NSI1

Spécialité numérique et sciences informatiques en classe de première

Lycée Rabelais
Saint Brieuc
2032-2024

Partie I

Représentation des données

Chapitre 1

Bases de numération

« Partons sur de bonnes bases. »

On note $\mathbb{N}$ l'ensemble des *entiers naturels* : $\mathbb{N} = \{0; 1; 2; \ldots\}$.

Nous avons l'habitude d'utiliser la base 10 pour représenter les entiers naturels, c'est-à-dire qu'on utilise 10 symboles, appelés *chiffres* pour les écrire : 0, 1, 2, ..., 9. Or il n'en a pas toujours été ainsi :

- au I$^{\text{er}}$ millénaire av. J.-C., les Babyloniens utilisaient la base soixante pour mesurer le temps et les angles ;
- durant le I$^{\text{er}}$ millénaire, les Mayas et les Aztèques se servaient de la base vingt (et d'ailleurs en France, 80 se lit « quatre-vingts ») ;
- entre le VII$^{\text{e}}$ et le XV$^{\text{e}}$ siècle, les astronomes arabes utilisaient la base cent cinquante pour élaborer des tables permettant de trouver la position d'un astre dans le ciel à un moment donné.

De nos jours, en Informatique, on utilise beaucoup la base deux, dite *binaire* et la base seize, appelée *hexadécimale*.
L'objectif de ce chapitre est de donner les méthodes permettant d'écrire un entier naturel dans une base donnée, plus précisément dans les bases 2, 10 et 16. Nous verrons également comment passer facilement du binaire à l'hexadécimal et vice-versa.

1. Écriture binaire d'un entier naturel

1.1. Pourquoi le binaire ?

Pour simplifier, disons qu'au niveau le plus « bas » d'un ordinateur, se trouvent des (millions de) transistors qui jouent chacun un rôle d'interrupteur. De multiples points de l'ordinateur peuvent alors être soumis à une tension (état 1) ou non (état 0). En considérant 2 de ces points, on voit que l'état de ce système peut être 00, 01, 10 ou 11. Cela fait 4 possibilités et le binaire est né !

1.2. Comprendre l'écriture en base 2

Puisqu'il n'y a que deux chiffres en binaire, compter est simple mais nécessite rapidement plus de chiffres qu'en base 10 :

Écriture décimale	0	1	2	3	4	5	6	7	8	...
Écriture binaire	0	1	10	11	100	101	110	111	1000	...

Notation

On écrira $(11)_{10}$ pour insister sur le fait qu'on parle du nombre 11 *en base 10*, et $(11)_2$ pour dire que c'est une écriture binaire.
Lorsque ce n'est pas précisé cela veut dire que l'écriture est en base 10.
Ainsi $(11)_2 = 3$, et de même, $(111)_2 = 7$.

Tout entier naturel admet une unique écriture décimale (c'est-à-dire en base 10), il en va de même en binaire :

Propriété : écriture binaire d'un entier naturel

Tout entier naturel possède une unique écriture en base 2, dite *écriture binaire*. Plus précisément, soit $n \in \mathbf{N}$, alors il existe un unique entier $k \in \mathbf{N}$ et $k+1$ nombres a_i, uniques et valant 0 ou 1 et tels que

$$n = a_0 2^0 + a_1 2^1 + \dots + a_k 2^k$$

ce qui s'écrit aussi

$$n = \sum_{i=0}^{k} a_i 2^i$$

Exemple

Lorsqu'on regarde le tableau précédent, on voit que $6 = (110)_2$.
Cela s'interprète ainsi :

Chiffre binaire	1	1	0
Valeur	2^2	2^1	2^0

et on obtient $6 = 0\times 2^0 + 1\times 2^1 + 1\times 2^2$.

Méthode 1 : passer de la base 2 à la base 10

Que vaut $(11101)_2$?

Chiffre binaire	1	1	1	0	1
Valeur	2^4	2^3	2^2	2^1	2^0

$$\begin{aligned}(11101)_2 &= 1\times 2^4 + 1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0\\ &= 16+8+4+1\\ &= 29\end{aligned}$$

Méthode 2 : passer de la base 10 à la base 2

Comprenons ce que veut dire une écriture décimale :

$$\begin{aligned}203 &= 200+3\\ &= 2\times 10^2 + 0\times 10^1 + 3\times 10^0\end{aligned}$$

Faisons la même chose en base 2 :

$$\begin{aligned}203 &= 128+64+8+2+1\\ &= 2^7+2^6+2^3+2^1+2^0\\ &= 1\times 2^7 + 1\times 2^6 + 0\times 2^5 + 0\times 2^4 + 1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 1\times 2^0\\ &= (11001011)_2\end{aligned}$$

Cette méthode est pratique quand l'entier est petit et que l'on connaît bien les premières puissances de deux.
Quand ce n'est pas le cas, une autre méthode (la méthode 3) peut être employée.

Évidemment, PYTHON connait le binaire. Voici comment écrire un `int` en binaire et comment obtenir l'écriture binaire d'un `int`.

Python

```
>>> 0b11001011 # faire précéder le nombre de 0b
203
>>> bin(29)
'0b11101'
```

1.3. Un algorithme pour déterminer l'écriture binaire d'un entier naturel

Méthode 3 : les divisions successives

Voici comment on trouve les chiffres de l'écriture *décimale* de 203 :

On divise 203 par 10, cela fait 20, il reste 3, c'est le chiffre des unités.
On recommence avec 20, on le divise par 10, cela fait 2, reste 0, chiffre des dizaines.
On continue, on divise 2 par 10, cela fait 0, reste 2, chiffre des des centaines.
Puisqu'on a trouvé un quotient de 0, on s'arrête.
On peut écrire cela simplement :

$$\begin{array}{r|r|r|l}
203 & 10 & & \\ \cline{2-2}
\mathbf{3} & 20 & 10 & \\ \cline{3-3}
 & \mathbf{0} & 2 & 10 \\ \cline{4-4}
 & & \mathbf{2} & 0
\end{array}$$

Voici maintenant comment on trouve son écriture binaire. On procède comme en base 10 mais en divisant par 2 :

$$\begin{array}{r|r|r|r|r|r|r|r|l}
203 & 2 & & & & & & & \\ \cline{2-2}
\mathbf{1} & 101 & 2 & & & & & & \\ \cline{3-3}
 & \mathbf{1} & 50 & 2 & & & & & \\ \cline{4-4}
 & & \mathbf{0} & 25 & 2 & & & & \\ \cline{5-5}
 & & & \mathbf{1} & 12 & 2 & & & \\ \cline{6-6}
 & & & & \mathbf{0} & 6 & 2 & & \\ \cline{7-7}
 & & & & & \mathbf{0} & 3 & 2 & \\ \cline{8-8}
 & & & & & & \mathbf{1} & 1 & 2 \\ \cline{9-9}
 & & & & & & & \mathbf{1} & 0
\end{array}$$

On a donc successivement établi :

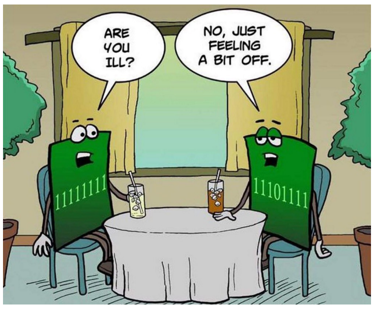
$203 = 101 \times 2 + 1$

$$
\begin{aligned}
&= (50 \times 2 + 1) \times 2 + 1 \\
&= ((25 \times 2 + 0) \times 2 + 1) \times 2 + 1 \\
&= (((12 \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 1 \\
&= ((((6 \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 1 \\
&= (((((3 \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 1 \\
&= ((((((1 \times 2 + 1) \times 2 + 0) \times 2 + 0) \times 2 + 1) \times 2 + 0) \times 2 + 1) \times 2 + 1 \\
&= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\
&= (11001011)_2
\end{aligned}
$$

Cette succession d'égalités n'est (heureusement) pas à écrire à chaque fois.

Les trois méthodes précédentes se programment facilement et la dernière est de loin la plus courte à écrire.

1.4. Vocabulaire



Un chiffre décimal peut être 0, 1, 2, 3, 4, 5, 6, 7, 8 ou 9.
Un chiffre binaire peut être seulement 0 ou 1. En Anglais, *chiffre binaire* se traduit par *binary digit*, que l'on abrège en *bit*. On garde cette dénomination en Français.
Le bit est « le plus petit morceau d'information numérique ».

Pour les écrire, on regroupe les chiffres décimaux par paquets de 3, comme dans 1 230 014 par exemple. En binaire on groupe les bits par 4, on écrira donc $17 = (1\ 0001)_2$.
La plupart du temps, en machine, les bits sont groupés par 8 (deux paquets de 4). Un tel paquet s'appelle un *octet*, et on écrit donc des *mots binaires* de longueur 8 tels que $0000\ 0011$: l'octet représente ici le nombre 3. Les bits à zéros ne sont pas inutiles.

Lorsqu'on considère un nombre écrit en binaire, on parle souvent de *bit de poids fort* et de *bit de poids faible* pour parler respectivement du bit associé à la plus grande puissance de 2, et du bit d'unités.
Considérons l'octet $(0010\ 0101)_2$. Son bit de poids fort est 0, son bit de poids faible est 1.

2. Écriture hexadécimale d'un entier naturel

La base « naturelle » de l'informatique est la base 2, mais elle n'est pas très pratique car elle donne lieu à des écritures trop longues. La base 10 nous paraît bien meilleure parce que nous avons l'habitude de l'utiliser, mais elle ne fait pas bon ménage avec la base 2 : il n'y a pas de méthode simple pour passer du décimal au binaire, et vice versa.
La base 16, ou base *hexadécimale*, est en revanche très adaptée à l'écriture des paquets de 4 bits, et par extension à celle des octets et autres écritures binaires.

En hexadécimal, on dispose de 16 chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F.

Propriété : écriture binaire d'un entier naturel

Tout entier naturel possède une unique écriture en base 16, dite *écriture hexadécimale*. Plus précisément, soit $n \in \mathbb{N}$, alors il existe un unique entier $k \in \mathbb{N}$ et $k+1$ nombres a_i, uniques et valant 0, 1, 2, ..., ou F et tels que

$$n = a_0 16^0 + a_1 6^1 + ... + a_k 16^k$$

ce qui s'écrit aussi

$$n = \sum_{i=0}^{k} a_i 16^i$$

Remarque

On a vu une propriété similaire en base 2 et en fait elle est valable *dans toutes les bases* b (où b est un entier naturel supérieur ou égal à 2). Cela justifie par exemple l'utilisation de la base 20 ou de la base 150.

Les méthodes que l'on a vu en base 2 et 10 se transposent en base 16.

Méthode 4 : passer de la base 16 à la base 10

Déterminons l'écriture décimale de $(D4A)_{16}$:

$(D4A)_{16} = 13 \times 16^2 + 4 \times 6 + 10 \times 16^0$ car D vaut 13 et A vaut 10.
$= 3402$

Méthode 5 : passer de la base 10 à la base 16

Déterminons maintenant l'écriture hexadécimale de 503 :

- $503 = 31 \times 16 + \underline{7}$

- $31 = 1 \times 16 + \underline{15}$ et 15 s'écrit $\underline{F}$.
- $1 = 0 \times 16 + \underline{1}$ et on arrête car le quotient est nul.
- $503 = (1F7)_{16}$.

Un éditeur hexadécimal montre le contenu d'un fichier, d'un disque dur, ou de la RAM d'un ordinateur. La première colonne indique l'adresse, puis 16 octets écrits en hexadécimal et enfin les caractères correspondants.

3. Hexadécimal et binaire : un mariage heureux

Le grand avantage qu'apporte l'hexadécimal s'illustre facilement :

Méthode 6 : passer de la base 2 à la base 16

$$\begin{aligned}(101101000011101)_2 &= (0101\ 1010\ 0001\ 1101)_2 \\ &= \left(\underbrace{0101}_{5}\ \underbrace{1010}_{A}\ \underbrace{0001}_{1}\ \underbrace{1101}_{D}\right)_2 \\ &= (5A1D)_{16}\end{aligned}$$

x	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E	20
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D	30
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C	40
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B	50
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A	60
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69	70
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	80
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87	90
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96	A0
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5	B0
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4	C0
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3	D0
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2	E0
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1	F0
10	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0	100

Table de multiplication hexadécimale.

Méthode 7 : passer de la base 16 à la base 2

$$(F7B)_{16} = \left(\underbrace{1111}_{F}\ \underbrace{0111}_{7}\ \underbrace{1101}_{B}\right)_2$$

4. Additions

On pose l'opération à la main : c'est la même chose qu'en base 10.

En base 2

La seule différence avec la base 10 c'est que deux 1 donnent $(2)_{10}$ donc $(10)_2$, donc un zéro et une retenue de 1. Quand il y a deux 1 et une retenue de 1 en plus, cela donne $(3)_{10}$ donc $(11)_2$, donc un 1 et une retenue de 1.

Exemple

$$
\begin{array}{ccccccccc}
 & & & {}^{1} & {}^{1} & {}^{1} & & & \\
 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
+ & & & & 1 & 1 & 1 & 0 & 0 \\
\hline
= & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1
\end{array}
$$

En base 16

C'est encore la même chose. Il faut bien se souvenir de la valeur de A, B, C, D, E et F.
Ajouter 8 et 3 ne provoque pas de retenue puisque $8 + 3 = 11$ et que 11 est B en base 16.
Dès que l'addition de 2 chiffres dépasse 15, il y aura une retenue : par exemple 9 et A donnent $(19)_{10}$, donc $(13)_{16}$. Ainsi on note 3 et une retenue de 1.

