

Partie I

Représentation

de l'information

Chapitre 1

Bases de numération

«Partons sur de bonnes bases.»

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

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^{er} millénaire av. J.-C., les Babyloniens utilisaient la base soixante pour mesurer le temps et les angles;
- durant le I^{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^e et le XV^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

Comprendons 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 connaît le binaire et travaille avec des valeurs entières de type `int` (`integer` veut dire « entier » en Anglais, pour plus de précisions, voir le chapitre 5, partie 2).

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 centaines.

Puisqu'on a trouvé un quotient de 0, on s'arrête.

On peut écrire cela simplement :

$$\begin{array}{r} 203 \\ \hline 3 | 20 \\ \hline 0 | 2 \end{array}$$

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

$$\begin{array}{r} 203 \\ \hline 1 | 101 \\ \hline 1 | 50 \\ \hline 0 | 25 \\ \hline 1 | 12 \\ \hline 0 | 6 \\ \hline 0 | 3 \\ \hline 1 | 1 \\ \hline 1 | 0 \end{array}$$

On a donc successivement établi :

$$203 = 101 \times 2 + 1$$

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

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 1230 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

$$n = a_0 16^0 + a_1 16^1 + \dots + a_k 16^k$$

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 $(D4A)_{16}$:

$$\begin{aligned} (D4A)_{16} &= 13 \times 16^2 + 4 \times 16^1 + 10 \times 16^0 \text{ car D vaut 13 et A vaut 10.} \\ &= 3402 \end{aligned}$$

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

Déterminons maintenant l'écriture hexadécimale de 503 :

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

$31 = 1 \times 16 + 15$ et 15 s'écrit F.

$1 = 0 \times 16 + 1$ et on arrête car le quotient est nul.

$503 = (1F7)_{16}$.

0x000340:	C6 10 80 E3 02 10 01 E0 B0 10 C3 E1 00 30 0F E1	À.Íã...à°.Ãá.O.á
0x000350:	DF 30 C3 E3 1F 30 83 E3 03 F0 29 E1 3C 10 9F E5	ßØÃá.0Iã.ð)ák..Iâ
0x000360:	0C 10 81 E0 00 00 91 E5 00 40 2D E9 00 E0 8F E2	.. à..’å..@-é.à à
0x000370:	10 FF 2F E1 00 40 BD E8 00 30 0F E1 DF 30 C3 E3	.ý/á..@½è.0.áß0Ãá
0x000380:	92 30 83 E3 03 F0 29 E1 0F 40 BD E8 B0 20 C3 E1	’0Iã.ð)á..@½è°.Ãá
0x000390:	B8 10 C3 E1 00 F0 69 E1 1E FF 2F E1 28 73 00 03	, Ãá..ðiá..ý/á(s..
0x0003A0:	90 34 00 03 F0 B5 47 46 80 B4 FF 20 E1 F1 FA FD	4..ðµGFÍ..ý áñúý
0x0003B0:	A0 21 C9 04 27 4A 10 1C 08 80 00 F0 D3 FA 26 49	!É.'J...I..ðóú&I
0x0003C0:	26 4A 10 1C 08 80 00 F0 F9 F8 00 F0 5B F9 DB F1	&J...I..ðúá.ð[úÜñ
0x0003D0:	1B FA 00 F0 DF F8 F8 F0 1F FB 4B F0 73 FE 00 F0	.ú..ðßøøð.úKðsp.ð
0x0003E0:	6F F8 71 F0 BB FA 00 F0 07 FC 00 F0 1B FE 1C 48	oøqð»ú.ð.ü.ð.p.H
0x0003F0:	E0 21 49 02 02 F0 7A FB F7 F0 90 FC 19 48 00 24	à!I..ðzû=ð ü.H.\$
0x000400:	04 70 19 48 04 70 F5 F0 1F F8 18 48 04 70 18 4F	.p.H.pðð.ø.H.p.ø
0x000410:	00 21 88 46 06 1C 00 F0 E5 F8 12 48 00 78 00 28	.!IF...ðåø.H.x.(
0x000420:	0E D1 39 8D 01 20 08 40 00 28 09 D0 20 08 40	.ñ9 . .@. (. ð. .@
0x000430:	0E 28 05 D1 DE F1 B8 FE DE F1 48 FE 00 F0 4A FA	.(.Ñþñ,þþñHþ.ðJú
0x000440:	57 F0 BE FF 01 28 15 D1 30 70 00 F0 2F F8 00 20	Wðñy. (.Ññp.ð/ø.
0x000450:	30 70 22 E0 FF 7F 00 00 04 02 00 04 14 40 00 00	Op"àýI.....@..
0x000460:	00 00 00 02 80 34 00 03 1C 5E 00 03 34 30 00 03I4...^..40..
0x000470:	40 30 00 03 0C 4D 00 20 28 70 00 F0 17 F8 57 F0	@0...M. (p.ð.øWð
0x000480:	69 FF 04 1C 01 2C 08 D1 00 20 F8 85 06 F0 16 FF	iý....,Ñ. øI.ð.ý
0x000490:	2C 70 00 F0 0B F8 42 46 2A 70 54 F0 57 FA 71 F0	.p.ð.øBF*pTðWúqð
0x0004A0:	67 FA 00 F0 F3 F9 B6 E7 34 30 00 03 00 B5 0A F0	gú.ðóù¶ç40...µ.ð
0x0004B0:	19 FE 00 06 00 28 01 D1 00 F0 28 F8 01 BC 00 47	.p...(.Ñ.ð(ø.4.G
0x0004C0:	10 B5 0B 48 00 24 04 62 44 62 04 60 09 48 00 F0	.µ.H.\$.bDb. ` .H.ø
0x0004D0:	37 F8 09 48 09 49 01 60 09 4A 0A 48 10 60 F2 20	7ø.H.I.`.J.H.`ø
0x0004E0:	00 01 09 18 0C 60 08 48 04 70 10 BC 01 BC 00 47` .H.p.4.4.G

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	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3	D0
E	E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2	E0
F	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

$$(F7B)_{16} = \left(\underbrace{1111}_{F} \underbrace{0111}_{7} \underbrace{1011}_{B} \right)_2$$

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

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

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

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

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 $n \in \mathbb{N}$, 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 :

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

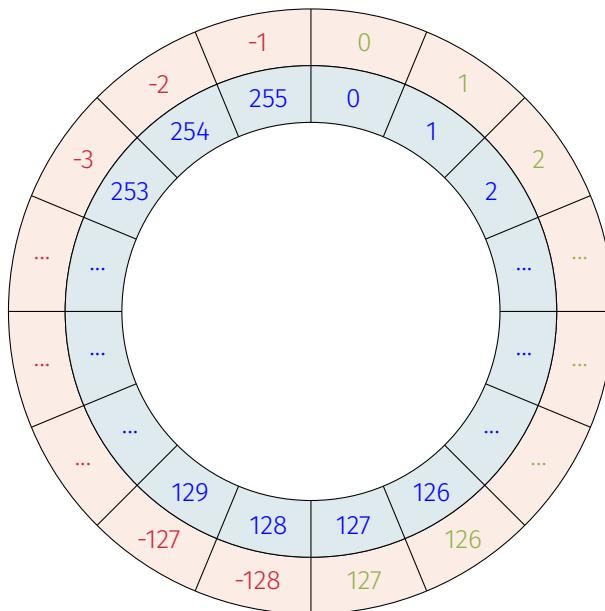
On obtient 10001 1110, qui représente -30... On aurait bien sûr préféré que cela nous donne 24.

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 8^e) 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 :

$$\begin{array}{r} 00101\ 100 \\ \underbrace{}_{\text{on change}} \quad \underbrace{}_{\text{on garde}} \end{array}$$

Ce qui nous donne

$$\begin{array}{r} 11010\ 100 \\ \underbrace{}_{\text{on a changé}} \quad \underbrace{}_{\text{on a gardé}} \end{array}$$

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 :

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

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 $140 - 128 = -16$. 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 $32\ 767$ (-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 = 1000 octets (fichiers textes)
- 1 mégaoctet = 1 Mo = 1000 ko (fichiers .mp3)
- 1 gigaoctet = 1 Go = 1000 Mo (RAM des PC actuels, jeux)
- 1 téraoctet = 1 To = 1000 Go (Disques durs actuels)
- 1 pétaoctet = 1 Po = 1000 To
- 1 exaoctet = 1 Eo = 1000 Po = 10^{18} octets (trafic internet mondial mensuel en 2022 : 376 Eo)
- 1 zétaoctet = 1 Zo = 1000 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 $\pi \approx 3,141\,592\,653\,589\,793$.

