CODAGE ET TYPAGE

1. À l'intérieur d'un microprocesseur…

Dans un ordinateur, il n'y a que des 0 et des 1… si l'on peut dire, puisqu'en fait il n'y a que du courant électrique, qui passe plus ou moins bien dans les circuits ! Quand il passe bien, on dit que c'est 1, quand il passe mal ou pas, on dit que c'est 0. On aurait pu dire blanc et noir tout aussi bien, c'est un choix, pas tout à fait arbitraire cependant comme on va le voir ci-dessous.
On doit donc trouver une correspondance, un code (pas secret !), pour faire correspondre ces chiffres aux données que l'on manipule.
Puisque l'ordinateur travaille avec des 0 et des 1, on utilise naturellement la base 2, qui n'utilise que les chiffres 0 et 1 pour représenter les nombres :

Base 10	0	1	2	3	4	5	6	7	8
Base 2	0	1	10	11	100	101	110	111	1000

La valeur 0 ou 1 s'appelle un *bit* (binary digit). Ces valeurs sont groupées suivant des puissances de 2, en effet là aussi c'est plus simple pour l'ordinateur de gérer les circuits suivants des puissances de 2. 8 bits forment un *octet*. Un byte est souvent un octet, mais pas toujours… je vous déconseille d'utiliser ce terme tant que vous n'en aurez pas la définition exacte (qui dépasse le cadre de l'ISN). Un *mot* est l'unité de base manipulée par un microprocesseur. Les ordinateurs actuels utilisent des mots de 8, 16, 32 ou 64 bits (respectivement appelés octet, mot, double mot, quadruple mot). Vous pouvez constater que « mot » est mal défini…

2. Les types et leur représentation.

Les données sont d'un certain type : nombre, caractère, texte, image… Les types de données rencontrés dans les différents langages de programmation se recoupent en grande partie. Les données sont toutes traduites en 0 et 1 dans la machine, mais le codage associé à un type permet de « comprendre » la donnée. Avant de voir comment sont codés les différents types en machine, il est nécessaire de faire un petit crochet par les mathématiques.

3. Bases de numération.

a. La base 10 (numération décimale).

Le principe de la numération positionnelle est d'exprimer un nombre en fonction de symboles et de puissances de la base de numération, ce qui sera plus clair avec un exemple en base 10 :

$$4502 = \underbrace{4}_{\text{symbole}} \times \underbrace{10^3}_{\text{base au cube}} + \underbrace{5}_{\text{symbole}} \times \underbrace{10^2}_{\text{base au carré}} + \underbrace{0}_{\text{symbole}} \times \underbrace{10}_{\text{base puissance 1}} + \underbrace{2}_{\text{symbole}} \times \underbrace{10^0}_{\text{puissance 0=1}}$$

En base 10, les symboles sont les chiffres de 0 à 9.
Ce système permet de faire des calculs plus simplement qu'avec d'autres systèmes, comme par exemple le système romain :

$$\text{Comparez} \quad \begin{array}{r} 3\ 8 \\ \times\ 5\ 0 \\ \hline \end{array} \quad \text{et} \quad \begin{array}{r} X\ X\ X\ V\ I\ I\ I \\ \times \qquad\qquad\qquad L \\ \hline \end{array}$$

b. La base 2 (numération binaire).

En base 2, on ne dispose que des chiffres 0 et 1, et des puissances de 2.

Exemple : $\overline{6}_{10} = \overline{110}_2 = 1\times 2^2 + 1\times 2 + 0$.

Notations : dans l'exemple précédent, $\overline{6}_{10}$ précise que le nombre est écrit en base 10 et $\overline{110}_2$ précise que le nombre est écrit en base 2. On utilise aussi 6_{10} et 110_2. On regroupe souvent les chiffres par groupes de 4 pour plus de lisibilité, et également pour la traduction en base 16 (cf. ci-dessous).