Exemple

$$
\begin{array}{ccccc}
 & & {}^{1} & & \\
 & 2 & 4 & 9 & 8 \\
+ & 1 & 7 & A & 3 \\
\hline
= & 3 & C & 3 & B
\end{array}
$$

5. Exercices

Exercice 1

Donner l'écriture décimale des onze premières puissances de deux.

Exercice 2

1. Calculer $2^6 + 2^4 + 2^3 + 2^0$.
2. En déduire l'écriture binaire de 89.

Exercice 3

1. Calculer $2^7 + 2^3 + 2^2 + 2^1$.
2. En déduire l'écriture décimale de $(10001110)_2$.

Exercice 4

En utilisant la méthode 2, donner l'écriture binaire de

1. 56

2. 35

3. 13

Exercice 5

En utilisant la méthode 3, donner l'écriture binaire de

1. 142

2. 273

3. 1000

Exercice 6

- Donner l'écriture décimale de $(1101\ 1010)_2$.
- Donner l'écriture binaire de 2016.
- Donner l'écriture hexadécimale de 2016.

Exercice 7

- Donner les écritures décimales de $(11)_2$, $(111)_2$, $(1111)_2$.
- Soit $n \in \mathbf{N}$, conjecturer la valeur de $\left(\underbrace{1 \dots 1}_{n \text{ chiffres}}\right)_2$.

Exercice 8

Pour multiplier par dix un entier naturel exprimé en base dix, il suffit d'ajouter un 0 à sa droite, par exemple, $12 \times 10 = 120$.
Quelle est l'opération équivalente pour les entiers naturels exprimés en base deux ?

Exercice 9

1. Donner l'écriture binaire de 174.

2. Donner celle de 17.

3. Poser l'addition de 174 et 17 en binaire.
4. Donner l'écriture décimale du résultat et vérifier.

Exercice 10

1. Donner l'écriture hexadécimale de 1022.
2. Donner celle de 3489.
3. Poser l'addition de 1022 et 3489 en hexadécimal.
4. Donner l'écriture décimale du résultat et vérifier.

Exercice 11*

Le Roi d'un pays imaginaire fait frapper sa monnaie par 8 nains : chacun d'entre eux produit des pièces d'or de 10g chacune.

Un jour, son Mage lui annonce : « Majesté, mon miroir magique m'a prévenu que certains de vos nains vous volent. Ils prélèvent 1g d'or sur chaque pièce qu'ils frappent. Pour vous aider à trouver les voleurs, voici une balance magique. Elle est précise au gramme près et peut peser autant que vous voulez. Malheureusement elle ne peut être utilisée qu'une fois. »

Le lendemain, le Roi convoque les 8 nains en demandant à chacun d'apporter un coffre rempli de pièces d'or qu'il a frappées.
On suppose que

- chaque nain dispose d'autant de pièces que nécessaire ;
- un nain honnête n'a que des pièces de 10g ;
- un nain voleur n'a que des pièces de 9g ;

Peux-tu aider le Roi pour démasquer les voleurs ?

Chapitre 2
Représentation des entiers

« Pour tout comprendre, lire ce chapitre en entier. »

Nous avons vu au chapitre précédent comment écrire les entiers naturels en binaire ou en hexadécimal. Maintenant nous allons étudier comment les entiers *relatifs*, c'est-à-dire positifs ou négatifs (on dit aussi *signés*) sont représentés en machine.
On va d'abord se limiter aux entiers naturels et on va voir qu'il n'y a pas de difficulté majeure à comprendre leur représentation en machine.

Remarque importante

> Il existe des *centaines* de langages de programmation. Les principaux sont : C, C++, C#, Java, Python, PHP et Javascript (cette liste est non exhaustive). Chaque langage utilise ses propres *types de variable* mais en général il y a beaucoup de ressemblances. On travaillera donc sur des exemples de types de variables utilisés dans tel ou tel langage...sachant que ce type n'existe pas nécessairement en Python.

1. Représentation des entiers naturels : l'exemple du unsigned char

En Informatique on dit souvent qu'un entier naturel est *non signé* (ou *unsigned* en anglais). Le type `unsigned char` se rencontre en C et en C++ (entre autres).

Un `unsigned char` est stocké sur un octet, c'est à dire 8 bits :

- l'octet 0000 0000 représente l'entier 0 ;
- 0000 0001 représente 1 ;
- et ainsi de suite jusqu'au plus grand entier représentable sur un octet : $(1111\ 1111)_2 = 255$.

On peut donc représenter les 256 premiers entiers avec un `unsigned char` et c'est logique : un octet, c'est 8 bits, chaque bit peut prendre 2 valeurs et $2^8 = 256$.
Si on a besoin de représenter des entiers plus grands, on pourra utiliser l'`unsigned short` : c'est la même chose mais ce type est représenté sur 2 octets. Donc on peut représenter les 2^{16} premiers entiers naturels, c'est à dire les nombres compris entre 0 et 65 535 inclus.

Cela continue avec l'`unsigned int` (sur 4 octets) et l'`unsigned long` (8 octets).

Remarque

> En PYTHON c'est différent : le type `int` (abréviation de *integer*, qui veut dire « entier » en Anglais) permet de représenter des entiers arbitrairement grands, les seules limitations étant la mémoire de la machine. Il n'y a qu'à évaluer 2^{100000} dans un *shell* PYTHON pour s'en convaincre.

2. L'exemple du type char

Le type `char` (qui n'existe pas en PYTHON) utilise un octet et l'on veut représenter des entiers relatifs (donc plus seulement positifs).

2.1. Une première idée... qui n'est pas si bonne

On pourrait décider que le bit de poids fort est un bit de signe : 0 pour les positifs et 1 pour les négatifs, par exemple. Les 7 autres bits serviraient à représenter la valeur absolue du nombre. Puisqu'avec 7 bits on peut aller jusqu'à $(111\ 1111)_2 = 127$, ce format permettrait de représenter tous les nombres entiers de -127 à 127.

Par exemple 1000 0011 représenterait -3 et 0001 1011 représenterait 27.

Il est un peu dommage que zéro ait 2 représentations : 0000 0000 et 1000 0000, mais ce qui est encore plus dommage c'est que lorsqu'on ajoute les représentations de -3 et de 27 voici ce qui se passe :

	1	0	0	0	0	0	1	1
+	0	0	0	1	1	0	1	1
=	1	0	0	1	1	1	1	0

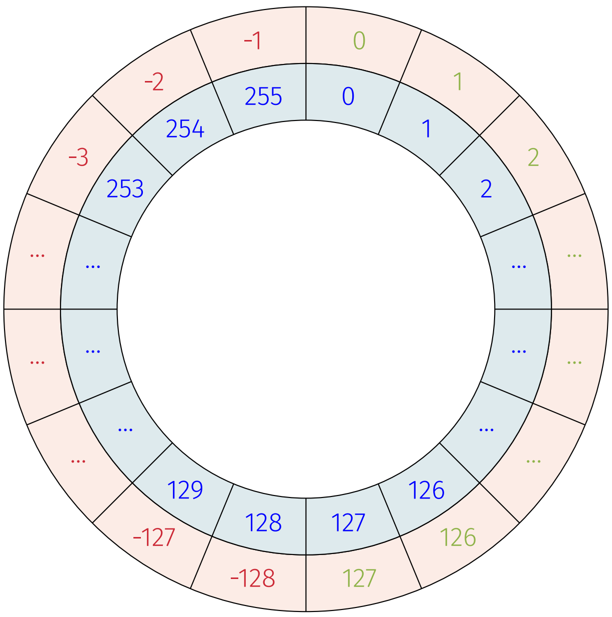
On obtient 10001 1110, qui représente -30... On aurait bien sûr préféré que cela nous donne 24.

2.2. La bonne idée : le complément à deux

Ce format, qui est utilisé avec le type `char`, permet de représenter les entiers de -128 à 127 :

Propriété : Représentation en complément à 2 sur un octet

- Soit x un entier positif plus petit ou égal à 127, alors on représente x par son écriture binaire (qui comprend 7 bits) et donc le bit de poids fort de l'octet (le 8e) est égal à zéro.
- Sinon si x est un entier strictement négatif plus grand que -128, on le représente par l'écriture binaire de $256+x$, qui est toujours représenté par un octet avec un bit de poids fort égal à 1.



Correspondance entre les valeurs « brutes » d'un octet (en bleu) et les entiers relatifs associés.

Exemples

Comment représenter 97 ? Ce nombre est positif, on le représente par son écriture binaire sur 8 bits : 0110 0001

Comment représenter -100 ? Ce nombre est négatif, il est donc représenté en machine par 256-100=156, c'est-à-dire 1001 1100

Que représente 0000 1101 ? Le bit de poids fort est nul donc cela représente $(0000\ 1101)_2$, c'est à dire 13.

Que représente 1000 1110 ? Le bit de poids fort est non nul. $(1000\ 1110)_2 = 142$ représente x avec donc $256 + x = 142$, c'est-à-dire $x = -114$.

Méthode

Pour passer d'un nombre à son opposé en complément à 2, en binaire on procède de *la droite vers la gauche* et

- on garde tous les zéros et le premier 1 ;
- on « inverse » tous les autres bits.

Exemple

Si on veut l'écriture en complément à 2 de -44 on commence par écrire 44 en base 2 :

$$44 = (0010\ 1100)_2$$

Puis on applique la méthode précédente :

$$\underbrace{00101}_{\text{on change}}\ \underbrace{100}_{\text{on garde}}$$

Ce qui nous donne

$$\underbrace{11010}_{\text{on a changé}}\ \underbrace{100}_{\text{on a gardé}}$$

Ainsi la représentation de -44 en complément à 2 sur 8 bits est 1101 0100.

Ce qui est agréable, c'est que **l'addition naturelle est compatible avec cette représentation** dans la mesure où

- on ne dépasse pas la capacité : on n'ajoutera pas 100 et 120 car cela dépasse 127 ;
- si on ajoute deux octets et que l'on a une retenue à la fin de l'addition (ce serait un 9^{e} bit), alors celle-ci n'est pas prise en compte.

Exemple

Ajoutons les représentations de 97 et -100 :

	0	1	1	0	0	0	0	1
+	1	0	0	1	1	1	0	0
=	1	1	1	1	1	1	0	1

> On a $(1111\ 1101)_2 = 253$, il représente donc x sachant que $256 + x = 253$, c'est à dire $x = -3$.
>
> On retrouve bien 97+(-100)=-3.

Attention à ne pas dépasser la capacité du format : en ajoutant 120 et 20 on obtient 140 qui, étant plus grand que 128, représente 140-156 = -116. C'est ce qu'affiche le programme suivant.

```
#include <iostream> // nécessaire pour utiliser cout

int main() // début de la fonction main
{
    char c1 = 120; // on définit une première variable
    char c2 = 20; // puis une deuxième
    char c3 = c1+c2; // on les ajoute
    std::cout << (int) c3; // on affiche le résultat en base 10
    return 0; // la fonction main renvoie traditionnellement
     ↪ zéro
}
```

3. Les principaux formats

En général, dans la majorité des langages (C, C++, C#, Java par exemple) les types suivants sont utilisés pour représenter les entiers relatifs (les noms peuvent varier d'un langage à l'autre) :

- `char` : Pour représenter les entiers compris entre -128 et +127. Nécessite un octet.
- `short` : Pour des entiers compris entre −32 768 et 37 267 (-2^{15} et $2^{15} - 1$). Codage sur 2 octets.
- `int` : (4 octets) entiers compris entre −2 147 483 648 et +2 147 483 647 (-2^{31} et $2^{31} - 1$).
- `long` : (8 octets) entiers compris entre −9 223 372 036 854 775 808 et +9 223 372 036 854 775 807 (-2^{63} et $2^{63} - 1$).

4. Quelques ordres de grandeur

- 1 kilooctet = 1 ko = 1 000 octets (fichiers textes)

- 1 mégaoctet = 1 Mo = 1 000 (fichiers .mp3)
- 1 gigaoctet = 1 Go = 1 000 (RAM des PC actuels, jeux)
- 1 téraoctet = 1 To = 1 000 (Disques durs actuels)
- 1 pétaoctet = 1 Po = 1 000 To
- 1 exaoctet = 1 Eo = 1 000 Po = 10^{18} octets (trafic internet mondial mensuel en 2022 : 376 Eo)
- 1 zétaoctet = 1 Zo = 1 000 Eo (stockage des données prévu en 2025 sur Terre : 175 Zo)

5. Exercices

Exercice 12

Donner les représentations en complément à deux sur un octet de :
88, 89, 90, -125, -2 et -3.

Exercice 13

Donner les représentations en complément à 2 (sur un octet) de -1 et de 92.
Ajouter ces deux représentations (en ignorant la dernière retenue). Quel nombre représente cette somme ?

Exercice 14

On considère le code C++ suivant :

```
#include <iostream> // bibliothèque d'affichage
int main() // début de la fonction principale
{
    unsigned char c = 0; // on définit la variable c
    for (int i = 0; i < 300; i++) // on fait une boucle
     ↪  pour
    {
        std::cout << "valeur de i : " << i;
        // on affiche la valeur de i
        std::cout << "  et valeur de c : " << (int)c;
         // on affiche la valeur de c en base 10
```

```
        std::cout << endl; // on revient à la ligne
        c++; // on augmente c
    }
    return 0; // la fonction principale renvoie zéro
}
```

Voilà ce que la console affiche :

```
valeur de i : 0 et valeur de c : 0
valeur de i : 1 et valeur de c : 1
```
et cætera
```
valeur de i : 254 et valeur de c : 254
valeur de i : 255 et valeur de c : 255
valeur de i : 256 et valeur de c : 0
valeur de i : 257 et valeur de c : 1
```
et cætera
```
valeur de i : 298 et valeur de c : 42
valeur de i : 299 et valeur de c : 43
```

Comment expliquer ceci ?