Ceci dit on peut donner une valeur approchée d'un nombre réel r en écriture décimale en utilisant l'*écriture scientifique* vue au collège :

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

- 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 :

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

Dans ce cas, le nombre d comporte 10 chiffres décimaux, appelés *chiffres significatifs*.

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 droite, 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 $e_7e_6 \dots e_0$ servent à coder l'exposant e de 2, qui vaut

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

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

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

Ce qu'on peut noter

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

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

Exemple : des 32 bits au nombre

Que représente 1 0100 0011 1110 0000 0000 0000 0000 0000?

- $s = 1$.
- $e = (0100 0011)_2 - 127 = 67 - 127 = -60$
- $m = ([1], 1110000000000000000000000000)_2 = 1 + 2^{-1} + 2^{-2} + 2^{-3} = 1,875$
- $1 0100 0011 1110 0000 0000 0000 0000 \rightsquigarrow (-1)^1 \times 1,875 \times 2^{-60}$

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 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 :

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

C'est l'équivalent dyadique de

$$\frac{1}{3} = (0,333\dots)_{10}$$

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 :

```
0.100000000000000055511151231257827021181583404541015625
```

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

- $(101,1)_2$
- $(1,011)_2$
- $(0,1111\ 111)_2$ en remarquant que c'est « $(111\ 1111)_2$ divisé par 2^7 .

Exercice 17

Écrire en base 2 les nombres suivants :

- 3,5
- 7,75
- 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.

- Quel est le plus grand nombre positif représentable ?
- 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.

- Quel est le flottant immédiatement plus grand que 1? Quelle distance les sépare ?
- Quel est le flottant immédiatement plus grand que le plus petit nombre positif représentable? Quelle distance les sépare ?
- 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 $\frac{1}{3}$. On note donc

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

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

$$\frac{1}{3} = (0, 0101 0101 \dots)_2$$

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 **1**111 0011, ce bit de poids fort jouant le rôle de *code correcteur*.
- Un autre exemple : 001 1110 est codé **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)																ISO/CEI 8859-15																	
x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF	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	0x																
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	1x																
2x	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	2x	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	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	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	[\	^	_	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	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	7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	
8x	€			,	f	"	...	†	‡	^	%	Š	‘	Œ	Ž	8x																	
9x				,	"	"	"	•	-	-	~	™	š	>	œ	ž	9x																
Ax	NBSP	i	ç	£	¤	¥	:	§	-	©	ª	«	¬	®	-	Ax	i	ç	£	€	¥	Š	§	š	©	ª	«	¬	®	-			
Bx	°	±	²	³	’	μ	¶	·	,	¹	º	»	¼	½	¾	ü	Bx	°	±	²	³	’	ž	μ	¶	·	ž	¹	º	»	œ	ÿ	ü
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Í	Í	Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Í	Í
Dx	Ð	Ñ	Ò	Ó	Ô	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	Þ	Dx	Ð	Ñ	Ò	Ó	Ô	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	Þ		
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	í	í	í	í	Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	í	í	í	
Fx	ð	ñ	ò	ó	ô	ö	÷	ø	ù	ú	û	ü	ý	þ	ý	Fx	ð	ñ	ò	ó	ô	ö	÷	ø	ù	ú	û	ü	ý	þ	ý		

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_5601-1987, 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 8365^{ème} 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 caractères 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, presque 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	110x xxxx 10xx xxxx
de 12 à 16	3	1110 xxxx 10xx xxxx 10xx xxxx
de 17 à 21	4	1111 0xxx 10xx xxxx 10xx xxxx 10xx 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 : **[00]10 0000 1010 1100**.
- On formate comme à la 3ème ligne du tableau :

1110 0010 1000 0010 1010 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 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()
```

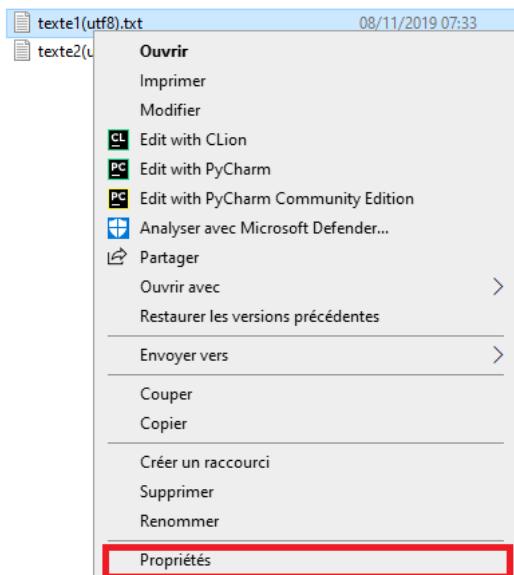
Permet d'écrire un fichier texte (d'où le '`wt`', pour write text) en UTF-8.

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 « Ă© » .

- 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

Programmation

avec Python

Chapitre 5

Valeurs et types

1. Python, machine à évaluer

PYTHON est tout d'abord une machine à calculer, ou plutôt une machine à évaluer : lorsque PYTHON rencontre une *expression*, c'est-à-dire une écriture qui produit une valeur, il commence par déterminer cette valeur.

Pour s'en rendre compte, il suffit d'écrire des expressions dans une *console*.

Python

```
>>> 5 - 3  
2  
  
>>> 11 / (1 + 2)  
3.6666666666666665  
  
>>> 30 / 15  
2.0
```

On se rend compte que les valeurs rencontrées ne sont pas présentées de la même manière : `5 - 3` a la valeur `2` alors que `30 / 15` a la valeur `2.0`.

Pour y voir plus clair, on peut appeler la fonction `type` qui

- en entrée prend une expression;
- renvoie le *type* de l'expression.

Python

```
>>> type(2)  
<class 'int'>  
  
>>> type(2.0)
```

```
<class 'float'>
```

Il y a donc au moins deux types de valeurs, le type `int` et le type `float`. En fait il existe une multitude de types prédéfinis selon la nature de la valeur à représenter et nous allons les passer en revue.

2. Le type `int`

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

Python

```
>>> 3 + 2
5

>>> 2 * 3
6

>>> 3 - 2 * 2
-1

>>> 10_000 # on utilise des _ pour séparer les chiffres
```

On dispose également de *deux opérations très pratiques* : soient `a` et `b` deux `int`, et `b` non nul, alors on

- `a // b` est le *quotient* de la *division euclidienne* de `a` par `b`;
- `a % b` est le *reste*.

Python

```
>>> 64 // 10 # 64, c'est 6 * 10 + 4
6

>>> 64 % 10
4

>>> 22 // 7 # 22, c'est 3 * 7 + 1
```

```
3  
  
>>> 22 % 7  
1
```

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  
10000  
  
>>> 2 ** (-1)  
0.5
```

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

3. Le type float

Il 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

```
>>> 2 / 7  
0.2857142857142857 # c'est une valeur approchée  
  
>>> 1 / 100_000  
1e-05  
  
>>> 1.2e-4  
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 module¹ `math`:

Python

```
>>> from math import *
>>> pi
3.141592653589793

>>> cos(pi / 3)
0.5000000000000001

>>> exp(2)
7.38905609893065

>>> log(2)
0.6931471805599453

>>> exp(log(2))
2.0
```

`exp` est la fonction exponentielle et `log` la fonction logarithme népérien² notée `ln` en France.

4. Le type str

Il sert à représenter les *chaînes de caractères* (`str` est l'abréviation de *string*, qui veut dire chaîne en anglais). Lorsqu'on écrit une valeur de type `str`, on peut utiliser les symboles `', "` ou même `'''` (suivant que la chaîne contient des apostrophes, ou des guillemets).

Python

```
>>> 'Bonjour.'
'Bonjour'

>>> 'J'aime Python.'
SyntaxError
```

1. Un module est un ensemble de *fonctions* et/ou de *constantes* que l'on peut importer.
 2. Voir le programme de mathématiques de terminale scientifique.

