Entiers

integer, int, <class 'int'>

In [3]:
```
197867654730978639873989999999999999999999999999 +1
```

Out[3]:
```
197867654730978639873990000000000000000000000000
```

Littéraux

In [5]:
```
0, -1, 12896, 19786765473097863987398999999999, 90, "Bonjour"
```

Out[5]:
```
(0, -1, 12896, 19786765473097863987398999999999, 90, 'Bonjour')
```

Opérateurs

4 opérations arithmétiques

In [34]:
```
1+2, 8-5, 5-9, 12*111, 12/100, 100/10
```

Out[34]:
```
(3, 3, -4, 1332, 0.12, 10.0)
```

Priorités des opérateurs

In [3]:
```
1+2*3, (1+2)*3, 1+(2*3)
```

Out[3]:
```
(7, 9, 7)
```

Division entière et modulo

Diviser a par b, trouver q, r, entiers, tels a = b * q + r

q s'appelle le quotient

r s'appelle le reste

quotient `a // b`

reste `a % b` encore appelé "modulo"

In [4]:
```
27/5, 27//5, 27%5
```

Out[4]:
```
(5.4, 5, 2)
```

Elévation à la puissance

In [5]:
```
2**3, -2**3, (-2)**3, -2+5**3
```

Out[5]:
```
(8, -8, -8, 123)
```

L'élévation à la puissance est prioritaire sur les multiplications/divisions, elles-mêmes prioritaires sur les additions/soustractions.

In [6]:
```
2**3**4, (2**3)**4,   2**(3**4)
```

Out[6]:
```
(241785163922925834941235 2, 4096, 241785163922925834941235 2)
```

L'élévation à la puissance est associative *à droite*

Représentation polynômiale des nombres entiers

En base B, un nombre entier peut être mis sous la forme d'un polynome en B

$$\sum_{i=0}^{n} a_i B^i$$

Avec $0 \leq a_i < B$

Il s'écrit alors

$$a_n \dots a_0$$

Par exemple 123 en base 10 ($B = 10$)

$$1 * 10^2 + 2 * 10^1 + 3 * 10^0$$

Pour l'informatique, les bases 2 (binaire), 8 (octal), 16 (hexadecimal) sont importantes.

In [6]:
```
0b1010, 10,0o12, 0xA # différentes écritures de 10 en base 2, 8, 10, 16
```

Out[6]:
```
(10, 10, 10, 10)
```

In [7]:
```
0x123
```

Out[7]:
```
291
```

Représentation naturelle des entiers naturels

La technologie fournit des dispositifs bistables (2 états, notés 0 et 1) par lesquels on représente les chiffres de la base 2 (bits). Les nombres se représentent donc en base 2, par une suite de bits. Pour des raisons technologiques, on regroupe les bits par paquets de 8 (2^3). On parle d'octet (byte). On représente des nombres de plus en plus grands sur des mots mémoires d'un octet, 2 octets, 4 octets, 8 octets.

In [7]:
```
2**8, 2**16, 2**32, 2**64
```

Out[7]:
```
(256, 65536, 4294967296, 18446744073709551616)
```

Selon la longueur n d'un mot mémoire, exprimée en nombre de bits, on peut ainsi représenter les nombres de 0 à $2^n - 1$.

Au delà de cette limite, la capacité d'un mot mémoire est dépassée, Python3 s'adapte et change de représentation. Un tableau est utilisé pour représenter une suite de chiffres arbitrairement longue.

Cette représentation permet les opérations sur les entiers. Ainsi, l'addition de 2 bits correspond à un OU EXCLUSIF et donne lieu à une retenue qui correspond au ET des 2 bits. L'addition de deux nombres représentées sur n bits s'effectue en additionnant les bits de positions correspondantes, en commençant par ceux de poids faibles et en reportant les retenues éventuelles sur le rang suivant.

Représentation des entiers relatifs

Pour distinguer les nombres entiers positifs et négatifs, on réserve un bit pour le signe. En général, 0 pour + et 1 pour -.

On peut alors représenter les nombres de $-2^{n-1} - 1$ jusqu'à $2^{n-1} - 1$.

Cette représentation naturelle a deux inconvénients.

1) Il y a deux représentations pour 0 (+0 et -0). Cela complique la comparaison de deux nombres et cela fait perdre une configuration utile.

2) L'algorithme de l'addition donne un résultat erroné lorsque l'un des deux nombres au moins est négatif. Pour $3 - 5$, il donne -8 au lieu de -2.