Exercice 15

On considère le code C++ suivant :

```
#include <iostream> // bibliothèque d'affichage
int main() // début de la fonction principale
{
    char c = 0; // on définit la variable c
    for (int i = 0; i < 257; i++) // on fait une boucle
     ↪ pour
    {
        std::cout << "valeur de i : " << i;
        // on affiche la valeur de i
        std::cout << "  et valeur de c : " << (int)c;
        // on affiche la valeur de c en base 10
        std::cout << endl; // on revient à la ligne
        c++; // on augmente c
    }
    return 0; // la fonction principale renvoie zéro
}
```

Voilà ce que la console affiche :

```
valeur de i : 0 et valeur de c : 0
valeur de i : 1 et valeur de c : 1
```
et cætera
```
valeur de i : 126 et valeur de c : 126
valeur de i : 127 et valeur de c : 127
valeur de i : 128 et valeur de c : -128
valeur de i : 129 et valeur de c : -127
```
et cætera
```
valeur de i : 254 et valeur de c : -2
valeur de i : 255 et valeur de c : -1
valeur de i : 256 et valeur de c : 0
```

Comment expliquer ceci ?

Chapitre 3

Représentation approximative des réels

« Tout cela est-il bien réel ? »

1. De la calculatrice à l'ordinateur

Dans une machine on *ne peut pas* représenter tous les nombres réels car la majorité a une écriture décimale illimitée et parmi celle-ci la majorité a une écriture décimale illimitée sans qu'aucun motif se répète, comme c'est le cas pour $\pi \simeq 3{,}141\,592\,653\,589\,793$.
Ceci dit on peut donner une valeur approchée d'un nombre réel r en écriture décimale en utilisant l'*écriture scientifique* vue au collège :

$$\boxed{r \approx (-1)^s \times d \times 10^e}$$

– s vaut 0 ou 1.

– d est un nombre décimal entre 1 inclus et 10 exclu.

– e est un entier relatif.

Ainsi, avec la calculatrice, on obtient :

$$\boxed{5,4^{-5} \approx (-1)^0 \times 2,177866231 \times 10^{-4}}$$

Dans ce cas, le nombre d comporte 10 chiffres décimaux, appelés *chiffres significatifs*.
Dans un ordinateur ne travaille pas avec des écritures décimales, mais avec des écritures *dyadiques* (l'équivalent des nombres décimaux en base 2).

Exemple : nombre décimal / nombre dyadique

- Considérons le nombre $x = 53,14$ (écrit en base 10). C'est un *nombre décimal* et on peut écrire :

$$\begin{aligned} x &= 53 + 1 \times 0,1 + 4 \times 0,04 \\ &= 5 \times 10^1 + 3 \times 10^0 + 1 \times 10^{-1} + 4 \times 10^{-2} \end{aligned}$$

- Les nombres dyadiques sont l'équivalent des nombres décimaux en base 2 : considérons $y = (101,011)_2$, on peut écrire :

$$\begin{aligned} y &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 4 + 1 + 0,25 + 0,125 \\ &= 5,375 \end{aligned}$$

Exemple : écriture scientifique pour un nombre dyadique

- En reprenant l'exemple précédent on a $53,14 = 5,314 \times 10^1$, c'est une écriture scientifique.
- Pour $(101,011)_2$, en se rappelant que multiplier par 2 décale la virgule d'un cran vers la gauche, on a $(101,011)_2 = (1,01011)_2 \times 2^2$.

Il faut donc décider d'un *format de représentation* des nombres dyadiques dans l'ordinateur :

- Combien de bits significatifs pour le nombre dyadique ?
- Quelle est la plage de valeurs pour l'exposant ?

2. Le format IEEE 754

Ce format est une norme pour représenter les nombres dyadiques (notamment par Python, en format 64 bits).
Par simplicité commençons par le format 32 bits (4 octets, donc). Voici comment cela fonctionne :

Définition

On considère un mot de 32 bits.

- Le premier bit s, indique le signe.
- Les 8 bits suivants $e_7 e_6 \dots e_0$ servent à coder l'exposant e de 2, qui vaut

$e = (e_7 \ldots e_0)_2 - 127$.

- Les 23 bits restants $m_1 \ldots m_{23}$ servent à coder la *mantisse*, notons $m = (1, m_1 \ldots m_{23})_2$.
- En définitive, ce mot de 32 bits représente

$$\boxed{(-1)^s \times m \times 2^e}$$

Ce qu'on peut noter

$$se_7 \ldots e_0 m_1 \ldots m_{23} \mapsto (-1)^s \times (1, m_1 \ldots m_{23})_2 \times 2^{(e_7 \ldots e_0)_2 - 127}$$

$se_7 \ldots e_0 m_1 \ldots m_{23}$ s'appelle une écriture *normalisée*.

Exemple : des 32 bits au nombre

Que représente $1\ \underbrace{0100\ 0011}\ \underbrace{1110\ 0000\ 0000\ 0000\ 0000\ 000}$?

- $s = 1$.
- $e = (0100\ 0011)_2 - 127 = 67 - 127 = -60$
- $m = (\boxed{1}, 11100000000000000000000)_2 = 1 + 2^{-1} + 2^{-2} + 2^{-3} = 1,875$
- $1\ 0100\ 0011\ 1110\ 0000\ 0000\ 0000\ 0000\ 000 \rightsquigarrow (-1)^1 \times 1,875 \times 2^{-60}$

Cela fait environ $-1,6263032587282567 \times 10^{-18}$

Attention

En PYTHON au format 64 bits, les nombres sont représentés sur 8 octets :

- 1 bit de signe ;
- 11 bits d'exposant (donc une plage de -1023 à 1024) ;
- 52 bits de mantisse.

Le principe est le même qu'en 32 bits.

3. Les limitations du format IEEE 754

Lorsqu'un format de représentation en virgule flottante est choisi (32 ou 64 bits), on ne peut pas représenter tous les nombres réels : il y a un plus grand nombre représentable (et son opposé pour les nombres négatifs) et un « plus petit nombre positif représentable » (le plus proche de zéro possible). De plus *on a automatiquement des valeurs approchées si le nombre que l'on veut représenter n'est pas de de la forme* $\frac{a}{2^n}$, *avec* $a \in \mathbb{Z}$ *et* $n \in \mathbb{N}$.

Par exemple, un nombre aussi simple que 0,1 (c'est-à-dire $\frac{1}{10}$) n'est pas de la forme $\frac{a}{2^n}$, avec $a \in \mathbb{Z}$ et $n \in \mathbb{N}$, donc son écriture dyadique ne se termine pas . On peut montrer que :

$$(0,1)_{10} = (0,0001\ 1001\ 1001\ 1001\ \ ...)_2$$

C'est l'équivalent dyadique de

$$\frac{1}{3} = (0,3333\,...)_{10}$$

Et puisque Python ne peut pas représenter intégralement le nombre 0,1 il l'approche du mieux qu'il peut : en fait pour Python la valeur de 0,1 est :

```
0.1000000000000000055511151231257827021181583404541015625
```

Mais celui-ci a la gentillesse d'afficher `0.1`.

Il faut se résigner à accepter les erreurs d'arrondis :

Python

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Cet exemple prouve que *tester l'égalité de 2* `float` *n'a pas d'utilité*. On aura plutôt intérêt à *tester si leur différence est très petite*.

De même, l'addition de plusieurs `float` donne un résultat qui peut dépendre de l'ordre dans lequel on les ajoute. Elle *n'est pas non plus associative* :

`a + (b + c)` et `(a + b) + c` peuvent avoir des valeurs différentes.

Les erreurs d'arrondis se cumulent. Pour les minimiser on aura intérêt à appliquer la règle suivante :

Propriété : Règle de la photo de classe

> Dans une somme de `float`, l'erreur est minimisée quand on commence par ajouter les termes de plus petite valeur absolue.

4. Exercices

Exercice 16

Donner l'écriture décimale des nombres suivants

a. $(101,1)_2$

b. $(1,011)_2$

c. $(0,1111\ 111)_2$ en remarquant que c'est « $(111\ 1111)_2$ divisé par 2^7.

Exercice 17

Écrire en base 2 les nombres suivants :

a. 3,5

b. 7,75

c. 27,625

Exercice 18 : étendue du « gruyère »

On considère le format IEEE 754 64 bits : 1 bit de signe, 11 bits d'exposant (donc une plage de -1023 à 1024) et 52 bits de mantisse.

1. Quel est le plus grand nombre positif représentable ?
2. Quel est le plus petit nombre positif représentable ?

Exercice 19 : taille des trous du « gruyère »

On considère le format IEEE 754 64 bits : 1 bit de signe, 11 bits d'exposant (donc une plage de -1023 à 1024) et 52 bits de mantisse.

1. Quel est le flottant immédiatement plus grand que 1 ? Quelle distance les sépare ?
2. Quel est le flottant immédiatement plus grand que le plus petit nombre positif représentable ? Quelle distance les sépare ?
3. Quel est le flottant immédiatement plus petit que le plus grand nombre positif représentable ? Quelle distance les sépare ?

Exercice 20

Écrire un programme déterminant le plus petit entier n pour lequel PYTHON considère que $1 + 2^{-n} = 1$.
Faire le lien avec le format IEEE 754 64 bits.

Exercice 21

Faire calculer `1+2**(-53)-1` puis `1-1+2**(-53)`.
Que remarque-t-on ?

Exercice 22

Écrire un programme déterminant le plus petit entier n pour lequel PYTHON considère que $2^{-n} = 0$.
En faisant le lien avec le format IEEE 754 64 bits, quelle valeur devrait-on trouver ?
Quelle explication peut-on imaginer (sachant qu'en PYTHON, les `float` sont bien codés sur 64 bits) ?

Exercice 23

On peut prouver (c'est dur) que

$$\sum_{n=1}^{+\infty} \frac{1}{n^4} = \frac{\pi^4}{90}$$

Appelons c cette constante.

1. Calculer à l'aide d'un script $S = \sum_{n=1}^{10^6} \frac{1}{n^4}$.

Ajoute-t-on des termes de plus en plus petits ou de plus en plus grands ?
Continuer le script pour afficher $c - S$ (on pourra utiliser `from math import pi`).

2. Calculer S « dans l'autre sens ».
Afficher $c - S$.

3. Qu'illustre cet exercice ?

Exercice 24 (quand nous aurons vu les fonctions)

1. Montrer qu'un triangle (3,4,5) est rectangle, ainsi qu'un triangle ($\sqrt{11}$, $\sqrt{12}$, $\sqrt{23}$).

2. Écrire une fonction `est_rectangle(a,b,c) :` qui renvoie `True` si `c**2 == a**2 + b**2` et, sinon, qui renvoie la différence entre `c**2` et `a**2+b**2`.

3. Tester la fonction `est_rectangle` avec les deux triangles précédents. Que remarque-t-on ? Comment modifier la fonction pour qu'elle soit plus satisfaisante ?

Exercice 25**

On aimerait trouver l'écriture dyadique (illimitée) de $\dfrac{1}{3}$. On note donc

$$\frac{1}{3} = (0, a_1 a_2 a_3 \ldots)_2$$

où a_i vaut 1 ou 0.

1. Expliquer pourquoi a_1 vaut *nécessairement 0*.
2. On note $x = \frac{1}{3}$. Montrer que x vérifie $4x = 1 + x$.
3. Quelle est l'écriture dyadique de $4x$?
4. Quelle est celle de $1 + x$?
5. En écrivant que ces 2 écritures représentent le même nombre, en déduire que
$$\frac{1}{3} = (0, 0101\ 0101 \ldots)_2$$

Chapitre 4
Représentation du texte

« Peux-tu décoder ce texte ? »

1. Le code ASCII

Pour représenter les caractères que nous utilisons pour écrire, on a historiquement choisi d'associer *un numéro* (ou code) à chacun de ces caractères. La correspondance entre chaque caractère et son code était appelée un *Charset*. Puisqu'à l'origine seul un petit nombre de caractères était utilisé (les caractères de base anglo-saxons), un octet suffisait pour les représenter tous.
Le fait de représenter en machine un jeu de caractères s'appelle réaliser un encodage (*encoding* en Anglais).

Le premier encodage utilisé fut l'ASCII, qui signifie *American Standard Code for Information Interchange*.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	space	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

La table ASCII

Le code ASCII se base sur un tableau contenant les caractères les plus utilisés en langue anglaise : les lettres de l'alphabet en majuscule (de A à Z) et en minuscule (de a à z), les dix chiffres arabes (de 0 à 9), des signes de ponctuation (point, virgule, point-virgule, deux points, points d'exclamation et d'interrogation, apostrophe ou *quote*, guillemet ou *double quotes*, parenthèses, crochets etc.), quelques symboles et certains caractères spéciaux invisibles (espace, retour-chariot, tabulation, retour-arrière, etc.).

Exercice 26

Combien y a -t-il de caractères dans ce catalogue ?
Combien de bits sont nécessaires pour pouvoir représenter tous leurs numéros de code ?

Pour représenter ces symboles ASCII, les ordinateurs utilisaient des cases mémoires de un octet, mais ils réservaient toujours le huitième bit pour le contrôle de parité : c'est un procédé de sécurité pour éviter les erreurs, qui étaient très fréquentes dans les premières mémoires électroniques.

Méthode : contrôle d'erreur par parité

- On dispose d'un mot de 7 bits, par exemple 111 0011
- On compte le nombre de bits à 1, il y en a 5.
- On rajoute le bit de poids fort à 1 pour qu'en tout, il y ait toujours *un nombre pair* de bits à 1.
  On obtient $\boxed{1}$111 0011, ce bit de poids fort jouant le rôle de *code correcteur*.
- Un autre exemple : 001 1110 est codé $\boxed{0}$001 1110.