```
>>> "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".'
```

La dernière évaluation produit une valeur correcte. PYTHON utilise simplement \ ' pour écrire les apostrophes qui sont à l'intérieur de la valeur.

Le symbole + sert à *concaténer 2 chaînes*, c'est-à-dire à les mettre bout à bout.

Python

```
>>> 'Yes' + 'No'  
'YesNo'  
  
>>> 'No' + 'Yes'  
'NoYes'
```

On peut même multiplier un `str` par un `int` :

Python

```
>>> 3*'Aïe ! '  
'Aïe ! Aïe ! Aïe ! '
```

PYTHON évalue `3*'Aïe ! '` comme `'Aïe ! ' + 'Aïe ! ' + 'Aïe ! '`.
Voici deux types très utiles que nous étudierons en détail plus tard.

5. Le type list

Une valeur de type `list` est une... liste ordonnée de valeurs. Celles-ci peuvent être du même type ou non.

Python

```
>>> [] # liste vide  
[]
```

```
>>> [1, 4, 5] # liste comportant 3 int
[1, 4, 5]

>>> [2.0, -4, 'Bonjour', 'Coucou']
[2.0, -4, 'Bonjour', 'Coucou']
```

L'intérêt de ce type est de rassembler plusieurs valeurs au sein d'une seule, qui pourra ensuite être *parcourue*.

6. Le type dict

Celui-ci sert à établir des *association* du type *clé : valeur*.

Python

```
>>> {} # dictionnaire vide
{}

>>> {'France': 'Paris', 'Canada': 'Ottawa', 'Pays-Bas':
    ↪ 'La Haye'}
{'France': 'Paris', 'Canada': 'Ottawa', 'Pays-Bas': 'La
    ↪ Haye'}
```

Tout comme le type `list`, ce type sert à *structurer les données*. L'exemple précédent fait correspondre des capitales à des pays.

7. Le type bool

Il sert à représenter les *valeurs booléennes*, valant `True` (vrai) ou `False` (faux).

Python

```
>>> False
False

>>> True
True
```

Cela peut paraître un peu pauvre, c'est trompeur : les *expressions logiques* sont des écritures dont la valeur est un booléen. Lorsque PYTHON les rencontre, il les évalue pour trouver soit **True** soit **False**.

Python

```
>>> 3 >= 2 # évalue si 3 est supérieur ou égal à 2
True

>>> 3 + 5 == 2 # évalue si 3 + 5 vaut 2
False

>>> 1 in [3, 4, 1, 5] # évalue si 1 est un élément de la
→ liste
True
```

Remarque

Pour tester si deux valeurs sont égales, on utilise `==` (et pas `=`).

Ce type dispose d'*opérations logiques* : **or** (ou), **and** (et) et **not** (non).

Python

```
>>> True and False # vrai que si les 2 sont vrais
False

>>> True or False # faux que si les 2 sont faux
True

>>> not (3 < 1) # contraire
True

>>> (2 < 1) or (3 >= 0)
True
```


Chapitre 6

Variables et affectations

1. Le symbole =

En mathématiques, le symbole = a plusieurs significations

- dans $2 + 2 = 4$, on peut comprendre = comme un opérateur d'évaluation :
 $2 + 2$, cela «donne» 4;
- dans $\mathcal{P} = 2 \times (\ell + L)$, on peut considérer que = sert à définir ce qu'est le périmètre d'un rectangle de dimensions ℓ et L ;
- dans $3x + 2 = 4x + 5$, le = sert à convenir que les 2 membres ont la même valeur et on cherche s'il existe un ou des nombres x qui satisfont l'égalité (appelée équation);
- *et cætera.*

En PYTHON, le symbole = n'a qu'un seul sens : il sert à l'*affectation*.

2. L'affectation

Il s'agit de « stocker » une valeur dans un endroit de la mémoire auquel PYTHON donne un nom¹. Voici un exemple d'affectation :

```
a = 2
```

- 2 est une *valeur* de type `int`;
- la *variable* a est créée;
- a est « attachée » à la valeur 2;
- par extension a est également de type `int`.

¹. En réalité c'est plus compliqué mais cela ne nous intéresse pas.

Au cours d'un programme la valeur associée à une variable peut changer...D'où le nom de *variable*.

Définition : affectation

Lors d'une affectation

- d'abord PYTHON évalue ce qu'il y a à droite du symbole =;
- ensuite il affecte cette valeur à la variable qui figure à gauche du symbole =;
- si la variable n'existe pas déjà, elle est créée automatiquement;
- le type de la variable, c'est le type de la valeur qu'on lui affecte.

Que fait le programme suivant ?

Python

```
x = 0  
x = x + 1  
print(x)
```

- il crée une variable `x` de type `int` valant `0`;
- il évalue `x + 1`, trouve `1` et affecte cette valeur à `x`;
- évalue `x`, trouve `1` et donc affiche `1`.

À retenir

En mathématiques, $x = x + 1$ est une équation sans solution.

En PYTHON, l'instruction `x = x + 1` sert à augmenter la valeur de `x` de `1` (on dit aussi *incrémenter*).

2.1. 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 # permet d'échanger a et b
>>> a
2

>>> b
10
```

2.2. 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 `**=`.

3. Le cas des variables de type str ou list

3.1. Les str

Les valeurs de type `str` sont composées de caractères *alphanumériques*. On peut accéder à chacun d'eux de la manière suivante :

Python

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

Voici comment PYTHON représente la chaîne 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]  
'?'
```

3.2. Les list

Cela se passe un peu comme pour les `str`

Python

```
>>> lst = [3, 4, 8]
>>> lst[1] # élément d'indice 1 de lst
4

>>> lst[-1] # dernier élément de lst
8

>>> len(lst)
3
```

Lorsqu'on essaie d'accéder à un élément dont l'indice est supérieur ou égal à la longueur de la liste, on obtient une erreur. Dans l'exemple précédent si on évalue `lst[3]` on obtient :

`IndexError: list index out of range`

Une étude détaillée des `list` est faite au chapitre 9.

Chapitre 7

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 <code>int</code> et <code>float</code> , lexicographique sur <code>str</code> ...
<=	inférieur ou égal	Idem
>	strictement supérieur	Idem
>=	supérieur ou égal	Idem
==	égal	«avoir même valeur» Attention : deux signes =
!=	différent	
<code>is</code>	identique	être le même objet
<code>is not</code>	non identique	
<code>in</code>	appartient à	avec <code>str</code> , <code>list</code> et <code>dict</code>
<code>not in</code>	n'appartient pas à	avec <code>str</code> et <code>list</code> et <code>dict</code>

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 : white

<code>and</code>	<code>True</code>	<code>False</code>	<code>or</code>	<code>True</code>	<code>False</code>
<code>True</code>	<code>True</code>	<code>False</code>	<code>True</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False</code>	<code>False</code>	<code>True</code>	<code>True</code>

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

Python

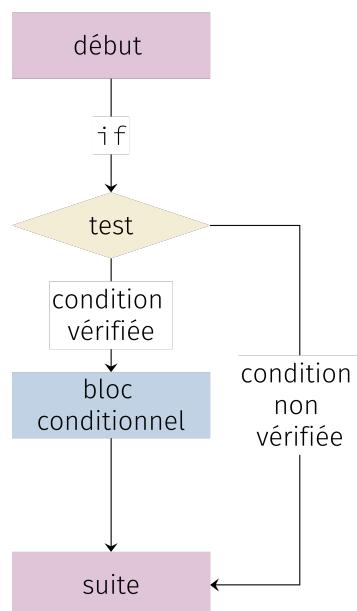
```
>>> True and False
>>> True or False
>>> not True
```

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` :



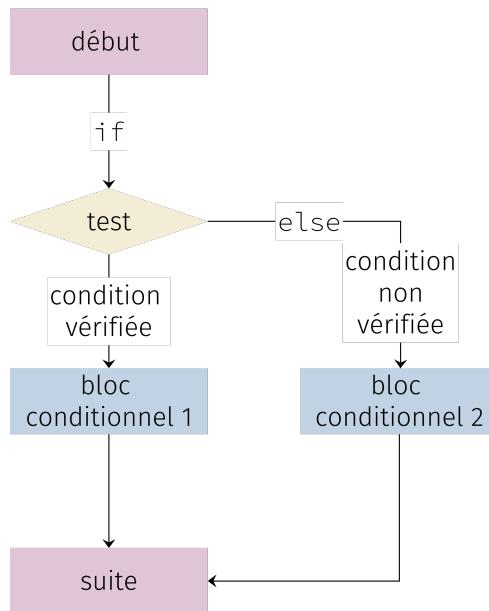
Attention : Un bloc conditionnel doit être *tabulé* par rapport à la ligne précédente : il n'y a ni `DébutSi` ni `FinSi` en PYTHON, ce sont les tabulations qui délimitent les blocs.

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...`:

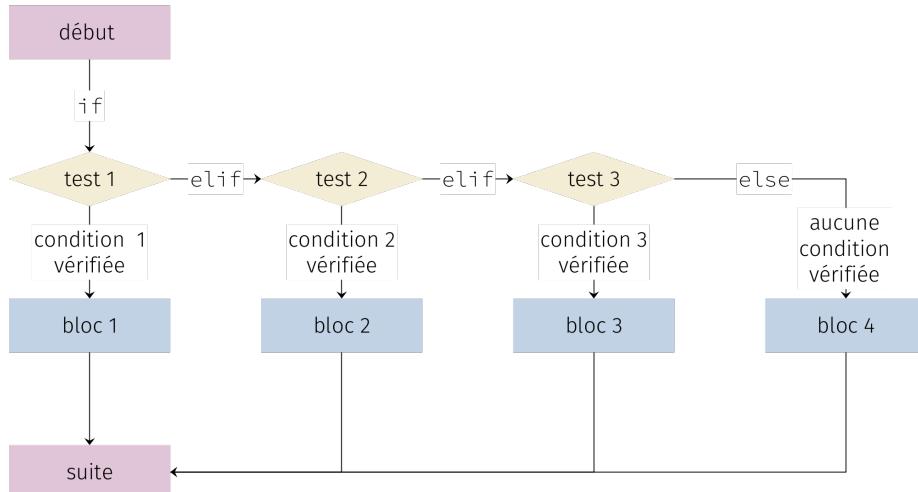
Python