Méthodes :

- *Du binaire au décimal* : on écrit tout simplement la somme de produits, sans les 0.
  $\overline{11010101}_2 = 1\times 2^7 + 1\times 2^6 + 1\times 2^4 + 1\times 2^3 + 1 = 216$

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

- *Du décimal au binaire* : on fait une succession de divisions euclidiennes par 2, en notant le quotient dans une colonne, et le reste dans une autre, à chaque étape :
  Exemple avec 111

division par 2 et quotient	reste	Suite des divisions euclidiennes
111	1	$111 = 55\times 2 + 1$
55	1	$55 = 27\times 2 + 1$
27	1	$27 = 13\times 2 + 1$
13	1	$13 = 6\times 2 + 1$
6	0	$6 = 3\times 2 + 0$
3	1	$3 = 2\times 1 + 1$
1	1	on arrête quand le quotient vaut 0 : $1 = \boxed{0}\times 1 + 1$

puis on reprend tous les restes dans l'ordre inverse : d'où le résultat : $\overline{111}_{10} = \overline{1101111}_{2}$

c. <u>La base 16 (numération hexadécimale).</u>

Dans la base 16, les symboles/chiffres vont de 0 à 15. Or 15 n'est pas un chiffre mais un nombre. On utilise donc les chiffres de 0 à 9, puis A pour 10, B pour 11, etc. jusqu'à F pour 15 (en majuscules souvent, parfois en minuscules).

L'avantage de la base 16 est qu'elle permet de « compacter » l'écriture binaire. En effet, puisque $2^4 = 16$, on peut regrouper par 4 les chiffres en base 2 (les « bits » par abus de langage) pour obtenir un chiffre en base 2

Base 2	0000	0001	0010	0011	0100	0101	0110	0111
Base 16	0	1	2	3	4	5	6	7

Base 2	1000	1001	1010	1011	1100	1101	1110	1111
Base 16	8	9	A	B	C	D	E	F

Exemple :

Les codes couleur, pour le web par exemple, sont souvent donnés sous l'intensité des trois couleurs additives de base (Rouge Vert Bleu pour un écran d'ordinateur, à l'inverse des couleurs soustractives Magenta Cyan Jaune). Chaque intensité de couleur est codée de 0 à 255, soit de 0 à FF en hexadécimal.

Le violet-rouge moyen a comme code RVB #C71585, qui est plus lisible que :

$$11100011100010101100000101 \text{ soit } \underbrace{11000111}_{\text{rouge}}\underbrace{00010101}_{\text{vert}}\underbrace{10000101}_{\text{bleu}}$$

Par ailleurs, les ordres de grandeur peuvent se deviner facilement ; dans ce violet-rouge moyen il y a :

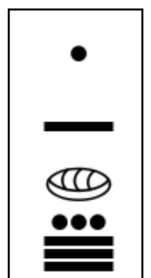
- C7 rouge : c'est beaucoup (C est un grand chiffre)
- 15 vert : c'est peu (1 est un petit chiffre)
- 85 bleu : c'est moyen (8 c'est la moitié de 16, soit la moitié de la base)

d. <u>Autres bases.</u>

Les principes exposés ci-dessus peuvent s'appliquer pour n'importe quelle base. En informatique, on n'utilise que les bases 2, 10 et 16. Dans la vie courante, il nous reste des traces du tout premier système de numération né avec l'écriture, à savoir le système sexagésimal utilisé par les Sumériens puis les Babyloniens (aux IIIe et IIe millénaire av. J.C.). Cette numération, dite mésopotamienne, utilise la base 60…elle nous est restée pour les heures et les degrés, mais est aussi présente en Chine depuis -1191, et en Inde depuis -3102.

D'autres systèmes utilisent plusieurs bases, ainsi la numération maya mélange base 60 et base 20 (numération vigésimale) :

$$9018 = 1\times\underbrace{20\times 360}_{\text{base}} + 5\times\underbrace{360}_{\text{base}} + 0\times\underbrace{20}_{\text{base}} + 18 = \underbrace{\overline{150\nabla}_{maya}}_{\nabla\text{ : symbole inventé pour 18}}$$, soit ceci :



Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

4. Les booléens.

Un booléen est en théorie un bit : Vrai ou Faux. On utilise généralement le codage 0 pour faux et 1 pour vrai, comme par exemple en logique ou en calcul booléen (que l'on verra dans quelques séances).
Typage : En Python, le type booléen « bool » est en fait un entier, qui vaut False pour 0 et True dans tous les autres cas. Ce qui peut donner des résultats surprenants.