$00000011+$

$10000101 =$

10001000

Pour corriger ces inconvénients, on utilise la représentation des nombres négatifs par leur complément à 2.

Pour construire le complément à 2, on construit tout d'abord le complément à 1. Pour cela, on inverse chaque bit sauf le bit de signe : remplacer les 0 par des 1 et les 1 par des 0. Par exemple pour -5 on a :

11111010

Pour construire le complément à 2, on ajoute 1 au complément à 1. Pour -5, cela donne :

11111011

L'algorithme naturel de l'addition donne alors un résultat différent.

$00000011+$

$11111011 =$

11111110

On reconnaît un nombre négatif à son bit de signe. On retire 1 pour avoir le complément à 1.

11111101

On inverse tous les bits sauf le bit de signe pour avoir l'écriture naturelle du nombre relatif

10000010

On lit un résultat correct, -2.

La représentation des nombres négatifs par leur complément à 2 permet de représenter sur n bits les nombres allant de

-2^{n-1} jusqu'à $+2^{n-1}-1$.

Toutes les configurations sont utilisées. Chaque nombre à une représentation distincte. L'algorithme naturel de l'addition peut être utilisé.

In [8]:
```
-2**2
```

Out[8]:
```
-4
```

In [2]:
```
(-2)**6
```

Out[2]:
```
64
```

In [6]:
```
type(-1)
```

Out[6]:
```
int
```

In [7]:
```
type(3.14)
```

Out[7]:
```
float
```

In [12]:
```
int(3.14), int("315"), int(3.81), round(3.81)
```

Out[12]:
```
(3, 315, 3, 4)
```

In [13]:
```
int(-3.14)
```

Out[13]:
```
-3
```

In [14]:
```
import math
```

In [15]:
```
math.trunc
```

Out[15]:
```
<function math.trunc(x, /)>
```

In [16]:
```
0xA, 0xB, 0xC, 0xD, 0xE, 0xF
```

Out[16]:
```
(10, 11, 12, 13, 14, 15)
```

In [17]:
```
type(math.trunc(-3.14))
```

Out[17]:
```
int
```

In [ ]:
```

```

In [18]:
```
0o77, 0o0, 0o1, 0o2, 0o3, 0o4, 0o5, 0o6, 0o7, 0o10
```

Out[18]:
```
(63, 0, 1, 2, 3, 4, 5, 6, 7, 8)
```

In [19]:
```
0b111, 0b0, 0b1, 0b10, 0b11, 0b100, 0b101, 0b110
```

Out[19]:
```
(7, 0, 1, 2, 3, 4, 5, 6)
```

Afficher un nombre au format binaire(selon https://stackoverflow.com/questions/699866/python-int-to-binary-string)

In [20]:
```
"{0:b}".format(63)
```

Out[20]:
```
'111111'
```

In [21]:
```
"{0:b}".format(-63)
```

Out[21]:
```
'-111111'
```

In [22]:
```
"{0:b}".format(-1)
```

Out[22]:
```
'-1'
```

In [23]:
```
bin(63), bin(-63), bin(-1)
```

Out[23]:
```
('0b111111', '-0b111111', '-0b1')
```

In [24]:
```
f"{63:b}", f"{-63:b}", f"{-1:b}"
```

Out[24]:
```
('111111', '-111111', '-1')
```

Pour avoir le bit de signe et la représentation en code complément à 2, il faut spécifier le nombre de bits sur lequel l'entier est représenté.

'{:0 compléter avec des 0 à gauche b écrire en binaire %i une valeur entière qui sera substituée par nbits

In [25]:
```
def int2bin(nbits,x):
    if x < 0:
    #
    #  on calcule le complément à 2
    #
        x = ( 1<<nbits ) + x
    #
    #  on complète l'écriture binaire par des 0 à gauche pour avoir nbits e
    #
    formatstring = '{:0%ib}' % nbits
    return formatstring.format(x)
```

In [26]:
```
int2bin(8,63), int2bin(8,-63), int2bin(8,-1), int2bin(8, -1<<1)
```

Out[26]:
```
('00111111', '11000001', '11111111', '11111110')
```

In [27]:
```
-1<<1
```

Out[27]:
```
-2
```

In [28]:
```
0xFF
```

Out[28]:
```
255
```

In [29]:
```
bin(-5)
```

Out[29]:
```
'-0b101'
```

In [ ]:
```

```