```

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.")
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 35

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 36

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

Exercice 37

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 38

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 39

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 40

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 41

Écrire un script qui résout dans \mathbb{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 \mathbb{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 42

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

$$A \text{ nand } B \text{ vaut } \text{not} \ (A \text{ and } B)$$

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 8

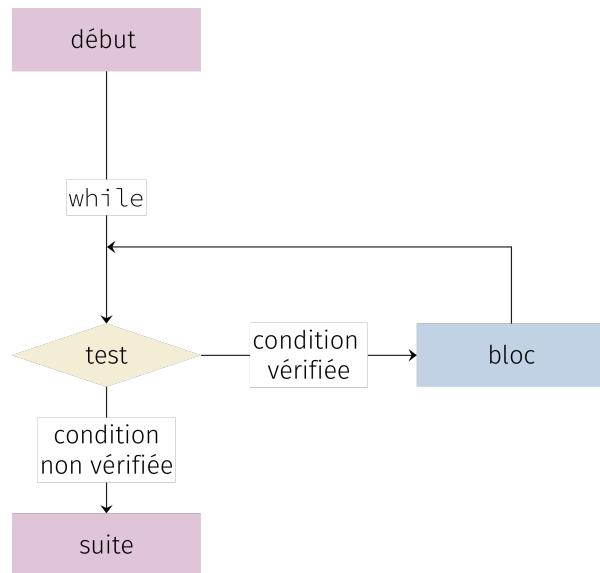
Boucles

« 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.')
```

La boucle `while` doit être utilisée 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 par 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 $1 + 2 + \dots + 100$:

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

```
voyelles = 'aeiouy' # ensemble de voyelles
phrase = input('Entrez une phrase 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

```
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

```
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 43

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

Exercice 44

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

Exercice 45

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 46

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

Exercice 47

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

Exercice 48

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 49

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 50

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

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

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 10000^e).

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

Exercice 51

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

Chapitre 9

Listes

Le type `list` permet de stocker des valeurs dans un ordre précis.

Python

```
# on crée une liste avec 3 valeurs
lst = ['bonjour', 3.14, True]
```

Une valeur de type `list` est itérable :

- on peut accéder à un élément de la liste, par exemple `lst[0]` ;
- on peut parcourir une liste.

Pour accéder à un élément d'une liste situé à un endroit précis, on doit connaître son *indice* : le premier élément d'une liste a l'indice zéro, le deuxième l'indice 1, et *cætera*.

Python

```
# on crée une liste avec 3 valeurs
>>> lst = ['bonjour', 3.14, True]
# premier élément : indice 0
>>> lst[0]
'bonjour'

# deuxième élément
>>> lst[1]
3.14
```

1. Opérations de base

1.1. Créer une liste

- `lst = list()` crée une liste vide;
- `lst = []` fait la même chose;
- `lst = ['a', 7, True]` crée une liste composée de 3 éléments.

Une liste peut contenir des éléments de plusieurs types mais en pratique on évite cela.

1.2. Modifier un élément

Le type `list` est *mutable* : on peut changer un ou des éléments d'une liste sans changer la liste elle-même.

Python

```
>>> lst = [2, 3, 4, 1]
# on change le deuxième élément
>>> lst[1] = 10
>>> lst
[2, 10, 4, 1]
```

1.3. Ajouter un élément en fin de liste

On reprend l'exemple précédent

Python

```
>>> lst.append(7) # ajoute 7 à la fin de la liste
>>> lst
[2, 10, 4, 1, 7]
```

Remarque

`lst = lst + [7]` a le même effet que `lst.append(7)` : on crée une « mini-liste » [4], on concatène les 2 listes et on remet le résultat dans `lst`.

En pratique la première méthode est la plus simple et aussi la plus rapide.

1.4. Insérer un élément à une place donnée

Pour une liste `lst` valant [2, 10, 4, 1], si on veut insérer la valeur 5 à l'indice 1 on écrira :

```
lst.insert(1, 5)
```

et `lst` vaudra [2, 5, 10, 4, 1]

La syntaxe est `lst.insert(indice, valeur)`

1.5. Retirer un élément à une position donnée

Si une liste `lst` a pour valeur [3, 7, 1] et qu'on veut supprimer son deuxième élément alors on écrit :

```
del lst[1]
```

Ensuite, `lst` aura la valeur [3, 1].

1.6. Retirer une valeur précise

Pour retirer une valeur *qui appartient à une liste* on procède ainsi :

Si `lst` a la valeur [1, 2, 5, 4, 2, 3] alors l'instruction

```
lst.remove(2)
```

Supprime la *première occurrence* de 2 dans `lst`.

Après cela, `lst` a la valeur [1, 5, 4, 2, 3].

1.7. Concaténer des listes

On peut procéder de 2 manières :

- `lst1.extend(lst2)` ajoute les éléments de la liste `lst2` à la fin de `lst1`;
- `lst1 = lst1 + lst2` crée une liste avec les éléments de `lst1` et ceux de `lst2`, puis replace le résultat dans `lst1`.

En pratique la première méthode est plus rapide.

1.8. Longueur d'une liste

La fonction `len`

- prend en entrée une liste lst;
 - renvoie la longueur de cette liste.

Ainsi `len([2, 3, 4])` vaut 3.

1.9. Divers

`lst.sort()` trie la liste dans l'ordre croissant.

`lst.reverse()` met les éléments dans l'ordre inverse.

2. Opérations avancées

2.1. Copier une liste (mauvaise méthode)

Python

```
>>> lst1 = [5, 6, 8]
>>> lst2 = lst1
>>> lst1[0] = 10
>>> lst1
[10, 6, 8]

>>> lst2
[10, 6, 8] # problème : lst2[0] a changé aussi !
```

Ce comportement «étrange» vient du fait que le type `list` est *mutable*. Nous allons expliquer cela plus tard dans ce chapitre.

2.2. Copier une liste (bonne méthode)

Python

```
>>> lst1 = [5, 6, 8]
>>> lst2 = lst1[:] # on copie tous les éléments de lst1
   ↵ dans lst2
>>> lst1[0] = 10
```

```
>>> lst1
[10, 6, 8]

>>> lst2
[5, 6, 8] # Ouf !
```

2.3. Extraire une sous-chaîne

Soit `lst` une liste de longueur n et p et q deux entiers tels que $0 \leq p < q \leq n$.
Alors

- `lst[p:q]` est la liste composée des éléments `lst[p], ..., lst[q-1]`;
- `lst[:q]` signifie `lst[0:q]`;
- `lst[p:]` signifie `lst[p:n]`.

Exemple

Si `lst` vaut `[2, 5, 3, 4, 9, 2, 5]` alors

- `lst[2:6]` vaut `[3, 4, 9, 2]`;
- `lst[3:]` vaut `[4, 9, 2, 5]`;
- `lst[:2]` vaut `[2, 5]`.

3. Parcourir une liste

3.1. Parcours selon les indices

Définition : parcours selon les indices

Soit `lst` une liste de longueur n .

Alors ses éléments sont `lst[0], ..., lst[n-1]` et on parcourt la liste en

- considérant un entier i qui jouera le rôle d'*indice*;
- faisant parcourir à i la plage de valeurs `range(n)`;
- considérant les `lst[i]`.

Exemple : un parcours selon les indices

On affiche les éléments d'une liste grâce à un parcours par les indices.

```
lst = [54, 65, 123]
n = len(lst) # n vaut 3
for i in range(n): # range(3), c'est 0, 1, 2
    print(lst[i])
```

Le parcours d'une liste par les indices est *crucial* lorsque lors du parcours, on veut savoir à quelle place on se trouve dans la liste. C'est le cas quand on veut déterminer si une liste est triée dans l'ordre croissant ou non : il faut regarder si chaque élément est plus petit que le suivant dans la liste.