5. Représentation des entiers en machine.

Ils sont traduits en base 2, sur 32 bits au moins. La valeur maximale n'est pas $2^{32}-1=4294967295$, comme on pourrait le croire. En effet il faut aussi pouvoir représenter les entiers négatifs (cf. paragraphes suivants). Le codage interne à l'ordinateur est classiquement sur un mot de 4 octets, pour un entier de 32 bits. En Python, les entiers sont aussi grands que l'on veut : si nécessaire, Python prend le nombre de mots nécessaires pour coder l'entier.
Typage : type plain integer, « `int` ». On trouve parfois dans des vieux programmes des entiers « longs » de type long integer, « `long` ».
Remarque : en Python, les entiers sont aussi longs que l'on veut

6. Le problème de la représentation des entiers négatifs.

On représente un entier relatif par un entier naturel… avec la convention que l'on expose ci-après, dite du complément à deux.
Supposons que l'on travaille avec des mots de 16 bits. Si l'on ne représente que des entiers positifs, leur valeur ira de 0 à $\overline{1111111111111111}^2=2^{16}-1=65535$. Si l'on souhaite représenter des entiers relatifs, on gardera le premier bit pour donner le signe. Il restera 15 bits pour la valeur absolue de l'entier. On pourra donc représenter les entiers de -32768 à 32767.

a. *Remarque préliminaire* : Une méthode inefficace pour commencer
Représentons le premier bit par « + » ou « - », plutôt que par 0 ou 1, puisque d'une part nous voulons des entiers signés, et d'autre part la représentation des bits par des symboles est arbitraire. L'idée « naturelle » est alors de poser $\overline{+111111111111111}^2=2^{15}-1=32767$ et $\overline{-111111111111111}^2=-\left(2^{15}-1\right)=-32767$. Cette méthode a plusieurs inconvénients, le plus visible étant les deux écritures de 0 : $\overline{0000000000000000}^2$ et $\overline{1000000000000000}^2$, où le premier 1 (respectivement 0) représente le + ou le -. Par ailleurs, essayez d'ajouter un nombre positif et un nombre négatif avec ce codage : le résultat est surprenant…**conclusion : se contenter de mettre 0 pour + et 1 pour - n'est pas utilisable en pratique !** On utilise la méthode du complément à deux.

b. Principe de la notation en complément à deux.

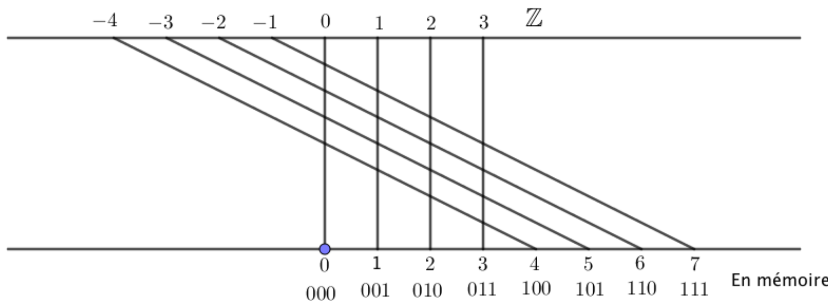
On représente les entiers positifs de manières standard

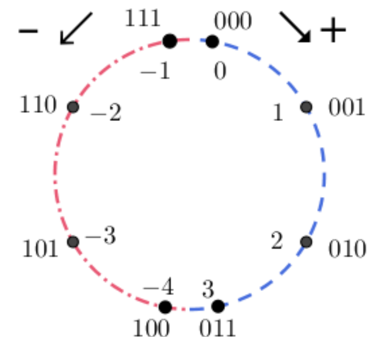
- 0 est représenté par $\overline{0000000000000000}^2$;
- 1 par $\overline{0000000000000001}^2$;
- etc jusqu'à $\overline{0111111111111111}^2=2^{15}-1=32767$
- Quand on ajoute 1, on passe alors à $\overline{10000000000000000}^2=2^{15}=32768$ si l'on considère des entiers positifs. Dans le cas d'un entier négatif, puisque le premier bit représente le signe « - », on considérera en fait que $\overline{10000000000000000}^2=-2^{15}=-32768$.
- Puis le suivant : $\overline{10000000000000001}^2=-\left(2^{15}-1\right)=-32767$
- Ainsi de suite jusqu'à $\overline{1111111111111111}^2=-1$

En résumé, un entier positif n est représenté par lui-même. Un entier négatif n est représenté par $n+2^N$, (avec des mots de N bits). On retient la formule $codage=n+2^N \Leftrightarrow n=codage-2^N$, ou encore $codage=2^N-|n|$; ne pas oublier que n est négatif !