Exercice 27

Voici un message reçu à l'issue d'une transmission :

```
53 E1 6C F5 70
```

Ces 6 octets sont censés représenter 6 caractères ASCII, codées sur 7 bits le 8^{e} étant réservé au contrôle d'erreur par parité.

1. Décoder ces 6 octets en disant s'il y a des erreurs ou non.
2. Quel était le message initial ?

2. L'insuffisance de l'ASCII

Pour coder les lettres accentuées, inutilisées en Anglais mais très fréquentes dans d'autres langues (notamment le Français), on a décidé d'étendre le codage des caractères au huitième bit (les erreurs-mémoire étant devenues plus rares et les méthodes de contrôle d'erreurs plus efficaces).

Exercice 28

> Combien de nouveaux symboles a-t-on pu coder en autorisant le huitième bit dans le codage ?

On a alors pu coder toutes ces lettres et ainsi que de nouveaux caractères typographiques utiles tels que différents tirets.

3. Le problème

Le fait d'utiliser un bit supplémentaire a bien entendu ouvert des possibilités mais malheureusement les caractères de toutes les langues ne pouvaient être pris en charge en même temps.
La norme ISO 8859–1 appelée aussi Latin-1 ou Europe occidentale est la première partie d'une norme plus complète appelée **ISO 8859** (qui comprend 16 parties) et qui permet de coder tous les caractères des langues européennes. Cette norme ISO 8859–1 permet de coder 191 caractères de l'alphabet latin qui avaient à l'époque été jugés essentiels dans l'écriture, mais omet quelques caractères fort utiles (ainsi, la ligature œ n'y figure pas).

Dans les pays occidentaux, cette norme est utilisée par de nombreux systèmes d'exploitation, dont Linux et Windows. Elle a donné lieu à quelques extensions et adaptations, dont **Windows-12527** (appelée **ANSI**) et ISO 8859-158 (qui prend en compte le symbole € créé après la norme ISO 8859-1). C'est une source de grande confusion pour les développeurs de programmes informatiques car un même caractère peut être codé différemment suivant la norme utilisée .
Voici les tableaux décrivant deux encodages :

Windows-1252 (CP1252)																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž	
9x		‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ
Ax	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
Bx	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

ISO/CEI 8859-15																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	*non utilisé*															
1x																
2x		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*non utilisé*															
9x																
Ax		¡	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Exercice 29

Ces deux encodages sont-ils totalement compatibles ? Pourquoi ?

4. La multiplicité des encodages

Au fil du temps une multitude d'encodages sont apparus, multipliant les sources de confusion.
Voici pour l'exemple une partie des encodages que PYTHON reconnaît :

Encodage	Alias Python	Langues concernées
ascii	646,us-ascii	English
big5	big5-tw, csbig5	Traditional Chinese
cp424	EBCDIC-CP-HE, IBM424	Hebrew
cp437	437, IBM437	English
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	Western Europe
cp720		Arabic
cp737		Greek
cp856		Hebrew
cp857	857, IBM857	Turkish
cp864	IBM864	Arabic
cp874		Thai
cp932	932, ms932, mskanji, ms-kanji	Japanese
cp1251	windows-1251	Bulgarian, Byelorussian, Macedonian, Russian, Serbian
cp1258	windows-1258	Vietnamese
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	Korean
gbk	936, cp936, ms936	Unified Chinese
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
iso8859_14	iso-8859-14, latin8, L8	Celtic languages
koi8_r		Russian
utf_8	U8, UTF, utf8	all languages

Exercice 30

Lequel de ces encodages semble le plus performant ?

5. L'Unicode

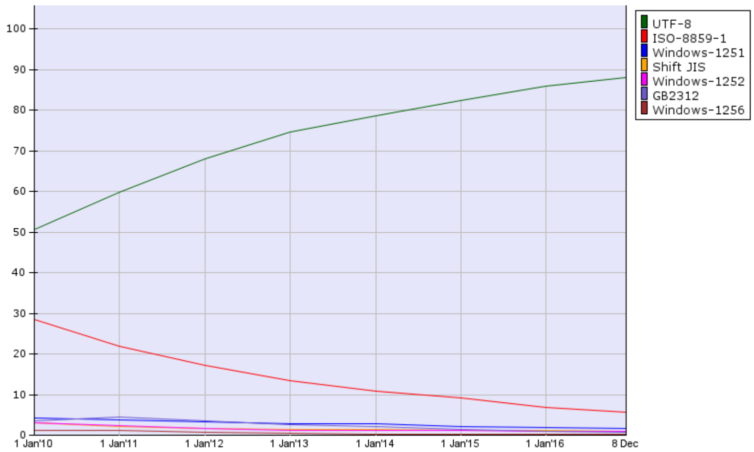
La globalisation des échanges culturels et économiques a mis l'accent sur le fait que les langues européennes coexistent avec de nombreuses autres langues aux alphabets spécifiques voire sans alphabet (le Japonais utilise entre autres un syllabaire, chaque symbole représentant une syllabe). La généralisation de l'utilisation d'Internet dans le monde a ainsi nécessité une prise en compte d'un nombre beaucoup plus important de caractères (à titre d'exemple, le mandarin possède à lui tout seul plus de 5000 caractères !).
Une autre motivation pour cette évolution résidait dans les possibles confusions dues au trop faible nombre de caractères pris en compte ; ainsi, les symboles monétaires des différents pays n'étaient pas tous représentés dans le système ISO 8859-1, de sorte que les ordres de paiement internationaux transmis par courrier électronique risquaient d'être mal compris. La norme Unicode a donc été créée pour permettre le codage de textes écrits quel que soit le système d'écriture utilisé.

Dans le système UTF-8, on attribue à chaque caractère un nom, une position normative et un bref descriptif qui seront les mêmes quelle que soit la plate-forme informatique ou le logiciel utilisés.
Un consortium composé d'informaticiens, de chercheurs, de linguistes et de personnalités représentant les états ainsi que les entreprises s'occupe d'unifier toutes les pratiques en un seul et même système : *l'Unicode*.

Définition

> L'*Unicode* est une table de correspondance Caractère-Code (Charset), et l'*UTF-8* est l'encodage correspondant (Encoding) le plus répandu.

De nos jours, par défaut, les navigateurs Internet utilisent le codage UTF-8 et les concepteurs de sites pensent de plus en plus à créer leurs pages web en prenant en compte cette même norme ; c'est pourquoi il y a de moins en moins de problèmes de compatibilité : l'UTF-8 est aujourd'hui majoritairement utilisé pour les sites du web, comme le montre ce graphique.



L'UTF-8 est également le codage le plus utilisé dans les systèmes GNU, Linux et compatibles pour gérer le plus simplement possible des textes et leurs traductions dans tous les systèmes d'écritures et tous les alphabets du monde.

6. L'UTF-8 côté technique

La norme Unicode définit entre autres un ensemble (ou répertoire) de caractères. Chaque caractère est repéré dans cet ensemble par un index entier aussi appelé « point de code ».
Par exemple le caractère « € » (euro) est le 8365ème caractère du répertoire Unicode, son index, ou point de code, est donc 8364 (on commence à compter à partir de 0).
Le répertoire Unicode peut contenir plus d'un million de caractères, ce qui est bien trop grand pour être codé par un seul octet (limité à des valeurs entre 0 et 255). La norme Unicode définit donc des méthodes standardisées pour coder et stocker cet index sous forme de séquence d'octets : UTF-8 est la plus utilisée d'entre elles (il y a aussi des variantes comme UTF-16 et UTF-32). En UTF-8, tout caractère est codé sur 1, 2, 3 ou 4 octets.
La principale caractéristique d'UTF-8 est qu'elle est *rétro-compatible avec la norme ASCII*, c'est-à-dire que tout caractère ASCII se code en UTF-8 sous forme d'un unique octet, identique au code ASCII.
Par exemple « A » (A majuscule) a pour code ASCII 65 et se code en UTF-8 par l'octet 65, il en va de même pour tous les caractère ASCII. Pour les autres, on procède comme ceci :

- Chaque caractère est associé à son index Unicode.
- En général, cet index est exprimé en hexadécimal. Actuellement, presques toutes les valeurs de 0000 à FFFF, c'est-à-dire de 0 à 65535 sont attribuées à

des « alphabets » associés à des langues, des plus communes aux plus rares, et à divers symboles, tels que les symboles mathématiques. Au delà de FFFF on trouve des alphabets associés à des langues anciennes (cunéiformes, hiéroglyphes...).

– En fonction du nombre de bits nécessaires pour représenter en binaire cet index, on utilise le codage suivant :

Nombre de bits de l'index	Nombre d'octets pour coder en UTF-8	Schéma de codage
de 0 à 7	1	0xxx xxxx
de 8 à 11	2	**110**x xxxx **10**xx xxxx
de 12 à 16	3	**1110** xxxx **10**xx xxxx **10**xx xxxx
de 17 à 21	4	**1111 0**xxx **10**xx xxxx **10**xx xxxx **10**xx xxxx

Par exemple le symbole € a un index Unicode qui vaut 8364.

– 8364 s'écrit 20AC en hexa, ce qui fait 10 0000 1010 1100 en binaire, soit 14 bits. On va donc utiliser 3 octets pour coder, conformément au schéma de codage.

– On commence par écrire le mot de 16 bits correspondant : $\boxed{00}$10 0000 1010 1100.

– On formate comme à la 3ème ligne du tableau :

1110 0010 **10**00 0010 **10**10 1100

Ce qui fait 3 octets : E2 82 AC en hexadécimal.

Exercice 31

> Si un ordinateur lit cet encodage UTF-8 du symbole € selon l'encodage ISO8859-15, qu'affichera-t-il ?

7. Conclusion

Même si l'encodage UTF-8 devient le standard international, certains développeurs, sites, ou applications en utilisent malgré tout encore d'autres.

Propriété

> La notion de texte brut n'existe pas en informatique : lorsqu'un ordinateur lit un fichier texte il n'a *a priori* aucun moyen de savoir quel est son encodage.

Beaucoup de documents indiquent donc en en tête leur format d'encodage : en HTML, on écrira dans l'en-tête d'une page :

HTML

```
<meta charset="utf8"/>
```

pour préciser qu'elle est encodée en UTF-8.

8. Et Python dans tout ça ?

En PYTHON, on pourra aussi écrire : `## -*- coding: utf8 -*-` en première ligne de tout script pour signifier la même chose, *et cætera*.

PYTHON gère très bien les encodages. On peut fabriquer un convertisseur très rapidement :

Python

```
fichier = open("nom_fichier", 'rt', encoding="utf8")
texte = fichier.read()
fichier.close()
```

Ceci permet de lire le contenu d'un fichier texte (d'où le `'rt'`, pour 'read text') d'un fichier texte encodé en UTF-8, et de le stocker dans la variable `texte`, de type `str`.

Python

```
fichier = open("nom_fichier", 'wt', encoding="utf8")
fichier.write("Salut")
fichier.close()
```

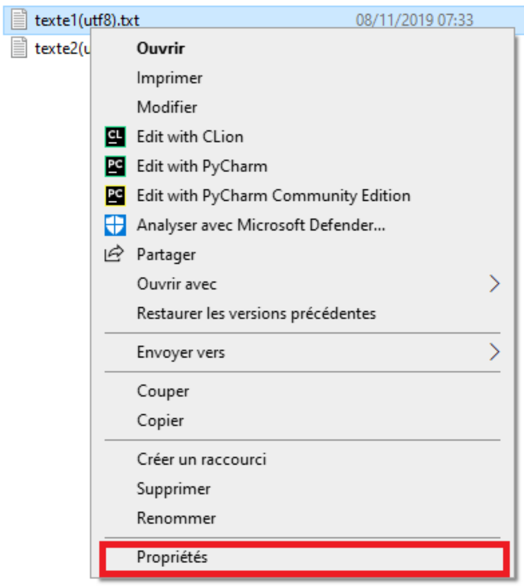
Permet d'écrire un fichier texte (d'où le `'wt'`, pour write text) en UTF-8.

9. Exercices

Exercice 32

Analyser les deux fichiers `texte1(utf8).txt` et `texte1(utf8).txt` : ils sont tous les deux encodés en UTF-8.

- Ouvrir chaque fichier. Quelles sont les différences de contenu entre ces deux fichiers ?
- Faire un clic droit puis `Propriétés` comme ceci :



  Quelles sont les tailles de ces deux fichiers ?
  Comment, dans le détail, la différence de taille s'explique-t-elle ?

Exercice 33

Nous allons examiner des problèmes d'encodage.

- Ouvrir le fichier `index1(utf8).html` avec un navigateur, puis le fichier `index2(utf8).html` avec un navigateur.
  Que remarquez-vous ?

- Ouvrir ce dernier fichier avec un éditeur de texte (comme le bloc-notes Windows).

D'où vient le problème ? Proposer une correction.
Expliquer en détail pourquoi à la place des « é », il y a des « Ã© » .

- Ouvrir le fichier `index3(ISO8859-15).html` avec un navigateur.
  D'où vient le problème ? Proposer une correction.
  Expliquer en détail pourquoi il y a des « points d'interrogation dans des losanges noirs ».

Exercice 34

Écrire un script qui corrige le problème de `index3(ISO8859-15).html` en le convertissant en UTF-8 (utiliser les scripts PYTHON fournis à la section précédente).

Partie II

Architectures matérielles et systèmes d'exploitation

Chapitre 5
Turing et Von Neumann

1. Un peu d'histoire

1.1. La machine de Turing

En 1936, Alan Turing publie un article de mathématiques, fruit de ses réflexions sur le thème : « est-il possible de déterminer de manière mécanique si un énoncé mathématique valide est vrai ou non ? ».
C'est une question cruciale : si sa réponse est « oui » cela veut dire qu'il sera peut-être possible de fabriquer une machine qui nous dira si un énoncé (par exemple un théorème qu'on aimerait démontrer) est vrai ou non. Plus besoin de démontrer car la machine le fera à notre place !
Turing est amené à proposer un modèle abstrait de machine de calcul que l'on appelle désormais *machine de Turing*.