C'est aussi le cas quand on veut par exemple déterminer l'indice de la première apparition d'une valeur dans une liste.

3.2. Parcours selon les éléments

C'est plus simple que le parcours selon les indices mais on pert un peu d'information car pendant le parcours, on ne sait pas à quelle place on se trouve dans la liste.

Définition : parcours selon les éléments

Le parcours des éléments d'une liste `lst` s'effectue à l'aide d'une simple boucle `for x in lst`. `x` prend alors successivement les valeurs de chacun des éléments de `lst`, dans l'ordre.

Exemple : un parcours selon les éléments

```
lst = [54, 65, 123]
for x in lst:
    print(x)
```

3.3. Bilan

On peut toujours utiliser un parcours de liste selon les indices. On peut toujours transformer un parcours selon les éléments en un parcours selon les indices. Le contraire est faux si l'on a *absolument* besoin de savoir quels sont les indices des éléments que l'on examine lors du parcours.

À éviter absolument

Les codes comme celui-ci :

```
for i in lst:  
    print(i)
```

On a l'impression que `i` est un indice mais c'est une valeur, et l'expérience prouve que dans 90% des cas, une erreur du type `lst[i]` survient, alors que...`i` n'est pas un indice ici !

De même les codes comme celui-là :

```
for x in range(len(lst)):  
    print(lst[x])
```

Là encore il y a beaucoup de chances que l'indice `x` soit pris pour une valeur.

Conseil

Réserver les noms de variables `i`, `j` et `k` pour les indices et `x`, `y` et `z` pour les éléments.

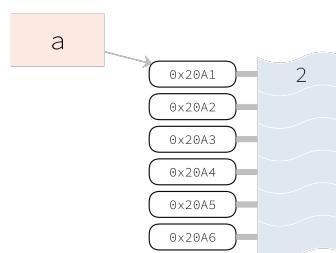
4. Mutabilité

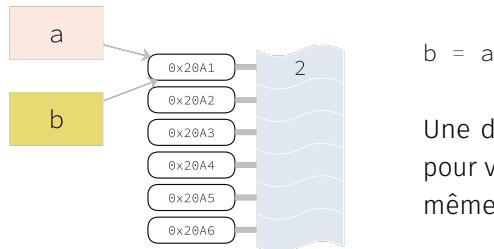
Examinons la différence entre un type *non mutable* tel que `int` et le type `list`, qui est *mutable*.

4.1. Variables de type non-mutable

`a = 2`

La valeur 2 est stockée en mémoire et une variable `a` est créée, associée à cette valeur.

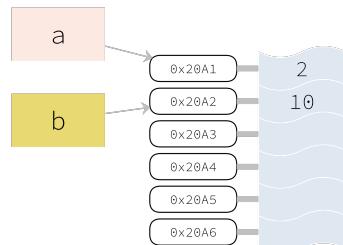


`b = a`

Une deuxième variable `b` est créée, avec pour valeur 2 également. Elles partagent la même adresse-mémoire.

`b = 10`

La valeur 10 est stockée dans une autre adresse mémoire (car la valeur 2 sert toujours pour `a`) et associée à `b`.

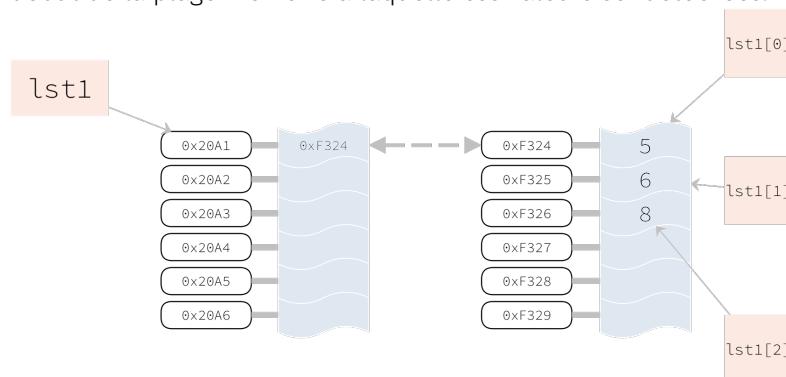


4.2. Variables de type mutable

Copier une liste (mauvaise méthode)

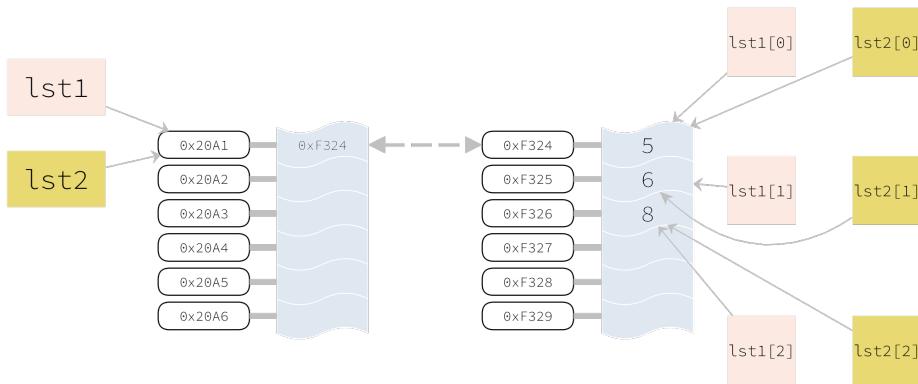
```
lst1 = [5, 6, 8]
```

Les éléments 5, 6 et 8 sont stockés en mémoire et `lst1` contient l'adresse du début de la plage mémoire à laquelle ces valeurs sont stockées.



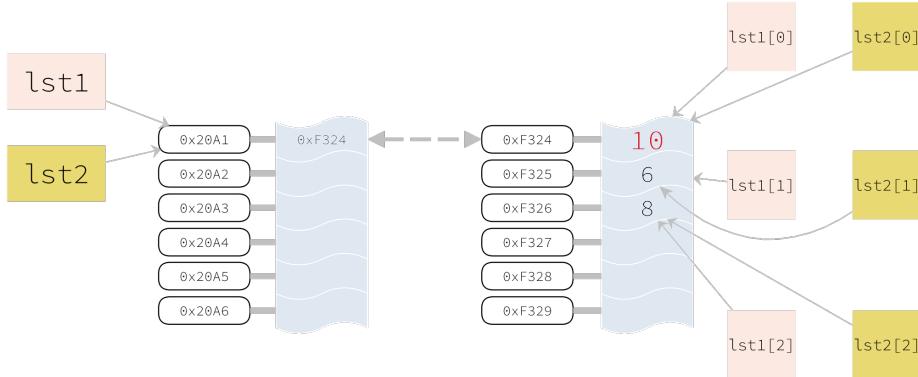
```
lst2 = lst1
```

La variable `lst2` est associée à la même valeur que `lst1`.



`lst1[0] = 10`

La valeur de `lst1[0]` est changée, elle l'est donc aussi pour `lst2`.

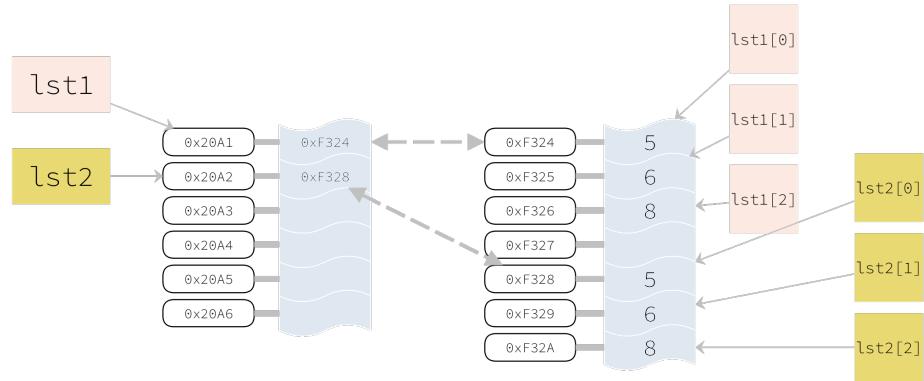


Voilà donc pourquoi lorsqu'on écrit `lst2 = lst1`, tout changement dans `lst1` se reflète aussi dans `lst2` et vice-versa.