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

Exemples : sur un mot de 3 bits, on représente les entiers positifs de 0 à $2^3-1=7$. Si on veut également représenter des entiers négatifs en complément à deux, on peut alors écrire les entiers de -4 à 3. Le premier schéma ci-dessous donne cette représentation en complément à deux pour des mots de 3 bits. Sur la première ligne, on a les nombres à représenter, compris entre -4 et 3. Sur la deuxième ligne figure leur codage, entre 0 et 7. Sur le deuxième schéma, on numérote en binaire de 000 à 111 dans le sens des aiguilles d'une montre. Puis on écrit les entiers par demi-cercle, dans le sens des aiguilles d'une montre vers les positifs, et dans le sens trigonométrique vers les négatifs.





c. <u>Compéments.</u>

- Le bit de poids fort est celui le plus à gauche, celui de poids faible celui le plus à droite. 1000 0000 a pour bit de poids fort 1, et 1111 1110 a pour bit de poids faible 0.
- En python :
    - Un nombre binaire s'écrit avec `0b` devant, et `bin(nombre)` permet d'obtenir la conversion en binaire.
    - ```
      >>> bin(199)
      '0b11000111'
      ```
    - Un nombre hexadécimal s'écrit avec `0x` devant, et `hex(nombre)` permet d'obtenir la conversion en hexadécimal.
    - ```
      >>> hex(199)
      '0xc7'
      ```
    - L'affichage par défaut d'une variable se fait en décimal. Même si son contneu résulte d'un calcul en binaire ou en hexadécimal, il faudra forcer l'affichage avec une des fonctions précédentes si on veut le voir apparaître tel quel.
- Le petit et le gros boutisme viennent d'une guerre de religion inventée par Jonathan Swift dans « Les Voyages de Gulliver ». Il s'agissait de savoir s'il faut manger les œufs par le gros bout ou par le petit bout.

  En informatique, il s'agit de savoir comment on écrit les nombres en mémoire : est-ce que l'on met les bits dans l'ordre « normal » ou dans l'ordre inverse. L'ordre usuel permet une lecture par les humains plus facile, tandis que l'ordre inverse permet des calculs plus simples par le microprocesseur. Ainsi un mot de 16 bits comme AB12 sera écrit en machine [AB] [12] en gros boutisme et [12] [AB] en petit boutisme.

7. <u>Les nombres réels.</u>

On ne peut représenter que des valeurs décimales, vu qu'on ne dispose que de mots de longueur finie. $\sqrt{2}$ et π ne peuvent pas être représentés exactement.

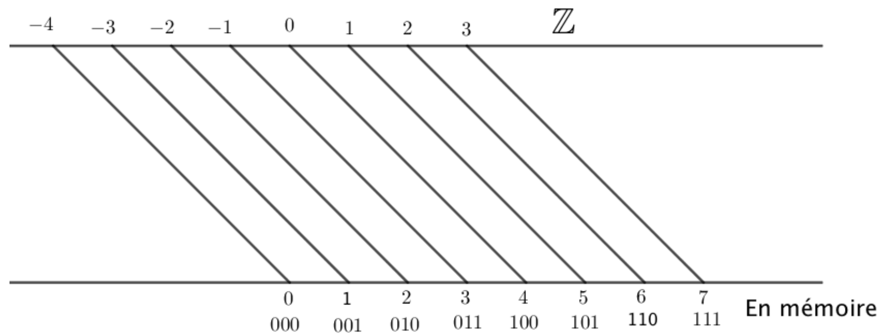
Les réels sont codés sous la forme de la représentation scientifique : $x=(-1)^s\times m\times b^e$ dite mantisse-exposant.

Par exemple $-100\pi\simeq(-1)^1\times 3{,}14159\times 10^2$ en décimal.