Alan Turing (1912-1954) était aussi marathonien.

Il faut donc imaginer une machine qui peut se déplacer sur un ruban aussi étendu qu'on le désire en avançant ou reculant d'une case à la fois.
Définir une machine M c'est se donner

– un ensemble d'*états* $\mathcal{E}$ dans lesquels la machine M pourra se trouver ;

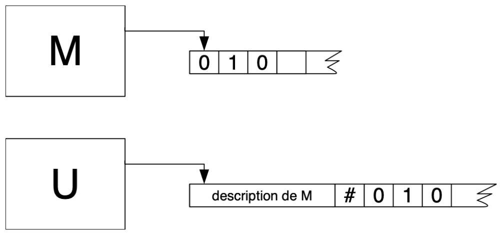
- un ensemble de *symboles* (un alphabet) $\mathcal{A}$, que la machine peut lire ou écrire sur le ruban ;

- un ensemble de *règles* qui décrivent, selon l'état dans lequel M se trouve et le symbole qu'elle lit, quel symbole elle écrit à la place de ce symbole sur le ruban et dans quel sens elle se déplace pour lire le prochain symbole. Cet ensemble de règles peut s'appeler un *programme*.

Exercice 35

> Faire l'activité « Machine de Turing ».

La machine décrite précédemment ne possède qu'un seul programme. Turing a donc eu l'idée de ce que l'on appelle maintenant une *Machine de Turing universelle* (MTU).



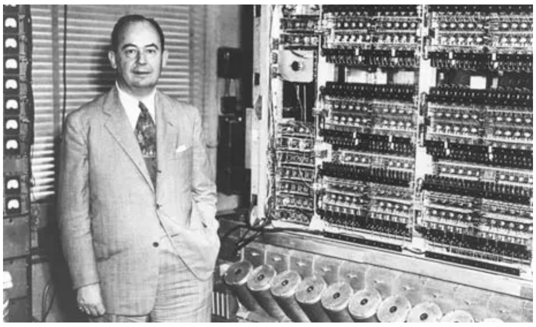
Une machine de Truing universelle.

Il s'agit d'une machine de Turing U qui est capable de simuler n'importe quelle machine de Turing M, pourvu qu'on lui fournisse l'ensemble des règles de M et son état de départ. C'est en quelque sorte l'origine de l'ordinateur programmable, le programme étant les règles de M, et M étant variable.

1.2. L'ordinateur programmable

Au milieu des années 40, John Von Neumann et ses collègues ont mis au point une version concrète de l'ordinateur programmable avec une architecture révolutionnaire pour l'époque, qui reste commune à la plupart des ordinateurs actuels et qui porte le nom d'*architecture de Von Neumann*. Celui-ci a expliqué que l'idée de machine de Turing universelle a directement inspiré le projet.

Parmi les premiers ordinateurs figurent l'ENIAC, ordinateur opérationnel à la fin de l'année 1945 et destiné à effectuer des calculs balistiques. C'est une énorme machine, de 90cm d'épaisseur, 2,40m de haut et … 30,5m de long, le tout pour un poids de 30 tonnes.



John Von Neumann (1903-1957).

Les premières personnes à programmer cet ordinateur sont six femmes, toutes mathématiciennes. L'ordinateur peur réaliser 100 000 additions par secondes, ou encore 357 multiplications par seconde, ou encore 38 divisions par seconde, le tout avec une capacité mémoire de 20 nombres signés à 10 chiffres en base 10.

Quatre des six programmeuses de l'ENIAC.

On peut voir la taille de quelques-uns des 17 468 tubes à vide qui entraient dans la composition de l'ordinateur. Dès 1947 les transistors ont remplacé les tubes à vides et n'ont cessé d'être miniaturisés. De nos jours les transistors sont directement gravés dans le silicium, leur taille fait quelques nanomètres (10^{-9}m) et une carte graphique RTX 2080 en comporte quasiment 20 milliards. La puissance de calcul a beaucoup augmenté puisque le microprocesseur d'un bon PC actuel a une puissance d'environ 150 GFLOPS, c'est à dire 150 milliards d'opérations sur des nombres en virgule flottante par seconde !

Exercice 36

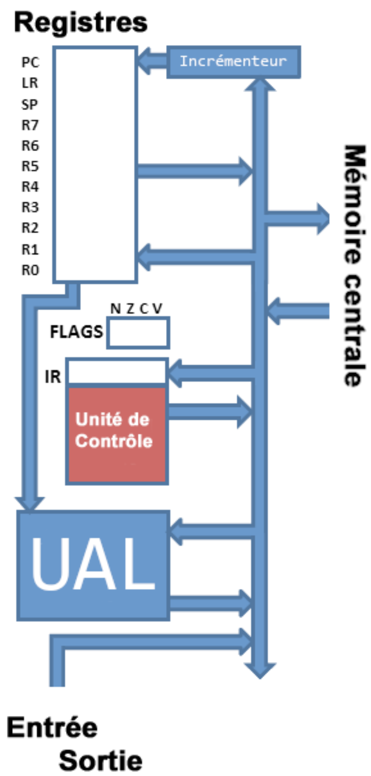
> Entre un transistor des années 50 (quelques centimètres) et un transistor actuel, quel est le facteur de réduction de taille ?

> Entre l'ENIAC et ses multiplications par seconde et un PC actuel, quel est le facteur d'augmentation de puissance de calcul ?

2. L'architecture de Von Neumann

Elle décompose l'ordinateur en 4 parties distinctes :

- l'*unité arithmétique et logique* ou UAL, dont le rôle est d'effectuer des opérations de base ;
- l'*unité de contrôle* et son registre IR, qui est chargée de séquencer les opérations ;
- la *mémoire* qui stocke à la fois les données à utiliser et le programme que l'unité de contrôle va séquencer.
- les périphériques d'*entrée-sortie* qui permettent de communiquer dans les 2 sens avec l'extérieur.



Modèle simplifié de *microprocesseur* (CPU : *Central Processing Unit*). Les données transitent par des bus (flèches bleues).

Dans le CPU se trouvent des registres :

- PC (*Program Counter*) qui indique à l'unité de contrôle où aller chercher la prochaine instruction ;
- LR (*Link Register*) qui contient l'adresse à laquelle le programme doit revenir dans le cas où on aurait appelé un sous-programme (l'équivalent d'une fonction) ;
- SP (*Stack Pointer* qui contient l'adresse du sommet de la pile (la pile est un endroit de la mémoire où l'on « empile » et « dépile » des données, comme une pile d'assiettes) ;

Certaines opérations déclenchent des « événements » : quand un résultat est nul, le *flag* (drapeau) Z est mis à un, *et cætera*.
Un processeur donné est capable d'exécuter un nombre d'instructions de base relativement limité. L'ensemble de ces instructions est appelé *langage machine*. Chaque instruction machine est composé d'une ou deux parties :

- un code opération (appelé *opcode*) qui indique le type de traitement à réaliser ;
- les données éventuelles sur lesquelles l'opération doit être réalisée.

Le fonctionnement d'un CPU est cyclique, la fréquence des cycles étant réglée par une horloge (par exemple un processeur moderne cadencé à 3GHz effectue 3 milliards de cycles par seconde).

Déroulement d'un cycle

- le contenu de la RAM pointé par PC est copié dans l'IR de l'unité de contrôle ;
- l'unité de contrôle décode l'instruction qu'on lui donne et la fait exécuter ;
- l'exécution provoque l'utilisation des registres, et/ou une lecture ou écriture dans la RAM, éventuellement un accès aux entrées/sorties.

Les instructions machine étant « désespérément austères » lorsqu'on les écrit en binaire ou en hexadécimal, on les écrit dans un langage compréhensible par les humains. Ce langage s'appelle l'*assembleur*.

Exercice 37

> Faire l'activité : Simulateur de CPU.

Chapitre 6
Logique

1. Du transistor à l'ordinateur

Le transistor est à la base de la plupart des composants d'un ordinateur. Pour faire simple c'est un composant avec une entrée, une sortie, et une alimentation.Quand il est alimenté, le transistor laisse passer le courant de l'entrée vers la sortie et dans le cas contraire le courant ne passe pas.
C'est l'élément de base des *circuits logiques*.

Définition : circuit logique

> Un circuit logique prend en entrée un ou plusieurs signaux électriques.
> Chacun de ses signaux peut être dans l'état 0 ou l'état 1.
> En sortie, le circuit logique produit un signal (0 ou 1) obtenu en appliquant des *opérations booléennes* aux signaux d'entrée.

Un circuit logique est une implémentation matérielle d'une *fonction logique*. La fonction logique est, quant à elle, la version « mathématique » du circuit. Nous confondrons ces deux notions par la suite.

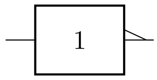
1.1. Opérateurs logiques de base

À l'aide des opérateurs suivants, on peut construire toutes les fonctions logiques.

L'opérateur « non »

C'est un opérateur **unaire** : il ne prend qu'une seule variable booléenne en entrée.

x	non x
0	1
1	0



La table de vérité et le symbole de porte européen du `non`.

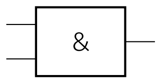
Cet opérateur renvoie « le contraire de ce qu'il a reçu ».
Parmi les notations que l'on rencontre pour noter « non x » il y a

- `NOT x`
- $\overline{x}$
- `!x`

L'opérateur « et »

C'est un opérateur **binaire** : il prend deux variables booléennes en entrée.

x	y	x et y
0	0	0
0	1	0
1	0	0
1	1	1



La table de vérité et le symbole de porte européen du `et`.

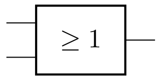
Un « et » ne renvoie vrai que si ses deux entrées sont vraies.
Parmi les notations que l'on rencontre pour noter « x et y » il y a

- `x AND y`
- $x \wedge y$
- `x && y`

L'opérateur « ou »

C'est également un opérateur **binaire**.

x	y	xou y
0	0	0
0	1	1
1	0	1
1	1	1



La table de vérité et le symbole de porte européen du ou.

Un « ou » ne renvoie faux que si ses deux entrées sont fausses.
Parmi les notations que l'on rencontre pour noter « x ou y » il y a

- `x OR y`
- $x \vee y$
- `x || y`

On peut montrer qu'il est possible de se passer de la fonction *et* et que toutes les fonctions logiques peuvent s'écrire à l'aide de fonctions *non* et *ou* (on peut même n'utiliser qu'une seule fonction : la fonction « non ou »). Le choix de ces trois fonctions *et*, *ou* et *non* est donc, en quelque sorte, arbitraire.

Définition : Équivalence de deux circuits/fonctions logiques

On dira que deux fonctions logiques sont équivalentes lorsqu'elles prennent le même nombre de variables en entrée (le même nombre de signaux si on parle de circuits) et si ces deux fonctions donnent le même résultat lorsque les variables d'entrées ont les mêmes valeurs : on dit que les fonctions ont même *table de vérité*. Lorsque deux fonctions logiques sont équivalentes, on dit aussi que leurs expressions booléennes sont équivalentes.

Exemples

- Les expressions booléennes not(A and B) et (not A) or (not B) sont équivalentes.
  Pour le prouver il suffit de vérifier que leurs tables de vérité sont les mêmes : on fait varier les valeurs de A et de B selon toutes les possibilités et on regarde le résultat de chaque expression.

A	B	A and B	not (A and B)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

A	B	not A	not B	(not A) or (not B)
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

Les dernières colonnes de chaque tableau sont les mêmes : les expressions sont donc équivalentes.

- En procédant de même on montre très facilement que « non non x » et « x » sont équivalentes.

Exercice 38

Montrer que not(A or B) et (not A) and (not B) sont équivalentes.

Exercice 39

On peut définir le « ou exclusif » noté xor comme ceci : A xor B n'est vrai que si A est vrai ou B est vrai, mais pas les deux.

1. Donner la table de vérité de xor.
2. Montrer que A xor B équivaut à (A or B) and (not(A and B)).
3. Montrer que A xor B équivaut à (A and (not B)) or ((not A) and B).
4. Représenter A xor B avec un circuit logique, en utilisant les symboles de porte logique européens.

Exercice 40

On définit l'opération « nor », notée ↓ par :

$$A \downarrow B = not(A \; or \; B)$$

Cette opération est dite *universelle* car elle permet de retrouver toutes les autres opérations.

1. Écrire la table de vérité de nor.
2. Montrer que $A \downarrow A = not\ A$.
3. En utilisant les exemples de la page précédente, en déduire que

$$(A \downarrow B) \downarrow (A \downarrow B) = A\ or\ B$$

4. Comment à partir de A, B et $\downarrow$ obtenir $A\ and\ B$?

1.2. Un exemple détaillé : le multiplexeur

Il s'agit d'une fonction très importante : soient A, B et C trois variables logiques, alors m(A,B,C)=B si A vaut 0 et C si A vaut 1.
m permet donc de sélectionner B ou C suivant la valeur de A. Voici la table de valeurs de m :

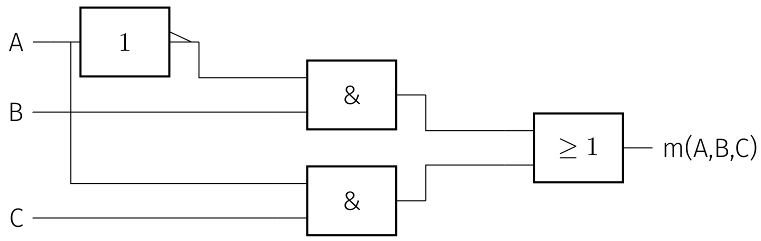
A	B	C	m(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

On va décomposer m à l'aide des opérateurs de base. Ici, un raisonnement simple permet d'y arriver :

- quand A vaut 0, on peut garder la valeur de B en faisant (*non* A) *et* B ;
- quand A vaut 1, on peut garder la valeur de C en faisant A *et* C ;
- il se trouve que ces deux observations vont bien ensemble car quand A vaut 0, A *et* C vaut automatiquement 0, et quand A vaut 1, (*non* A) *et* B vaut automatiquement 0 ;
- on en conclut que l'**expression symbolique** de m est

m(A,B,C) = (A *et* C) ou ((*non* A) *et* B).