Copier une liste (bonne méthode)

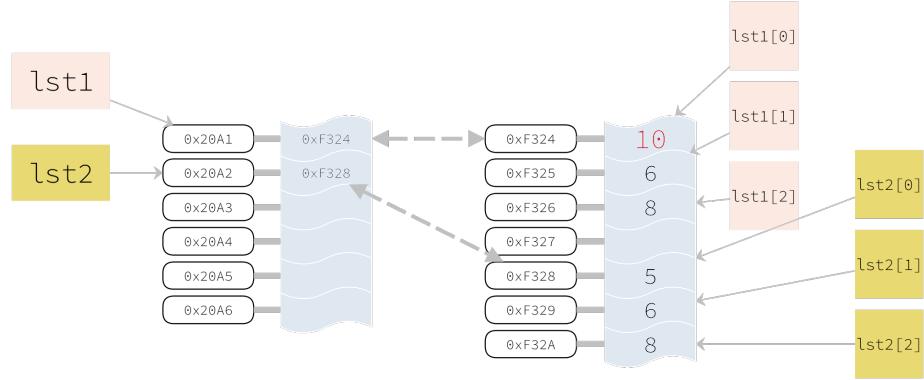
```
lst2 = lst1[:]
```

Les éléments de `lst1` sont recopiés dans une autre zone mémoire, et `lst2` « pointe » sur l'adresse du début de cette zone.



`lst1[0] = 10`

Le changement n'affecte pas `lst2`.



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`

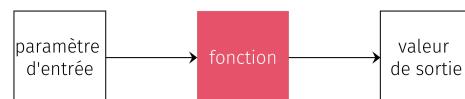
- `len`

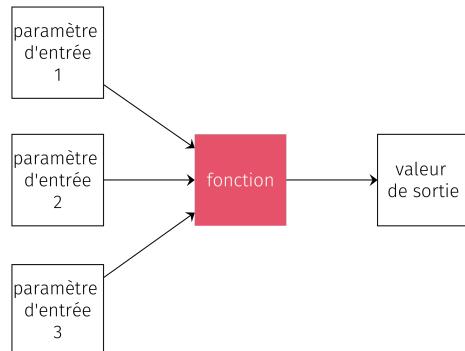
- prend en entrée une liste;
- renvoie le nombre d'éléments de cette liste.

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

1.2. De multiples formes

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



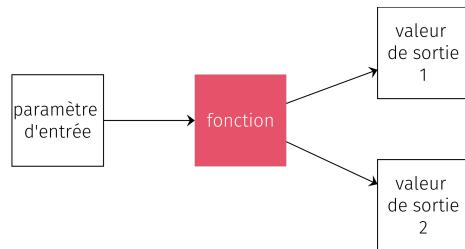


Certaines fonctions sont comme à gauche.

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

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.

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

type du paramètre formel

3.2. Variables locales

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

3.3. valeur de sortie

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

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

type de la valeur renvoyée

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

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

```
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

```
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

```
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.

Chapitre 11

Écriture en compréhension

1. Écritures simples

Jusqu'à présent, pour construire des listes on a souvent :

- créé une liste `lst` vide;
- construit une boucle `for` ou `while`;
- peuplé la liste avec `lst.append`.

Exemple

```
lst = []
for i in range(10):
    lst.append(i*i)
print(lst)
```

Ce programme affiche [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

C'est la liste des carrés des 10 premiers entiers naturels.

En mathématiques, l'ensemble des carrés des 10 premiers entiers naturels se note

$$\{i^2 \mid i \in \mathbb{N}, i < 10\}$$

C'est une écriture en *compréhension*.

On peut faire la même chose en PYTHON :

```
lst = [i*i for i in range(10)]
```

Évidemment, c'est plus rapide que la méthode précédente... Et on peut faire bien plus! On peut utiliser une liste pour en construire une autre, par exemple en ajoutant 1 à chacun des éléments :

Python

```
>>> lst1 = [2, -1, 3, 4, 7]
>>> lst2 = [x + 1 for x in lst1]
>>> lst2
[3, 0, 4, 5, 8]
```

Dans le même esprit, on peut construire une liste dont les éléments sont ceux de la première, mais avec une conversion de type :

Python

```
>>> lst1 = ['2', '0', '13']
>>> lst2 = [int(x) for x in lst1]
>>> lst2
[2, 0, 13]
```

Ou encore fabriquer la liste des initiales à partir d'une liste de prénoms :

Python

```
>>> lst1 = ['Fred', 'Titouan', 'Tinaïg']
>>> lst2 = [prenom[0] for prenom in lst1]
>>> lst2
['F', 'T', 'T']
```

2. Écritures avec conditions

Il est possible d'utiliser `if` en compréhension : mettons dans `lst2` le double de chaque élément de `lst1` supérieur à 10 (dans l'ordre de parcours).

Python

```
>>> lst1 = [8, 0, 11, 10, 3, 15]
>>> lst2 = [2 * x for x in lst1 if x > 10]
>>> lst2
[22, 30]
```

Il est possible d'utiliser `if ... else ...` en compréhension, mais à ce moment là il faut écrire les conditions au début : créons une nouvelle liste en remplaçant tous les nombres négatifs de `lst1` par zéro.

Python

```
>>> lst1 = [8, -10, 11, -4, -3, 15]
>>> lst2 = [(x if x > 0 else 0) for x in lst1]
# les parenthèses sont facultatives
>>> lst2
[8, 0, 11, 0, 0, 15]
```

Créons une liste contenant les indices des éléments de `lst1` qui sont strictement positifs.

Python

```
>>> lst1 = [8, -10, 11, -4, -3, 15]
>>> lst2 = [i for i in range(len(lst1)) if lst1[i] > 0]
>>> lst2
[0, 2, 5]
```

3. Les écritures en compréhension imbriquées

0	0	0	0
0	0	0	0
0	0	0	0

Si on veut représenter ce «tableau de nombres» par une liste, on peut écrire cet liste de 3 lignes comportant chacune 4 éléments :

```
lst = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]].
```

Cependant il est plus pratique d'écrire

```
lst=[[0 for j in range(4)] for i in range(3)]
```

4. Pour conclure

On peut combiner toutes les techniques que nous venons de voir. Par exemple on peut créer une liste de listes de listes avec des conditions, *et cætera*. La seule limite, c'est l'imagination et la capacité à écrire en PYTHON !

Chapitre 12

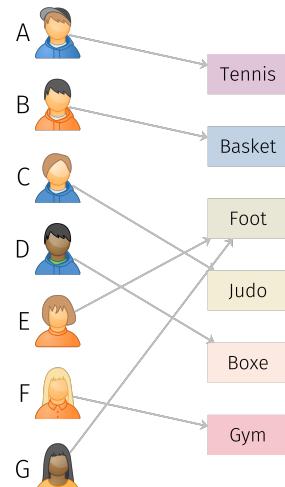
Dictionnaires

1. Un nouveau type

On demande à des jeunes quel est leur sport préféré, les résultats sont présentés sur la figure ci-contre.

Un sport peut être cité par *plusieurs* jeunes, en revanche chaque jeune ne peut citer qu'*un seul* sport.

On pourrait utiliser une ou plusieurs listes pour représenter ces données mais il y a mieux : le *dictionnaire*. La variable `sport` est de type `dict` :



Python

```
sport = {'A': 'Tennis', 'B': 'Basket',  
        'C': 'Judo', 'D': 'Boxe',  
        'E': 'Foot', 'F': 'Gym',  
        'G': 'Foot'}
```

Définition : dictionnaire

Un dictionnaire est un ensemble d'*éléments*.
les éléments sont des couples de la forme *clé* : *valeur*.

La syntaxe est :

```
variable = { cle1 : valeur1, cle2 : valeur2, ...}
```

Les valeurs peuvent être de n'importe quel type. Les clés peuvent être

- des `bool`, des `int`, des `float`;
- des `str`...
- mais pas des `list`!

On peut tout de même utiliser des tuples en guise de clés : les tuples ressemblent aux `list` mais sont *non mutables*.

`a = (1, 2, 3)` est un exemple de `tuple`.

2. Opérations sur les dictionnaires

2.1. Accéder à une valeur par sa clé

Pour connaître le sport préféré de 'A', c'est simple :

Python

```
>>> sport['A']
'Tennis'
```

2.2. Créer de nouveaux couples clé : valeur

Contrairement aux listes, il n'y a pas de méthode `append`.

Pour intégrer l'information « le sport préféré de H est le Rugby » on écrira simplement :

Python

```
>>> sport['H']='Rugby'
```

2.3. Créer un dictionnaire vide et le peupler

On peut partir d'un dictionnaire vide et remplir ses valeurs au fur et à mesure :

Python