En informatique, cette écriture est d'abord convertie en binaire, et on l'écrira sous la forme $x=(-1)^s\times m\times 2^e$

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

Pour un mot de 64 bits, en norme IEEE 754, on aura 1 bit pour le signe (0 pour + et 1 pour -), 11 bits pour l'exposant et 52 bits pour la mantisse. Comme la mantisse, en base 2, commence forcément par un 1, on l'omet et on garde les 52 bits pour les chiffres après la virgule.
L'exposant est compris entre -1022 et 1023, on le représente comme l'entier naturel $n+1023$, compris entre 1 et 2046. On réserve 0 et 2047 pour les situations exceptionnelles (+∞ , -∞, NaN : Not a Number pour les divisions par 0 entre autres, cf. ci-dessous). Cette représentation, différente de celle des entiers relatifs, est appelée représentation biaisée.



Représentation biaisée pour des exposants sur 3 bits.

Norme (IEEE 754) en simple précision.
Pour un mot de 32 bits, on aura 1 bit pour le signe (0 pour + et 1 pour -), 8 bits pour l'exposants et 23 bits pour la mantisse.
Comme la mantisse, en base 2, commence forcément par un 1, on l'omet et on garde les 23 bits pour les chiffres après la virgule.
Les exposants 00000000 et 11111111 sont réservés pour les situations exceptionnelles (+∞ , -∞, NaN : Not a Number).
Les exposants vont donc de 00000001 à 11111110. Il sont représentés avec un décalage de $127=2^7-1$. L'exposant est compris entre -126 et 127 ; 00000001 représente $1-127=-126$ et 11111110 représente 127.
Remarques :

- la norme IEEE 754 n'est pas à connaître par cœur, du moins en ce qui concerne le nombre de bits pour mantisse et exposant. Les procédés de calcul le sont par contre.
- 0 n'est pas représentable avec les conventions exposées ci-dessus ! En effet, les nombres les plus proches de 0 que l'on puisse obtenir sont $\pm 2^{-126}$ (en 32 bits). Par convention, ces valeurs, correspondant à un codage de la mantisse et de l'exposant sont 0. Il y a donc un 0 positif et un 0 négatif suivant le bit de signe.
- De même on représente +∞ et -∞ par un codage de mantisse 0 et d'exposant 255 (toujours en 32 bits).
- `Nan` est un codage particulier, Not a Number, que l'on obtient par exemple lors d'une divisons par 0.
- Enfin, signalons l'existence des nombres dénormalisés qui permettent de représenter les nombres très proches de 0, et d'éviter que les tests `x – y == 0` et `x == y` donnent des résultats différents ! L'étude de ces nombres n'est pas au programme de 1ère. Ce n'est pas très compliqué, vous trouverez facilement des informations sur le web si vous souhaitez vous pencher sur le problème.

Exemple : Codage de $100\pi \simeq 314{,}15625$ sous la forme d'un flottant en simple précision.
Typage : type « `float` » en Python. La notation scientifique décimale en Python s'écrit avec le symbole `e`, par exemple $1{,}235\times10^{18}$ s'écrit `1.235e18`.
Remarque :On peut représenter les nombres complexes en Python (vus en terminale), c'est un des rares langages le permettant. On utilise j et non i, et on fait systématiquement précéder j d'un nombre, sinon il est considéré comme une variable. Par exemple, pour écrire $3+i$, on tapera `3 + 1j`.

8. Codage des caractères
Pour permettre les échanges d'informations textuelles entre systèmes informatiques, le codage **ASCII** (American Standard Code for Information Interchange) a été proposé en 1963 (avant c'était le bazar).

