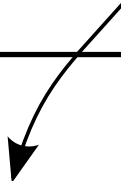
> '-0x91' *('-' ne correspond pas à un symbol hexadécimale)*  
    hexa

Pour visualiser le vrai encodage machine:

```
import binascii
import struct

stockage_hexa=binascii.hexlify(struct.pack('i',145))
print(stockage_hexa)
```

type 'int'  
sur 4 octets



# Visualiser stockage en Python

On peut afficher les différents types:

```
struct.pack(type, nombre)
```



h : short  
i : int  
l : long  
q : long long

```
stockage_hexa=binascii.hexlify(struct.pack('l',145))
```

```
> 91 00 00 00 00 00 00 00
```

```
stockage_hexa=binascii.hexlify(struct.pack('l',-145))
```

```
> 6F FF FF FF FF FF FF FF
```

# Stockage vers nombre

Opération inverse:

```
valeur_decimale=struct.unpack('i',binascii.unhexlify(b'91000000'))  
print(valeur_decimale)
```

> 145

```
valeur_decimale=struct.unpack('i',binascii.unhexlify(b'6FFFFFFF'))  
print(valeur_decimale)
```

> -145

# Nombres flottants

L'encodage des nombres flottants est codifié par la norme IEEE 754

Le type 'float' du C est un nombre flottant stocké sur 4 octets

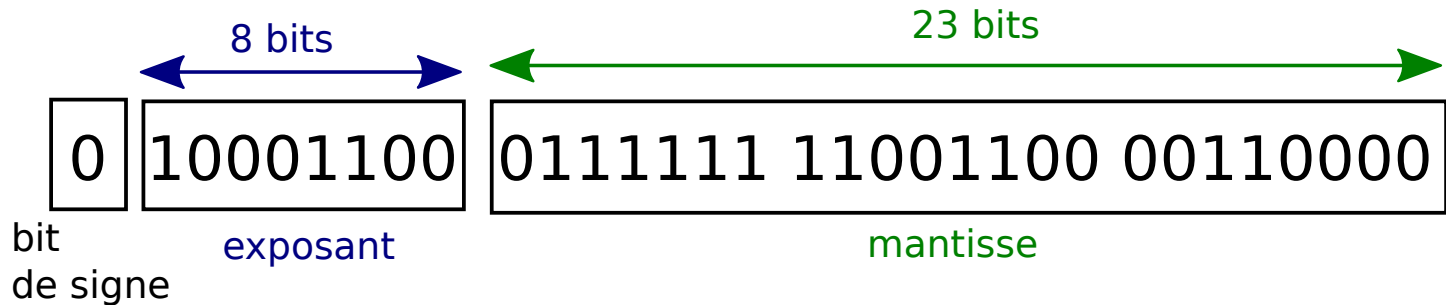
Le type 'double' du C est un nombre flottant stocké sur 8 octets  
*(double précision)*

*Attention: L'appellation 'float' de Python réfère à un double précision*



# Nombres flottants

Soit le nombre binaire: 01000110 00111111 11001100 00110000



Soit

s: le bit de signe

e: le nombre décimale correspondant à l'exposant

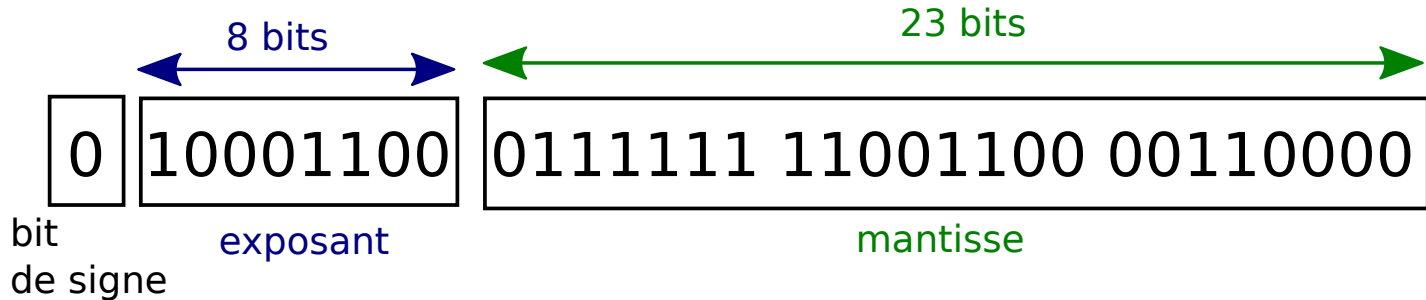
$b_i$ : les bits de la mantisse ( $i$  dans  $[1,23]$ )

Le nombre flottant associé à ce binaire est donné par

$$(-1)^s 2^{e-127} \left( 1 + \sum_{i=1}^{i=23} b_i 2^{-i} \right)$$

# Nombres flottants

Soit le nombre binaire: 01000110 00111111 11001100 00110000



$$s=0, e=140$$

$$\text{nombre} = (-1)^s 2^{e-127} \left( 1 + \sum_{i=1}^{23} b_i 2^{-i} \right)$$

$$= 2^{140-127} (1+2^{-2}+2^{-3}+2^{-4}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-18}+2^{-19})$$

$$= 12275.046875$$

# Nombres flottants: stockage

Le nombre 12275.046875

Est représenté en binaire en 'float' simple précision par  
01000110 00111111 11001100 00110000

Qui vaut en hexa: 46 3F CC 30

Sur une architecture Little Endian, ce nombre sera stocké  
sous la forme:

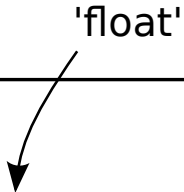
30 CC 3F 46

# Flottants: visualisation Python

```
import binascii
import struct

stockage_hexa=binascii.hexlify(struct.pack('f',12275.046875))
print(stockage_hexa)

valeur_decimale=struct.unpack('f',binascii.unhexlify(b'30CC3F46'))
print(valeur_decimale)
```



# Nombres flottants

## Remarques:

On ne peut pas représenter tous les nombres exactement

ex. 0.4 ne se représente pas de manière exacte en flottant

```
binascii.hexlify(struct.pack('f',0.4))
```

```
> CDCCCC3E
```

Correspond au nombre binaire:

```
00111110 11001100 11001100 11001101
```

qui vaut:  $2^{125-127} (1+2^{-1}+2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+2^{-16}+2^{-17}+2^{-20}+2^{-21}+2^{-23})$

~ 0.400000005960465

# Double précision

'Double' est similaire aux 'float' avec plus de précision

Un double est stocké sur 8 octets

1 bit de signe

11 bits pour l'exposant  $e$

52 bits de mantisse

$$N = (-1)^s 2^{e-1023} \left( 1 + \sum_{i=1}^{i=52} b_i 2^{-i} \right)$$

# Double précision: Python

'double' précision

```
stockage_hexa=binascii.hexlify(struct.pack('d',0.4))  
print(stockage_hexa)  
  
valeur_decimale=struct.unpack('d',binascii.unhexlify(b'9A9999999999D93F'))  
print(valeur_decimale)
```

# Double précision: approximation

Rem. 0.4 n'est toujours pas représentable exactement en double précision.

0.4 est stocké comme: 9A 99 99 99 99 99 D9 3F

Soit, le nombre binaire:

```
00111111 11011001 10011001 10011001  
10011001 10011001 10011001 10011010
```

Qui correspond au nombre décimale:

0.4000000000000000002220446...



# Nombres flottants

Pour aller plus loin:

- <http://floating-point-gui.de/>
- *[David Goldberg, What Every Programmer Should Know About Floating-Point Arithmetic]*