```
>>> d = dict()
>>> d['bonjour'] = 'hello'
>>> d['crayon'] = 'pencil'
>>> d['se prélasser'] = 'to bask'
```

2.4. Supprimer un élément du dictionnaire

`del d['crayon']` supprime l'élément 'crayon': 'pencil'.

fusionner 2 dictionnaires

```
>>> d1 = {"anglais": "bread",
          "français": "pain",
          "slovaque": "chlieb"}
>>> d2 = {"allemand": "brot", "italien": "pane"}
>>> d1.update(d2) # fusionne d2 dans d1
>>> d1
{"anglais": "bread", "français": "pain", "slovaque": "chlieb",
→ "allemand": "brot", "italien": "pane"}
```

2.5. Parcourir l'ensemble des clés d'un dictionnaire

Python

```
for cle in d1.keys():
    print(cle)
```

Ce script affiche

anglais
français
slovaque
allemand
italien

2.6. Parcourir l'ensemble des valeurs d'un dictionnaire

Python

```
for valeur in d1.values():
    print(valeur)
```

Ce script affiche

```
bread
pain
chlieb
brot
pane
```

2.7. Précisions

`d1.keys()` et `d1.values()` ressemblent à des listes mais n'en sont pas!¹

Pour avoir par exemple la liste des clés de `d1` on écrira :

```
list(d1.keys())
```

ou bien en *compréhension* (ce qui revient au même mais peut s'avérer utile)
`[k for k in d1.keys()]`

2.8. Erreurs de clé

```
print(d1['suédois'])
```

Ce script produit une erreur :

```
KeyError : 'suédois'
```

Exemple : Utilisation d'un dictionnaire

On veut créer un tableau de 10×10 cases avec la valeur 0 dedans.

On peut bien sûr créer cela avec une liste de listes (en compréhension) mais on peut également utiliser un dictionnaire :

```
{(x, y) : 0 for x in range(1, 11) for y in range(1, 11)}
```

1. Ce sont des *itérateurs*, structures destinées à être parcourues.

Avantages :

- plus simple à manipuler : on écrit `d[x, y]` au lieu de `d[x][y]` ;
- on n'est pas obligé de faire commencer les indices à zéro.

Inconvénients :

- prend plus de place en mémoire (on s'en fiche un peu);
- plus flexible entraîne plus de possibilité d'erreurs!

3. Utilisation des dictionnaires

Typiquement, pour stocker des données structurées :

```
reseau = {'nom'      : 'local',
          'ip'       : '192.168.1.0',
          'masque'   : '255.255.255.0',
          'passerelle': '192.168.1.254'}
```

On utilise fréquemment des listes de dictionnaires, ou bien des dictionnaires de listes.

Chapitre 13

Complexité

1. Des exemples

Voici trois exemples de problèmes qu'on peut vouloir résoudre

1.1. Test de primalité

Étant donné un entier n plus grand que 1, cet entier est-il premier¹ ou non ?

Par exemple, 127 est-il premier ? Pour y répondre, on va diviser 127 par 2, par 3, par 4, et cætera, et regarder si une de ces divisions « tombe juste » ou non. Si c'est le cas, 127 n'est pas premier. On n'a pas besoin de pousser les divisions jusqu'à 126, il suffit simplement d'aller jusqu'à $\lfloor \sqrt{127} \rfloor$, c'est-à-dire onze². Quand on le fait, on se rend compte que 127 est premier.

1.2. Recherche de la présence d'un élément dans une liste

Étant donnée une liste d'entiers `lst` de longueur n et un entier `val`, cet entier appartient-il ou non à `lst` ?

Par exemple, avec une liste `lst` valant [4, 2, 7, 8, 9] et une valeur `val` de 5, il faut parcourir *intégralement* `lst` pour constater que `val` n'y figure pas.

1.3. Table de multiplication

Étant donné un entier $n \in \mathbf{N}^*$, on veut afficher tous les nombres de la forme $i \times j$, avec i et j entiers compris entre 1 et n .

Par exemple pour n valant 4, j'obtiens la table suivante :

1. est-ce qu'il existe d'autres diviseurs de n que 1 et n ?
2. en effet, si n admet un diviseur, alors on peut écrire $n = pq$ avec p et q deux entiers et il y en a obligatoirement un des deux qui est plus petit ou égal à \sqrt{n} , donc à $\lfloor \sqrt{n} \rfloor$ puisqu'il est entier.

1	2	3	4
2	4	6	8
3	6	9	12
4	8	12	16

Pour chacune des situations précédentes, n est appelé *taille du problème*. Plus n augmente, plus le nombre d'opérations (au sens large : calculs, tests, accès aux éléments d'une liste, *et cætera*) augmente.

À quelle vitesse ce nombre d'opérations augmente-t-il ?

S'il y a plusieurs algorithmes pour résoudre un même problème, y en a-t-il un plus efficace que les autres, c'est-à-dire dont le nombre d'opérations augmente moins vite lorsque n augmente ?

2. Complexité temporelle

Évaluer la complexité temporelle d'un algorithme, c'est estimer le nombre d'opérations *significatives* qui entrent en jeu lors de la résolution par cet algorithme d'un problème de taille n .

Méthode : évaluation de la complexité temporelle

Pour évaluer l'efficacité d'un algorithme en terme de nombre d'opérations

- d'abord on décide ce qu'est une opération significative. On appelle ceci une OPEL;
- seules les OPEL sont considérées comme coûteuses en temps et sont comptabilisées;
- les autres opérations sont *négligées*;
- on cherche à estimer le nombre d'OPEL nécessaires à la résolution par l'algorithme d'un problème de taille n .

On peut « imaginer » une fonction c_M (au sens mathématique du terme) qui serait définie pour toute taille n du problème et donnerait le nombre *moyen* d'OPEL nécessaires pour résoudre un problème de taille n .

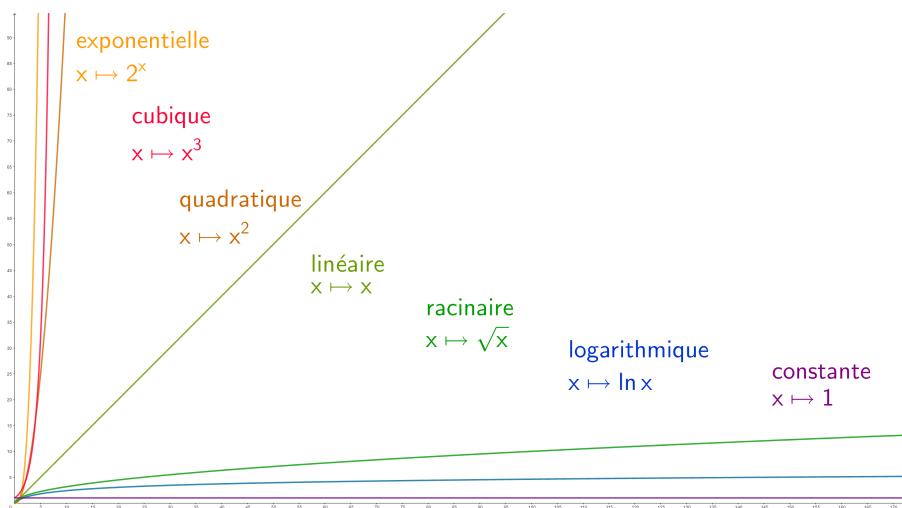
Cette *complexité moyenne* est très rarement calculable car les calculs trop compliqués.

Plus simple mais tout aussi utile que c_M , est la complexité *dans le pire des cas*.

Définition : complexité dans le pire des cas

Pour un problème de taille n , on note $c(n)$ le nombre maximal d'OPÉL pour résoudre un problème de cette taille.

En pratique, on ne calcule pas *exactement* $c(n)$. On se contente d'indiquer à quelle vitesse $c(n)$ augmente lorsque n augmente, en se servant de *fonctions de référence* telles que celles représentées ci-dessous.



Il est très utile de connaître la complexité d'un algorithme car cela nous permet d'estimer le temps de résolution d'un problème quand on connaît l'ordre de grandeur de n comme le montre le tableau ci-dessous.

	5	10	20	50	250	1 000	10 000	1 000 000
constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns
logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns
racinaire	22 ns	32 ns	45 ns	71 ns	158 ns	316 ns	1 µs	10 µs
linéaire	50 ns	100 ns	200 ns	500 ns	2,5 µs	10 µs	100 µs	10 ms
quadratique	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2,8 h
c cubique	1,25 µs	10 µs	80 µs	1,25 ms	156 ms	10 s	2,7 h	316 ans
exponentielle	320 ns	10 µs	10 ms	130 jours	10^{59} ans

3. Application à nos exemples

3.1. Test de primalité

Python