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

Chaque caractère est associé à un mot de 7 bits. Par exemple R est associé à $\overline{1010010}_2 = \overline{52}_{16}$ et le symbole 5 à $\overline{0110101}_2 = \overline{35}_{16}$. Par commodité, chaque caractère est en fait codé dans un octet. Sont codés en plus des caractères de contrôle, comme le « line feed », saut de ligne et « carriage return », retour chariot. Ces deux caractères font passer à la ligne, et donnent des comportements différents sous Windows ou sous Unix (système à la base de Linux et d'OsX). Sont également codés les chiffres, les signes de ponctuation, des opérations arithmétiques (+), un « beep », un caractère d'effacement…Les 32 premiers caractères en ASCII,codés de 0 à 1F ne sont pas imprimables. Le bit excédent (puisque la norme à l'époque était des mots de 8 bits et non de 7) est un bit de contrôle qui permettait d'éviter les erreurs de transmission. C'est le bit de poids fort, appelé bit de parité. Il vaut 0 ou 1 de manière à ce que le nombre de 1 dans l'octet soit toujours pair.
Comme l'ASCII ne permettait pas le codage des lettres accentuées, d'autres normes de codage sont apparues (Windows cp 1252, MacRoman, ISO 8859-15). Par ailleurs, la présence d'un bit « libre » dans les octets du codage ASCII permet de représenter plus de caractères. La majorité de ces normes intègrent l'ASCII, mais ne sont pas compatibles entre elles. Comme il n'y a pas moyen de savoir quel codage est utilisé pour un document, cela donne parfois des surprises à la lecture, avec des caractères étonnants.
La norme **ISO 8859** a utilisé le bit libre de la norme ASCII pour proposer des caractères supplémentaires, comme les caractères accentués, le *ß* allemand, etc… Les 128 premiers caractères sont ceux de l'ASCII, et les suivants dépendent…de la sous-norme ! Il y a en fait 16 tables différentes, par exemple la norme **ISO 8859-1**, dite latin-1, est pour l'Europe occidentale, la 8859-6 pour l'Arabe, etc…
Actuellement la norme qui tend à s'imposer, venant de Linux et du HTML, est l'**UTF-8**. Elle est présente par défaut sur les dernières versions de Windows et d'OsX. La principale différence par rapport aux normes précédentes est qu'un caractère est représenté par un mot de 1 à 4 octets, de longueur variable.
L'UTF-8 est une implémentation de la norme **Unicode,** qui peut contenir plus de 4 millions de caractères différents. Ceci permet de représenter les idéogrammes chinois, les caractères nordiques, cyrilliques, arabes, etc. Les 256 premiers caractères de l'UTF-8 sont ceux de la norme ISO 8859 (et donc les 128 premiers caractères sont ceux de l'ASCII). Chaque caractère est associé à un unique entier appelé **point de code**.
En UTF-8, si le bit de poids fort est 0, alors il s'agit du caractère ASCII correspondant. Sinon, les bits de poids fort indiquent le nombre d'octets pour coder le caractère, avec un nombre de 1 correspondant au nombre d'octets, suivi d'un 0. Les caractères sont codés sous la forme U+*xxxx* où *xxxx* est la valeur hexadécimale du code. Chaque octet a une valeur comprise entre 128 et 255, ils commencent tous par 10.
De manière simplifiée, on a :

Caractères codés	Représentation binaire en UTF-8	Signification
U+0000 à U+007F	0 *xxx xxxx*	1 octet codant 7 bits
U+0080 à U-07FF	110 *x xxxx* 10*xx xxxx*	2 octets codant 11 bits
U+0800 à U+FFFF	1110 *xxxx* 10*xx xxxx* 10*xx xxxx*	3 octets codant 16 bits
U+10000 à U+10FFFF	11110 *xxx* 10*xx xxxx* 10*xx xxxx* 10*xx xxxx*	4 octets codant 21 bits

Citons également la norme UTF-16, qui code un caractère sur un minimum de 16 bits, et la norme UTF-32, qui codent les caractères systématiquement sur 32 bits, ce qui simplifie le décodage mais perd beaucoup de place.
En Python :

- aussi bien les caractères isolés, que les chaînes de caractères sont de type « `str` » (string, qui veut aussi dire chaîne en anglais). D'autres langages distinguent les caractères isolés des chaînes.
- `"\uxxx"` donne un caractère Unicode, par exemple `"\u0052"` donne `'R'`.
- En Python, les chaînes de caractères sont traitées en partie comme des tableaux. Les fonctions `ord()` et `chr()` permettent de travailler en Python sur les caractères. Les caractères Unicode s'écrivent sous la forme `"\upoint_de_code_en_hexa"`.

  Exemple :
```
>>> a = « bonjour »
>>> premierCar = a[0]        # premierCar contient « b »
>>> ord(premierCar)          # renvoie 98, le code décimal ASCII de « b »
```


Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

`>>> a + « les petits »` # renvoie « bonjour les petits » ; c'est une opération de # concaténation de chaîne

`>>> a + chr(65)` # renvoie bonjourA, c'est à dire « bonjour » concaténée avec le # caractère dont le point de code décimal Unicode vaut 65 : A

`>>> "\u00e9"` # codage hexadécimal utf-8 Python du caractère « é »

- Pour obtenir l'encodage hexadécimal d'un caractère, il suffit de `faire hex(ord('é'))` qui renvoie `'0xe9'`.