Le circuit logique modélisant m est le suivant :



Exercice 41

On veut construire un « additionneur » selon le principe suivant :

- A et B représentent deux bits à ajouter
- S et R sont respectivement
  - la somme (sur un bit, donc) de A et B ;
  - la retenue.

Par exemple si A et B valent 1, alors S vaudra 0 et R vaudra 1.

1. Donner les tables de vérités de R et de S.
2. Exprimer R et S en fonction de A et B et des opérations « non », « ou » et « et ».
3. Représenter R et S avec un (ou des) circuit(s) logique(s), en utilisant les symboles de porte logique européens.

Exercice 42

Pour chiffrer un message, une méthode, dite du masque jetable, consiste à le combiner avec une chaîne de caractères de longueur comparable. Une implémentation possible utilise l'opérateur XOR (ou exclusif) dont voici la table de vérité :

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Dans la suite, les nombres écrits en binaire seront précédés du préfixe `0b`.

1. Pour chiffrer un message, on convertit chacun de ses caractères en binaire (à l'aide du format UNICODE), et on réalise l'opération XOR bit

à bit avec la clé.

Après conversion en binaire, et avant que l'opération XOR bit à bit avec la clé n'ait été effectuée, Alice obtient le message suivant :

`m = 0b 0110 0011 0100 0110`

a. Le message `m` correspond à deux caractères codés chacun sur 8 bits : déterminer quels sont ces caractères. On fournit pour cela la table ci-dessous qui associe à l'écriture hexadécimale d'un octet le caractère correspondant.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Exemple de lecture : le caractère correspondant à l'octet codé `4A` en hexadécimal est la lettre `J`.

b. Pour chiffrer le message d'Alice, on réalise l'opération `XOR` bit à bit avec la clé suivante :

`k = 0b 1110 1110 1111 0000`

Donner l'écriture binaire du message obtenu.

2. a. Donner la table de vérité de l'expression booléenne `(a XOR b) XOR b`.

b. Bob connaît la chaîne de caractères utilisée par Alice pour chiffrer le message. Quelle opération doit-il réaliser pour déchiffrer son message ?

Partie III

Programmation avec Python

Chapitre 7
Types de variables

« Chic type ! »

À retenir

- les types de variables simples sont `bool`, `int` et `float` ;
- les types `str` et `list` sont structurés : on accède à leurs éléments par des indices entiers ;
- le type `dict` représente des couples « clé-valeurs » ;
- il est parfois possible (et souhaitable) de convertir le type d'une variable.

PYTHON distingue plusieurs types de variables, voici les principaux :

1. Le type bool

Il sert à représenter les *variables booléennes* : une variable de type `bool` est une valeur logique et vaut `True` (vrai) ou `False` (faux).

Python

```
>>> a = bool()
>>> a
False
>>> b = True
>>> b
True
>>> c = (3 > 2)
>>> c
True
```

Les `bool` servent très souvent lorsqu'on veut *tester* si une condition est vraie ou non (on y reviendra plus tard).
Sur les `bool`, on dispose d'*opérations logiques* : `or` (ou), `and` (et) et `not` (non).

Python

```
>>> True and False
False
>>> True or False
True
>>> not (3 < 1)
True
```

2. Le type int

Il sert à représenter les *entiers relatifs* (*integer* signifie « entier » en Anglais).
Sur les `int`, on dispose des opérations `+` (addition), `-` (soustraction) et `*` (multiplication).

Python

```
>>> a = int()
>>> a
0
>>> b = 3
>>> c = 2
>>> b + c
5
>>> 2 * b
6
>>> b - 2 * c
-1
```

On dispose également de *deux opérations très pratiques* : soient `a` et `b` deux `int`, et `b` non nul, alors on peut effectuer la *division euclidienne* de `a` par `b`.
`a // b` est le *quotient* et `a \% b` est le *reste*.

Python

```
>>> 64 // 10
6
```

```
>>> 64 % 10
4
>>> 22 // 7
3
>>> 22 % 7
2
```

On dispose de l'opération d'*exponentiation* (opération puissance), notée `**`. **Attention :** Cette opération peut produire un résultat *non-entier*, de type `float` (voir partie suivante).

Python

```
>>> 2 ** 3
8
>>> 10 ** 4
10_000
>>> 2 ** (-1)
0.5
```

Pour finir, la *division décimale* peut être effectuée sur des entiers, mais elle renvoie un résultat de type `float`.

Python

```
>>> 10 / 2
5.0
>>> 2 / 3
0.6666666666666666
```

3. Le type float

Le type `float` sert à représenter les *nombres à virgule flottante* (*to float* : flotter en Anglais). Ce sont (en gros) des nombres décimaux. PYTHON comprend et utilise la notation scientifique : `2.35e6` vaut $2,35 \times 10^6$, c'est-à-dire 2 350 000.

Python

```
>>> a = float()
>>> a
```

```
0.0
>>> b = 3 / 2
>>> b
1.5
>>> c = 1 / 100_000
>>> c
1e-05
>>> d = 1.2e-4
>>> d
0.00012
```

On peut pratiquer sur les `float` toutes les opérations vues avec les `int`. Pour des fonctions plus compliquées telles le cosinus ou l'exponentielle, on fait appel au module[1] `math` :

Python

```
>>> from math import *
>>> pi
>>> cos(pi / 3)
0.5000000000000001
>>> exp(2)
7.38905609893065
>>> log(2)
0.6931471805599453
>>> exp(log(2))
2.0
```

Ce que PYTHON note `log` est la *fonction logarithme népérien*[2] notée ln en France.

4. Le type str

Le type `str` pour représenter les *chaînes de caractères* (`str` est l'abréviation de *string*, qui veut dire chaîne en anglais). Lors d'une affectation à une variable de type `str`, on peut utiliser les symboles `'`, `"` ou même `'''` (suivant que la chaîne contient des apostrophes, ou des guillemets).

[1]Un module est un ensemble de *fonctions* et/ou de *constantes* que l'on peut importer si besoin est.

[2]Voir le programme de mathématiques de terminale scientifique.

Python

```
>>> a = str()
>>> a
''
>>> b = 'Bonjour.'
>>> b
'Bonjour'
>>> 'J'aime Python.'
SyntaxError
>>> "J'aime Python."
"J'aime Python."
>>> "Je n'aime pas qu'on m'appelle "geek"."
SyntaxError
>>> """Je n'aime pas qu'on m'appelle "geek"."""
'Je n\'aime pas qu\'on m\'appelle "geek".'
```

On peut additionner 2 variables de type `str`, on peut aussi multiplier une variable de type `str` par un `int`.

Python

```
>>> a = 'Yes'
>>> b = 'No'
>>> a + b
'YesNo'
>>> b + a
'NoYes'
>>> 2 * a + 3 * b
'YesYesNoNoNo'
```

Les variables de type `str` sont composées de caractères *alphanumériques*.
On peut y accéder de la manière suivante :

Python

```
>>> chaine = 'Bonjour !'
>>> chaine[0]
'B'
>>> chaine[5]
```

```
'u'
```

Voici comment PYTHON représente la chaine précédente :

i	0	1	2	3	4	5	6	7	8
chaine[i]	B	o	n	j	o	u	r		!

On a parfois besoin de connaître la longueur (*length* en anglais) d'une chaîne de caractères :

Python

```
>>> chaine = 'onzelettres'
>>> len(chaine)
11
```

On peut aussi accéder facilement au dernier (ou à l'avant dernier) caractère d'une variable de type `str` :

Python

```
>>> a = "M'enfin ?!"
>>> a[-1]
'!'
>>> a[-2]
'?'
```

Enfin on peut vouloir *extraire une sous-chaîne* d'une chaine : `chaine[p:q]` renvoie la sous-chaine de caractère qui commence par `chaine[p]` et qui se termine par `chaine[q-1]` :

Python

```
>>> a = 'Salut'
>>> a[1:3]
'al'
```

Pour terminer, on peut vouloir tronquer à gauche ou à droite. La syntaxe est la même :
- `chaine[p:]` renvoie la sous-chaine commençant par `chaine[p]`.
- `chaine[:p]` renvoie la sous-chaine se terminant par `chaine[p-1]`.

Python

```
>>> a = 'Salut'
>>> a[:3] + '0' + a[3:]
'Sal0ut'
```

5. Le type list

Le type `list` sert à représenter des *listes ordonnées*, indexées par $\{0; 1; \ldots; n\} \quad (n \in \mathbb{N})$.
Pour une variable `a` de type `list`, on accèdera à `a[0]`, `a[1]`...
Les éléments d'une liste peut être de types différents.

Python

```
>>> vide = list()
>>> vide
[]
>>> notes = [12, 11, 20, 9]
>>> notes[1]
11
>>> fouillis = [True, 1, 2.1, 'salut']
>>> fouillis[0]
True
```

On dispose des mêmes opérations que pour le type `str` (après tout, une chaine de caractère est une liste de lettres) : addition de deux listes, multiplication par un `int`.

On reviendra plus tard sur ce type très utile.

6. Le type dict

Le type `dict` pour représenter des *dictionnaires*.
Les dictionnaires généralisent les listes car ils peuvent être indexés par des ensembles quelconques.
Pour une variable `couleur` de type `dict`, on aura par exemple `couleur['ciel']='bleu'`.

Python

```
>>> chanteur = {'Nirvana': 'Kurt Cobain', 'U2': 'Bono'}
>>> chanteur['U2']
'Bono'
```

La notion de dictionnaire peut s'avérer assez compliquée à gérer (surtout au départ quand on manque de pratique) mais elle est très puissante et parfois très élégante.

7. La fonction type et les conversions de type

Pour connaître (ou vérifier) le type d'une variable, on peut utiliser la fonction `type`.

Python

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> type('hello')
<class 'str'>
>>> type(22 / 7)
<class 'float'>
```

Enfin, pour changer le type d'une variable, on peut utiliser les fonctions `int`, `float`, `str` et `list` (quand cela a un sens).

Python

```
>>> a = 2020
>>> b = float(a)
>>>       b
2020.0
>>> c = str(b)
>>> c
'2020.0'
>>> d = list(c)
>>> d
['2020.0']
```

8. Et les constantes ?

Certains langages permettent de définir des *constantes* (C++, JAVA par exemple). Une constante est « une variable dont la valeur est fixée lors de son initialisation et qui ne peut plus être modifiée ensuite.

Attention

> On ne peut pas définir de constantes dans PYTHON.
> Cependant on utilise la convention suivante : les variables écrites en majuscules sont à considérer comme des constantes.

Python

```
TAILLE_MINI = 140
TAILLE_MAXI = 200
```

9. Quelques « trucs » utiles

Affectations multiples

PYTHON permet d'affecter plusieurs valeurs à plusieurs variables en même temps.

Python

```
>>> a, b= 10, 2
>>> a
10
>>> b
2

>>> a, b = b, a
>>> a
2
>>> b
10
```

Notation condensée

On est souvent amené à écrire des instructions telles que `a = a + 1` ou `b = b / 2`. Cela peut être lourd quand les variables ne s'appellent pas `a` ou `b` mais `rayon_sphere` ou `largeur_niveau`. On peut utiliser les notation suivantes :

Python

```
>>> rayon_sphere = 3.4
>>> rayon_sphere /= 2
>>> rayon_sphere
1.7
>>> largeur_niveau = 19
>>> largeur_niveau += 1
>>> largeur_niveau
20
```

On dispose également de `*=`, `//=`, `\%=`, `-=` et `**=`.

10. Exercices

Exercice 43 : fonction print

La fonction `print` sert à afficher à l'écran le contenu d'une variable, ou bien une chaîne de caractère.
Recopier et observer :

```
>>> print('Bonjour')
>>> a = 2020
>>> print('Nous sommes en ', a)
>>> b = 7
>>> print('Dans ', b,' ans, nous serons en ', a + b)
```

Exercice 44 : affectations 1

Qu'affiche le script suivant (ne pas chercher à l'écrire en machine) ?

```
a = 1
b = a + 1
c = 2 * b - ( a + b - 4)
print(b, c)
```

Exercice 45 : affectations 2

Qu'affiche le script suivant ?

```
a, b = 2, 5
b, c = b + 1, a + b
print(c)
```

Exercice 46 : affectations 3

Qu'affiche le script suivant ?

```
a, b, c = 1, 2, 3
a, b, c = b, c, a
a, b, c = b, c, a
a, b, c = b, c, a
print(a, b, c)
```

Exercice 47 : fonction input

C'est le pendant de la fonction `print` : elle permet à l'utilisateur d'entrer une chaîne de caractères à l'aide du clavier.
Recopier et observer :

```
>>> prenom = input('Quel est ton prénom ? ')
>>> reponse = 'Enchanté, ' + prenom
>>> print(reponse)
```

Exercice 48

Le script suivant produit une erreur :

```
age = input('Quel est ton age ? ')
vieux = 10 + age
print('Dans 10 ans tu auras ', vieux, ' ans.')
```

Observer le message d'erreur (surtout la dernière ligne).
À l'aide d'une *conversion de type*, rectifier le script.

Exercice 49

D'où vient le problème ? Proposer une solution (deux si tu es astucieu·x·se).