Pour aller plus loin:

`'caractère'.encode('utf-8')` donne le point de code du caractère, sous forme d'une suite de bytes ; par exemple `'字'.encode('UTF-8')` renvoie `b'\xe5\xad\x97'`. A l'inverse, `b'\xc3\xa9'.decode('utf-8')` renvoie 'é'. *Remarque :* les bytes sont vus dans le cours architecture, ici ils correspondent à des octets. On n'obtient pas le codage hexadécimal utf-8, pour cela il faut faire : `'é'.encode('unicode_escape'))` qui renvoie `b'\\xe9'`, à comparer à `"\u00e9"`.

EXERCICES CODAGE

*On demande des calculs **sans `bin` et `hex` de Python** ! Vous pouvez utiliser Python comme outil de vérification.*

Exercices (entiers positifs)

1. Trouver la représentation en base 2 des nombres 1, 3, 7, 15, 31 et 63. Expliquer le résultat.
2. Trouver la représentation en base 16 des nombres 158 et 2045. Ecrire en décimal le nombre hexadécimal BEEF.
3. On donne les nombres suivants, qui sont soit en base 2, soit en base 16. S'ils sont en binaire les écrire en hexa et vice-versa.
   111011001010 E15A 1010100101 45
4. Trouver en base 10 la représentation du nombre $\overline{10010110}_2$ (la notation $\overline{xxxx}_2$ signifiant : nombre exprimé en base 2)
5. Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite. Quelle est l'opération équivalente en base 2 ? Le vérifier sur 3, 6 et 12.
6. Calculer en binaire la somme $\overline{1101101}+\overline{1001011}$. On fera comme en primaire, avec des retenues.
7. Comment peut-on multiplier un nombre en binaire par 2 ? Le diviser par 2 ? Que peut-il se passer dans ce dernier cas ?
8. Écrire en langage naturel un algorithme permettant d'ajouter deux entiers exprimés en binaire. L'implémenter en Python, en utilisant des tableaux de 0 et de 1 pour représenter les nombres en binaire.

Exercices (entiers relatifs ; codage sur 8 bits sauf mention contraire)

9. Quels nombres peut-on représenter avec des entiers de 8 bits ? De 32 ou 64 bits ?
10. Coder -23 et -78.
11. Trouver la représentation décimale des entiers relatifs dont la représentation en binaire est $\overline{00110010}$, $\overline{10000000}$ et $\overline{10110011}$.
12. Calculer la représentation binaire de 4, puis de -4, sur des mots de 8 bits. Vérifier que l'on obtient la représentation de -4 à partir de celle de 4 en inversant les 1 et les 0, puis en ajoutant 1.
13. Appliquer la méthode de l'exercice précédent pour trouver la représentation de -16. Vérifier que « ça marche ».
14. Représenter les entiers relatifs 96 et 48 sur 8 bits, et ajouter les résultats en base 2. Ce résultat est le codage d'un entier relatif sur 8 bits ; donnez-en la valeur décimale. Que remarquez-vous ? Expliquer.
15. Comment faire une soustraction à partir d'un part, de la méthode d'addition en binaire, et d'autre part, de la représentation des entiers relatifs ? Appliquer à 15 - 7 (sur des mots de 8 bits)
16. Écrire un programme Python qui donne le complément à deux d'un nombre binaire. En entrée on donnera une liste de longueur quelconque comportant les chiffres, et en sortie on aura une liste de même longueur comportant les chiffres du complément.

Exercices (représentation des décimaux ; entiers codés sur 8 bits et flottants sur 32 bits).

17. Trouver le décimal représenté par le mot : 0100 0011 0101 0100 1110 0000 0000 0000
18. Coder 141,42 sur 32 bits (c'est assez pénible XD, réfléchissez sur le nombre de bits de la mantisse et donc le nombre d'opérations à faire).

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/

19. Comment est représenté le nombre décimal 2^{-122} ?
20. Comment est représenté le nombre entier 7 ? Et le nombre décimal 7,0 ? Conclusion sur l'espace mémoire.
21. Comment est représenté le nombre 0,1 ? que remarquez-vous ?
22. Quelle précision perd-on si on divise à nombre à virgule par 2, puis qu'on le multiplie à nouveau par 2 ? (cet exercice a plusieurs réponses !)
23. Quelques tests sous Python ; expliquer les résultats obtenus (plusieurs calculs au a.)
    a. `1e75 + 275 - 1e75` ; `1e75*2 - 1e75` ; `(1.2 + 3.5) + 6.1 == 1.2 + (3.5 + 6.1)` `3.5*(1.2 + 1.1)` ; `3.5*(1.2 + 1.1)` ; `print("%.30f"%0.1)`
    b. `>>> x = 1e1000`
       `>>> y = x**x`
       `>>> z = x//x` (essayer avec `x = 1e175` suivant les machines/versions de Python)
24. Trouver le plus petit x flottant tel que `x == x + 1` renvoie `True`
25. Trouver le plus grand x flottant tel que `2.0**100 + x == 2.0**100` renvoie `True`
26. Résoudre « à la main » et dans $\mathbb{R}$ l'équation $\frac{1}{16}x^2+\frac{1}{10}x+\frac{1}{25}=0$. Sous Python, rentrer les trois coefficients en tapant `a = 1/4`, etc. , puis calculer Δ. Conclure.

Exercices (représentation des caractères et chaines)

27. Écrire un programme en Python qui, étant donné une chaîne de caractères, renvoie un tableau contenant les codes ASCII correspondants. Et un autre qui fait le contraire (pour tester ce dernier, n'utiliser que les codes de 65 à 90 –majuscules-, 97 à 122-minuscules-, 32 –espace-).
28. Modifier le programme précédent pour qu'il crypte un message, en décalant les lettres de 3 (code de César). On n'utilisera que des minuscules et des espaces. Les espaces devront rester identiques.
29. En Python, la norme retenue est UTF-8. Expliquer pourquoi il y a une contradiction apparente avec que l'on peut constater ci dessous.

	Caractère 1	Caractère 2
Dans Windows	Alt +265E	Alt +1F996
UTF-8 (hex)	0xE2 0x99 0x9E (e2999e)	0xF0 0x9F 0xA6 0x96 (f09fa696)
UTF-8 (binary)	11100010:10011001:10011110	11110000:10011111:10100110:10010110
UTF-16 (hex)	0x265E (265e)	0xD83E 0xDD96 (d83edd96)
UTF-16 (decimal)	9 822	55 358 56 726
UTF-32 (hex)	0x0000265E (265e)	0x0001F996 (1f996)
UTF-32 (decimal)	9 822	129 430
Python source code	u"\u265E"	u"\U0001F996"

30.

a. Qui sont les deux personnages suivants, dont les noms sont donnés ci-dessous avec les caractères Unicode, dans une des langues dans lesquels ils sont célèbres ? Indice : l'un est un héros des films Marvel Avengers, l'autre est son papounet.

```
fiston =  "\u00DE"+"\u00F3"+"\u0072"
papounet = "\u00D3"+"\u00F0"+"\u0069"+"\u006E"+"\u006E"
```

Comparer avec `"\u00DE\u00F3\u0072"`, tester `"\u00DE\u00F3\u0072tue"`

b. La bibliothèque unicodedata permet d'avoir des informations sur les caractères unicode. Dans les quelques fonctions ci-dessous, *`chr`* peut être un caractère ou son point de code en format Python `"\uxxxx"`.

- `unicode.name(chr)` renvoie le nom du caractère
- `unicode.decomposition(chr)` renvoie la décomposition du caractère en tant que chaine de caractère.

Explorer ces deux fonctions pour trouver les points de code et les noms des caractères qui forment les caractères « é » (et « ȍ »…si vous arrivez à taper ce dernier…). Donner le nom des caractères dont les points de code décimaux sont 4031 et 9736. Et éventuellement 129430, ça peut planter. Penser à d'abord convertir les points de codes en hexadécimal, puis écrire `"\u_bon_nombre_de_0_hexadécimal"`.

Cours de Frédéric Mandon sous licence Creative Commons BY NC SA, https://creativecommons.org/licenses/by-nc-sa/3.0/fr/