```
>>> chaine = 'Mon nombre préféré est le '
>>> nb = 7
>>> print(chaine+nb)
```

Exercice 50

Donner 3 manières de définir en une ligne une variable `a` de type `float` valant 2.

Exercice 51

Que fait le script suivant (essayer de le comprendre sans le taper) ?

```
pouces = input('Entrez la valeur en pouces à convertir :
↪  ')
pouces = float(pouces)
print('Cela fait ', pouces * 2.54, ' centimètres.')
```

Écrire un script qui demande la taille d'un fichier en kilooctets (ko) puis la convertit en octets (1ko = 2^{10} octets).

Exercice 52

Le numéro de sécurité sociale est constitué de 13 chiffres auquel s'ajoute la clé de contrôle (2 chiffres).
Exemple :

$$\underbrace{1\ 89\ 11\ 26\ 108\ 268}_{\text{chiffres}}\ \underbrace{91}_{\text{clé}}$$

La clé de contrôle est calculée par la formule : 97 - (numéro de sécurité sociale modulo 97).
À l'écrit, retrouver la clé de contrôle de votre numéro de sécurité sociale (ou à défaut, de l'exemple).
Quel est l'intérêt de la clé de contrôle ?
Écrire un script qui, à partir des 13 chiffres du numéro de sécurité sociale, affiche le numéro complet.

Exercice 53

L'identifiant d'accès au réseau du lycée est construit de la manière suivante : initiale du prénom puis les 8 premiers caractères du nom (le tout en minuscule).
Exemple : Alexandre Lecouturier donne `alecoutur`.

Écrire un script qui, à partir des deux variables `prenom` et `nom`, affiche l'identifiant.
Tu pourras créer une variable `nom`, une variable `prenom` et utiliser la fonction `input`.
Pour passer `nom` et `prenom` en minuscule, il suffit de taper :

```
nom = nom.lower()
prenom = prenom.lower()
```

Chapitre 8

Tests et conditions

« Ceci n'est pas un test ! »

1. Des outils pour comparer

Ce sont les *opérateurs de comparaison* :

Opérateur	Signification	Remarques
`<`	strictement inférieur	Ordre usuel sur `int` et `float`, lexicographique sur `str`…
`<=`	inférieur ou égal	Idem
`>`	strictement supérieur	Idem
`>=`	supérieur ou égal	Idem
`==`	égal	« avoir même valeur » *Attention* : deux signes =
`!=`	différent	
`is`	identique	être le même objet
`is not`	non identique	
`in`	appartient à	avec `str`, `list` et `dict`
`not in`	n'appartient pas à	avec `str` et `list` et `dict`

Python

```
>>> a = 2
>>> a == 2
>>> a == 3
>>> a == 2.0
>>> a is 2.0
>>> a != 100
>>> a > 2
>>> a >= 2
```

Python

```
>>> a = 'Alice'
>>> b = 'Bob'
>>> a < b
>>> 'e' in a
>>> 'e' in b
>>> liste = [1,10,100]
>>> 2 in liste
```

Ces opérateurs permettent de réaliser des tests basiques. Pour des tests plus évolués on utilisera des « mots de liaison » logiques.

2. Les connecteurs logiques

- `and` permet de vérifier que 2 conditions sont *vérifiées simultanément*.
- `or` permet de vérifier qu'*au moins une* des deux conditions est vérifiée.
- `not` est un opérateur de *négation* très utile quand on veut par exemple vérifier qu'une condition est fausse.

Voici les tables de vérité des deux premiers connecteurs :

`and`	*True*	*False*
True	`True`	`False`
False	`False`	`False`

`or`	*True*	*False*
True	`True`	`True`
False	`True`	`False`

À ceci on peut ajouter que `not True` vaut `False` et vice-versa.

Python

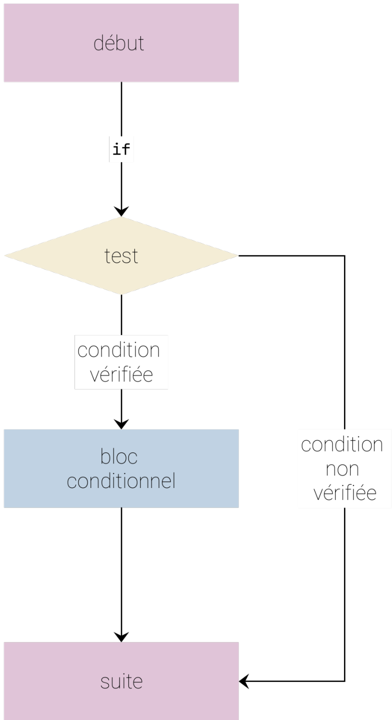
```
>>> True and False
>>> True or False
>>> not True
```

Python

```
>>> resultats = 12.8
>>> mention_bien = resultats >= 14 and resultats < 16
>>> print(mention_bien)
```

3. if, else et elif

Voici le schéma de fonctionnement d'un test `if` :

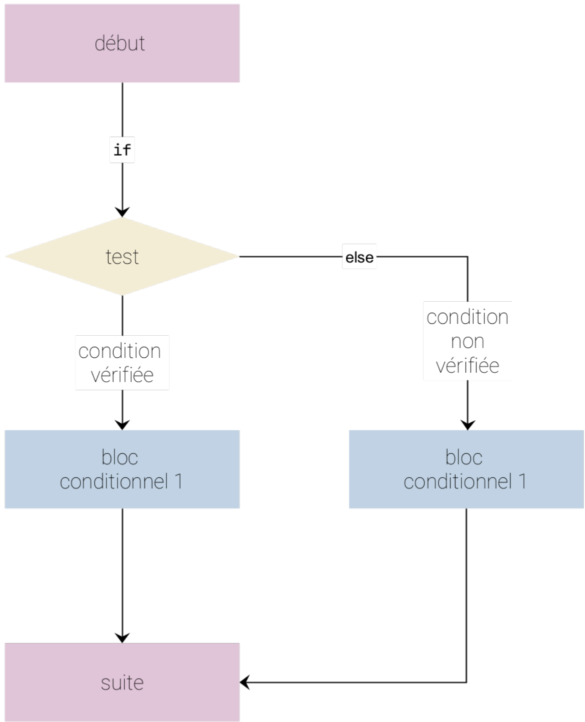


Attention : Un bloc conditionnel doit être *tabulé* par rapport à la ligne précédente : il n'y a ni `DébutSi` ni `FinSi` en PYTHON, ce sont les tabulations qui délimitent les blocs.

Python

```
phrase='Je vous trouve très joli'
reponse = input('Etes vous une femme ?(O/N) : ')
if reponse == 'O':
    phrase += 'e'
phrase +='.'
print(phrase)
```

Voici le schéma de fonctionnement d'un test `if...else` :



Python

```
print('Bonjour')
age = int(input('Entrez votre age : '))
if age >= 18:
    print('Vous etes majeur')
else:
    print('Vous etes mineur.')
print('Au revoir.')
```

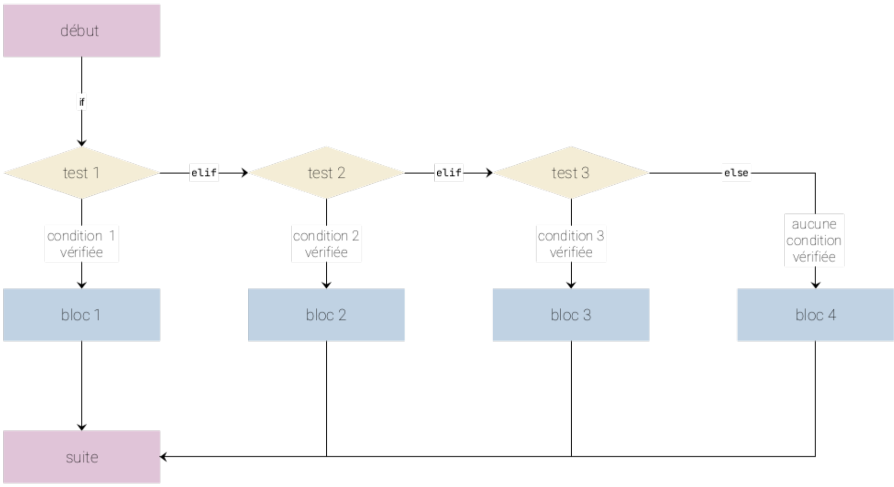
Voici un exemple de fonctionnement d'un test `if...elif...` :

Python

```
print('Bonjour')
prenom = input('Entrez un prénom : ')
if prenom == 'Robert':
    print("Robert, c'est le prénom de mon grand-père.")
elif prenom == 'Raoul':
    print("Mon oncle s'appelle Raoul.")
```

```python
elif prenom == 'Médor':
    print("Médor, comme mon chien !")
else:
    print("Connais pas")
print('Au revoir.')
```

Et voici un schéma décrivant son fonctionnement :



On peut bien sûr inclure autant de `elif` que nécessaire.

4. Exercices

Exercice 54

Écrire un script qui demande son âge à l'utilisateur puis qui affiche `'Bravo pour votre longévité.'` si celui-ci est supérieur à 90.

Exercice 55

Écrire un script qui demande un nombre à l'utilisateur puis affiche si ce nombre est pair ou impair.

Exercice 56

Écrire un script qui demande l'âge d'un enfant à l'utilisateur puis qui l'informe ensuite de sa catégorie :

– trop petit avant 6 ans ;

- poussin de 6 à 7 ans inclus ;
- pupille de 8 à 9 ans inclus ;
- minime de 10 à 11 ans inclus ;
- cadet à 12 ans et plus ;

Exercice 57

Écrire un script qui demande une note sur 20 à l'utilisateur puis vérifie qu'elle est bien comprise entre 0 et 20. Si c'est le cas rien ne se produit mais sinon le programme devra afficher un message tel que `'Note non valide.'`.

Exercice 58

Écrire un script qui demande un nombre à l'utilisateur puis affiche s'il est divisible par 5, par 7 par aucun ou par les deux de ces deux nombres.

Exercice 59

En reprenant l'exercice du chapitre 1 sur les numéros de sécurité sociale, écrire un script qui demande à un utilisateur son numéro de sécurité sociale, puis qui vérifie si la clé est valide ou non.

Exercice 60

Écrire un script qui résout dans **R** l'équation du second degré $ax^2 + bx + c = 0$.
On commencera par `from math import sqrt` pour utiliser la fonction `sqrt`, qui calcule la racine carrée d'un `float`.
On rappelle que lorsqu'on considère une équation du type $ax^2+bx+c = 0$

- si $a = 0$ ce n'est pas une équation de seconde degré ;
- sinon on calcule $\Delta = b^2 - 4ac$ et
  - Si $\Delta < 0$ l'équation n'a pas de solutions dans **R** ;
  - Si $\Delta = 0$ l'équation admet pour unique solution $\frac{-b}{2a}$;
  - Si $\Delta > 0$ l'équation admet 2 solutions : $\frac{-b-\sqrt{\Delta}}{2a}$ et $\frac{-b+\sqrt{\Delta}}{2a}$.

Pour vérifier que le script fonctionne bien on pourra tester les équations suivantes :

- $2x^2 + x + 7 = 0$ (pas de solution dans $\mathbb{R}$) ;
- $9x^2 - 6x + 1 = 0$ (une seule solution qui est $\frac{1}{3}$) ;
- $x^2 - 3x + 2$ (deux solutions qui sont 1 et 2).

Exercice 61

L'opérateur `nand` est défini de la manière suivante : si `A` et `B` sont deux booléens alors

`A nand B` vaut `not (A and B)`

Construire la table de vérité de `nand` en complétant :

A	B	A and B	not (A and B)
`False`	`False`		
`False`	`True`		
`True`	`False`		
`True`	`True`		

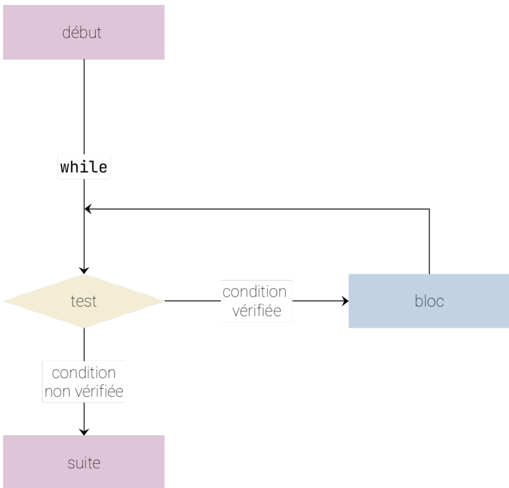
Chapitre 9
Boucles

« Tant que tu n'y arrives pas recommence. »

On s'intéresse dans ce chapitre aux *structures itératives*, plus communément appelée *boucles*.

1. La boucle while

Voici son schéma de fonctionnement :



La boucle `while` exécute un bloc d'instructions conditionnel *tant que* une condition est vérifiée.
Dès que la condition n'est plus vérifiée, le bloc conditionnel n'est plus exécuté.

Python

```
reponse=''
print('Bonjour !')
```

```
while reponse !='n':
    reponse = input('Voulez-vous continuer ? (o/n) : ')
print('Au revoir.')
```

Les boucles `while` doivent être utilisées avec soin : si la condition est toujours vérifiée, le programme ne s'arrêtera pas :

Python

```
while True:
    print('Au secours !')
```

Voici un exemple typique d'utilisation de la boucle `while` :

On place un capital de 2000 euros sur un compte à intérêts annuels de 2%. On aimerait savoir au bout de combien de temps, sans rien toucher, le solde du compte dépassera 2300 euros.

Python

```
 solde = 2000  # solde initial
n = 0  # nombre d'annees
while solde <= 2300:  # condition de boucle
    n += 1  # augmente le compteur d'annees
    solde *= 1.02  # actualise le solde
print(' Il nous faudra ', n, 'ans.')  # affichage final
```

2. La boucle for ... in range(...)

Commençons examiner un nouveau type : `range` (plage de valeurs)

Python

```
>>> a = range(10)
>>> type(a)
>>> print(list(a))
```

Si `range(10)` ressemble beaucoup à la liste `[0,1,...,9]`, la finalité de `range(10)` est d'être un *itérateur*, c'est-à-dire une objet dont on peut parcourir le contenu pour créer une boucle :

Python

```
for i in range(10):
    print(i)
```

La syntaxe complète de `range` est : `range(<debut>,fin,<increment>)`.

Par défaut, si ce n'est pas précisé, `debut=0`, et `increment=1`.

`range(<debut>,fin,<increment>)` renvoie la plage de valeurs suivantes :

– On part de la valeur de début, appelons la `val`

– Tant que `val < fin` :
  - ajouter `val` à la plage
  - ajouter `increment` à `val`

Ainsi, `range(2,52,10)` renvoie la plage de valeurs `2,12,22,32,42`, mais `range(2,53,10)` renvoie la plage de valeurs `2,12,22,32,42,52`.

Très souvent, on se contente d'utiliser une instruction du type `range(n)`, où `n` est de type `int`.

Voici un exemple : Calculons $1 + 2 + ... + 100$:

Python

```
somme = 0
for i in range(1, 101):
    somme += i
print(somme)
```

3. La boucle for ... in ...

On peut généraliser le paragraphe précédent à toute *variable itérable*, c'est extrêmement puissant : les `str`, les `list` et les `dict` sont des types itérables.

Voici des exemples :

Comptons le nombre de voyelles d'une chaîne de caractères :

Python

```python
voyelles = 'aeiouy'  # ensemble de voyelles
phrase = input('Entrez une phrases sans accents :
 ↪  ').lower()  # phrase mise en minuscules
compteur = 0  # comptera les voyelles
for lettre in phrase:  # on parcourt la phrase
    if lettre in voyelles:  # est-ce une voyelle ?
        compteur += 1  # si oui on comptabilise
print('Nombre de voyelles : ', compteur)  # affiche le
 ↪  nombre
```

Faisons la moyenne d'une liste de notes :

Python

```python
liste_notes = [12, 11.5, 13, 18, 13, 11, 9]
moyenne = 0
for note in liste_notes:
    moyenne += note
moyenne /= len(liste_notes)
print(moyenne)
```

Pour le dernier exemple on utilise le type `dict` : soit `a` une variable de ce type :

– `a.keys()` renvoie la liste des clés (des indices du dictionnaire).

– `a.values()` renvoie la liste des valeurs prises par le dictionnaire.

Voici un second programme de moyenne :

Python

```python
resultats = {'EPS': 12, 'maths': 15, 'info': 18}
moyenne = 0
for note in resultats.values():
    moyenne += note
moyenne /= len(resultats)
print(moyenne)
```

4. Quelle boucle utiliser ?

Si la boucle dépend d'une condition particulière on préfèrera la boucle `while`.

Si le nombre d'itérations de la boucle est connu on préfèrera une boucle `for`. On peut utiliser une boucle `for` sur toute *structure itérative*, par exemple une variable de type `range`, `str`, `list` ou, dans une certaine mesure, `dict`.

5. Exercices

Exercice 62

Calculer à l'aide d'un script la somme des carrés des 1000 premiers entiers non nuls.

Exercice 63

Calculer à l'aide d'un script la somme des carrés des 1000 premiers multiples de 3 non nuls.

Exercice 64

Écrire un script qui demande une phrase à l'utilisateur, puis affiche la phrase en rajoutant des tirets.
Exemple : on entre `'Salut à toi'` le script affiche `'S-a-l-u-t- -à- -t-o-i-`.

Exercice 65

Calculer à l'aide d'un script le nombre n à partir duquel la somme $1^2 + 2^2 + ... + n^2$ dépasse un milliard.

Exercice 66

Écrire un script qui demande une phrase et compte le nombre d'occurrences de la lettre « a » dans celle-ci.

Exercice 67

Programmer le jeu du "plus petit plus grand" :

- L'ordinateur choisit un nombre entier au hasard compris entre 0 et 100.
  Au début du script, importer la fonction `randint` du module `random` avec `form random import randint`.
  Pour obtenir un entier au hasard, utiliser `randint(0,100)`.

- L'utilisateur propose un nombre, l'ordinateur répond « gagné », « plus petit » ou « plus grand ».
- Le programme continue tant que l'utilisateur n'a pas gagné.

Exercice 68

On considère la suite s définie par :

$$\begin{cases} s_0 & = & 1000 \\ s_{n+1} & = & 0,99s_n + 1 \quad \text{pour tout } n \in \mathbb{N} \end{cases}$$

- Écrire un script calculant les premiers termes de s (vous décidez le nombre de termes).
- Utiliser ce script pour conjecturer la limite de s.
- Modifier ce script pour obtenir le plus petit entier n tel que l'écart entre s_n et sa limite soit inférieur ou égal à 10^{-4}.

Exercice 69

φ (lettre phi, équivalent du « f » en grec) est défini par :

$$\varphi = 1 + \frac{1}{1 + \frac{1}{1+\frac{1}{\cdots}}}$$

Sur papier, fabriquer une suite par récurrence commençant ainsi :

- 1
- $1 + \frac{1}{1}$
- $1 + \frac{1}{1+\frac{1}{1}}$
- Et cætera (trouver une relation simple pour calculer le terme suivant à partir du terme actuel).

Programmer un script qui calcule successivement les termes de cette suite (aller jusqu'à 10000e).

Comparer avec la valeur exacte de φ, qui est $\frac{1+\sqrt{5}}{2}$.

Exercice 70

Écrire un script qui détermine si un entier est premier ou pas.

Chapitre 10
Fonctions

« Quelle est la fonction de ce chapitre ? »

1. Exemples de fonctions

1.1. Un objet déjà connu

Nous avons déjà rencontré des fonctions *côté utilisateur* :

- `input`

  - prend en entrée une chaîne de caractères ;
  - renvoie la chaîne de caractère saisie par l'utilisateur.

  On peut noter ceci `input(chaine: str) -> str`

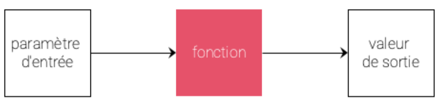
- `len`

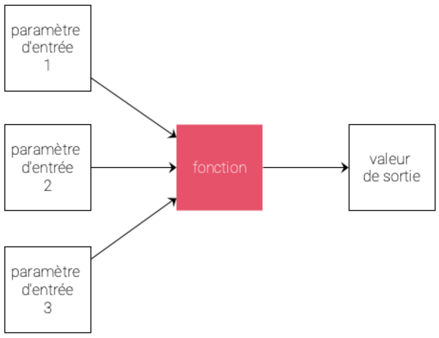
  - prend en entrée une liste ;
  - renvoie le nombre d'éléments de cette liste.

  On peut noter cela `len(lst: list) -> int`

1.2. De multiples formes

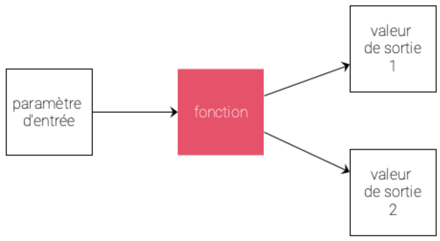
Les deux exemples précédents rentrent dans la catégorie représentée à droite.

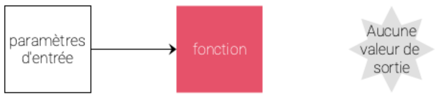




Certaines fonctions sont comme à gauche.
Par exemple `max(20,3,10)` renvoie 20.

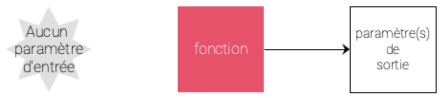
D'autres fonctions sont comme à droite.
On verra des exemples plus tard.





D'autres encore sont comme à gauche.
Par exemple `print("salut")` ne renvoie rien mais affiche `salut` à l'écran.

D'autres suivent le schéma ci-contre.
Par exemple dans le module `time`, la fonction `time` ne prend aucun paramètre d'entrée mais renvoie l'heure qu'indique l'horloge de l'ordinateur.
On peut par exemple l'utiliser pour stocker une heure précise en tapant `maintenant = time()`.



Enfin certaines suivent ce schéma.
Par exemple dans le module `pygame`, `pygame.display.flip` ne prend aucun paramètre d'entrée, ne renvoie aucune valeur, mais actualise la fenêtre graphique.
On l'appelle donc en tapant `pygame.display.flip()`.

Il est possible de créer de nouvelles fonctions.
On parle alors de fonctions *côté concepteur*.

Il faut donc définir rigoureusement ce qu'est une fonction.

2. Définition de la notion de fonction

Définition : fonction

> Une *fonction* est un « morceau de code » qui représente un *sous-programme*.
> Elle a pour but d'effectuer une tâche *de manière indépendante*.

Exemple

> On veut modéliser la fonction mathématique f définie pour tout nombre réel x par
>
> $$f(x) = x^2 + 3x + 2$$
>
> On écrira alors
>
> ```
> def f(x : float) -> float:
>     return x ** 2 + 3 * x + 2
> ```
>
> Pour évaluer ce que vaut $f(10)$ et affecter cette valeur à une variable, on pourra désormais écrire `resultat = f(10)`.

Que fait la fonction `mystere` ?

```
def mystere(a : float, b : float) -> float:
    if a <= b:
        return b
    else:
        return a
```

La fonction `mystere` :

- prend en entrée deux paramètres de type `float` `a` et `b` ;

- renvoie le plus grand de ces deux nombres.

La réponse que l'on vient de formuler s'appelle *la spécification* de la fonction f.

Définition : fonction

Donner la spécification d'une fonction `f` c'est

- préciser le(s) type(s) du (des) paramètre(s) d'entrée (s'il y en a);
- indiquer sommairement ce que fait la fonction `f`;
- préciser le(s) type(s) de la (des) valeur(s) de sortie (s'il y en a).

3. Anatomie d'une fonction

Python

```
def f(lst: list) -> int
    mini = lst[0]
    n = len(lst)
    for i in range(n):
        lst[i] < mini:
        mini = lst[i]
    return mini
```

La fonction `f`

- prend en entrée une liste (sous entendu d'entiers);
- renvoie le plus petit entier de cette liste.

3.1. Paramètre formel

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```

Paramètre formel

Le paramètre d'entrée est *formel* : *le nom de cette variable n'existe qu'à l'intérieur de la fonction*. Si ce nom de variable existe déjà à l'extérieur de la fonction, *ce n'est pas la même variable*.

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```

Type du paramètre d'entrée

Le type du paramètre d'entrée peut être spécifié. Ce n'est pas obligatoire mais très fortement recommandé pour « garder les idées claires ».

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```

Variables locales

Toutes les variables *créées* dans une fonction n'existent *que dans cette fonction*. Elles ne sont pas accessibles depuis l'extérieur de la fonction. On dit que ce sont des *variables locales*.

3.2. valeur de sortie

```
def f(L: list) -> int:
    mini = L[0]
    n = len(L)
    for i in range(n):
        if L[i] < mini:
            mini = L[i]
    return mini
```

Type de la valeur de sortie

Le type de la valeur de sortie peut être précisé, c'est également recommandé.

4. En pratique

4.1. Des exemples

Python

```
1   def f(x : float) -> float:
2       return x ** 2 + 3 * x + 2
3
4   print(f(1)) # Affiche 6
```

Le programme commence à la ligne 4 !
Les 2 premières lignes servent à définir la fonction `f`, elles ne sont exécutées que lorsqu'on évalue `f(1)`.

Python

```
def f(x : float) -> float:
    return x ** 2 + 3 * x + 2

print(x) # Provoque une erreur
```

L'erreur vient du fait que la variable `x` *n'est pas définie*. Le « `x` qu'on voit dans la fonction `f` » est un paramètre formel et n'existe que dans `f`.

Python

```
def f(x : float) -> float:
    a = 2
    return x + a

print(a) # Provoque une erreur
```

L'erreur vient du fait que la variable `a` *est locale* : elle n'est définie que durant l'exécution de `f`.

Python

```
def f(x : float) -> float:
    a = 2
    return x + a
```

```python
print(f(4)) # Affiche 6
print(a) # Provoque une erreur
```

C'est encore la même erreur : une fois `f(4)` évaluée, `a` n'existe plus.

Python

```python
1 def f(x : float) -> float:
2     a = 2
3     return x + a
4
5 a = 3
6 print(f(4)) # Affiche 6
7 print(a) # Affiche 3 et pas 2
```

La variable `a` définie dans la fonction `f` n'est pas la même que celle qui est définie à la ligne 5.
Celle définie à la ligne 2 est *locale*.
La variable `a` de la ligne 5 est appelée *globale*.

Python

```python
def f(x : float) -> float:
    return x + a

a = 3
print(f(4)) # Affiche 7
```

À retenir

Une fonction a le droit d'*accéder en lecture* à une variable globale, mais n'a pas *a priori* le droit d'en modifier la valeur.

4.2. À éviter autant que possible

Python

```python
1 def f(x : float) -> float:
2     global a
3     a = a + 1
4     return x + a
5
6 a = 3
7 print(f(4)) # Affiche 8
8 print(a) # Affiche 4
```

À la ligne 2, on signale à Python que `f` a la droit de modifier la variable globale `a`. C'est fortement déconseillé : sauf si on ne peut pas faire autrement, une fonction ne doit pas modifier les variables globales.

Table des matières