```
from math import sqrt
def est_premier(n: int) -> bool:
    # on parcourt les entiers de 2 à racine(n)
    for i in range(2, int(sqrt(n)) + 1):
        # si une division tombe juste
        if n % i == 0:
            # alors n n'est pas premier
            return False
    # si aucune ne tombe juste il est premier
    return True
```

On convient qu'une OPEL est une division. Pour n fixé, on effectue au pire des cas les divisions par 2, 3, ..., $\lfloor \sqrt{n} \rfloor$, il y en a *grossost modo* \sqrt{n} : notre algorithme est de complexité racinaire³.

3.2. Recherche de la présence d'un élément dans une liste

Python

```
def present(lst: list, val: int) -> bool:
    # on parcourt la liste par ses éléments
    for x in lst:
        # si on trouve val
        if x == val:
            # on renvoie True
            return True
    # si on ne l'a pas trouvé, on renvoie False
    return False
```

Ici une OPEL sera un accès à un élément de la liste, c'est à dire le fait d'attribuer les valeurs à la variable x . Pour une liste de taille n fixé, dans le pire des cas val n'appartient pas à lst et on effectue n OPEL : notre algorithme est de complexité linéaire.

³. en réalité c'est sans doute plus que cela car plus n est grand, plus les nombres qui entrent en jeu dans les divisions sont grands et plus les divisions prennent de temps.

3.3. Table de multiplication

Python

```
def affiche(n: int) -> None:
    for i in range(1, n + 1):
        for j in range(1, n + 1):
            print(i * j)
```

Ici une OPEL est une multiplication. Pour n fixé, dans tous les cas on effectue deux boucles imbriquées, donc $n \times n$ sont effectuées et notre algorithme est de complexité quadratique.

4. Exercices

Exercice 52

Une OPEL est une addition, déterminer la complexité de la fonction suivante :

```
def somme(n: int) -> int:
    s = 0
    for i in range(1, n + 1):
        s += i
    return s
```

Exercice 53 : nombre de bits d'un entier

La fonction suivante

- prend en entrée un n non nul;
- renvoie le nombre de bits nécessaires à l'écriture de n en base 2.

```
def nb_bits(n: int) -> int:
    b = 0
    while n > 0:
        b += 1
        n = n // 2
    return b
```

Une OPEL est une division par 2. En testant la fonction `nb_bits` sur des entiers de plus en plus grands, conjecturer la complexité de cette fonction.

Exercice 54* : addition de matrices

On considère des *matrices*, qui sont représentées par des listes de listes d'`int` (n lignes et n colonnes). Par exemple les matrices

$$M_1 = \begin{pmatrix} 1 & 2 & -1 \\ 0 & -1 & 2 \\ -1 & 1 & 0 \end{pmatrix} \quad \text{et} \quad M_2 = \begin{pmatrix} -3 & 1 & 0 \\ 1 & 0 & -2 \\ 1 & -1 & 1 \end{pmatrix}$$

sont deux matrices carrées d'ordre 3 et sont représentées en PYTHON par
`m1 = [[1, 2, -1], [0, -1, 2], [-1, 1, 0]]` et
`m2 = [[-3, 1, 0], [1, 0, -2], [1, -1, 1]]`

Il est possible d'ajouter 2 matrices de même taille en procédant « case par case » :

$$M_1 + M_2 = \begin{pmatrix} -2 & 3 & -1 \\ 1 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

En convenant qu'une OPEL est une addition de deux `int`, sans écrire l'algorithme, donner la complexité de l'algorithme d'addition de deux matrices.

Exercice 55* : multiplication de matrices

Il est possible de multiplier 2 matrices de même taille suivant l'algorithme suivant :

```
def produit(m1: list, m2: list) -> list:
    # n est la taille des matrices
    n = len(m1)
    # on crée une matrice remplie de zéros
    p = [[0 for j in range(n)] for i in range(n)]
    # pour chaque ligne
    for i in range(n):
        # pour chaque colonne
        for j in range(n):
            # on ajoute tous ces nombres
            for k in range(n):
                p[i][j] += m1[i][k] * m2[k][j]
    # puis on renvoie la matrice produit
    return p
```

Déterminer la complexité de cet algorithme.

Chapitre 14

Recherche dichotomique

«Plus petit ou plus grand?»

1. Présentation de l'algorithme

Lorsqu'on cherche si un élément appartient ou non à une liste, il suffit de la parcourir en comparant chacun de ses éléments à celui que l'on cherche. Cette démarche peut être améliorée si la liste possède des propriétés particulières, notamment si c'est une liste d'entiers triée.

On veut écrire une fonction `recherche_dichotomique` qui :

En entrée prend

- une liste `liste_triee` de n entiers triée dans l'ordre croissant;
- un entier `val`.

Renvoie

- l'indice de `val` dans `liste_triee` si `val` appartient à `liste_triee`;
- -1 si `val` n'appartient pas à `liste_triee`.

Exemple

- `recherche_dichotomique([11, 20, 32, 33, 54], 32)` renvoie 2 car 32 est l'élément d'indice 2 de la liste.
- `recherche_dichotomique([20, 32, 33, 54], 40)` renvoie -1 car 40 ne figure pas dans la liste.

Méthode

On compare `val` avec l'élément m qui se situe «à peu près au milieu de `liste_triee`».

- si `val` est égal à m c'est gagné, on renvoie l'indice de m dans

- liste_triee;
- sinon si $val > m$ on recommence avec la liste des éléments situés après m .
- sinon c'est que $val < m$ et on recommence avec la liste des éléments situés avant m .

On itère ce procédé tant que la liste des valeurs à examiner n'est pas vide. Si on arrive à une liste vide c'est que val , n'est pas dans liste_triee.

Voici l'algorithme traduit en PYTHON :

Python

```

01  def recherche_dichotomique(liste_triee, val):
02      # début de la plage de valeurs à regarder
03      gauche = 0
04      # fin de la plage
05      droite = len(liste_triee) - 1
06      # tant que la plage est non vide
07      while gauche <= droite:
08          # on prend grossièrement le milieu
09          milieu = (gauche + droite) // 2
10          # si on trouve val au milieu c'est gagné
11          if liste_triee[milieu] == val:
12              return milieu
13          # si on a dépassé val
14          elif liste_triee[milieu] > val:
15              # alors on regarde avant
16              droite = milieu - 1
17              # sinon on regarde après
18          else:
19              gauche = milieu + 1
20          # si on est sorti de la boucle
21          # c'est qu'on n'a pas trouvé val
22      return -1

```

2. Comprendre l'algorithme

Commençons par le faire «tourner à la main» sur un exemple, avec liste_triee valant [1, 3, 4, 8, 9, 13, 20, 21].

On cherche la valeur 4.

Pour chaque itération on a noté dans un tableau les valeurs des variables (celles de gauche et droite avant qu'elles ne soient modifiées) et si la fonction renvoie quelque chose ou non.

n° d'itér	gauche	droite	milieu	liste_triee[milieu]	valeur renvoyée
1	0	7	3	8	NON
2	0	2	1	3	NON
3	2	2	2	4	OUI : 2

Ainsi la fonction a renvoyé 2, indice de la valeur 4 dans la liste, au bout de 3 itérations.

On cherche la valeur 15 :

n°itér	gauche	droite	milieu	liste_triee[milieu]	return?
1	0	7	3	8	NON
2	4	7	5	13	NON
3	6	7	2	20	NON
-	7	6	-	-	OUI : -1

La dernière ligne du tableau signifie qu'au bout de la 3^e itération, les conditions de boucles ne sont plus vérifiées (car gauche > droite) et que -1 est renvoyé.

Exercice 56

On cherche la valeur 6 dans la liste précédente.

Complète le tableau (des lignes resteront peut-être vides).

n°itér	gauche	droite	milieu	liste_triee[milieu]	return?

On cherche la valeur 21, complète le tableau (des lignes resteront peut-être vides).

n°itér	gauche	droite	milieu	liste_triee[milieu]	return?

3. Analyse de l'algorithme

Quatre questions se posent :

1. Pourquoi, lorsque la fonction renvoie un entier positif, est-ce bien la position de `val` dans `liste_triee`? C'est un problème de *correction*.
2. Quand la fonction renvoie -1, est-ce que cela veut bien dire que `val` n'est pas dans `liste_triee`? C'est un problème de *complétude*.
3. Pourquoi la boucle *tant que* s'arrête-t-elle toujours? On dit que c'est un problème de *terminaison*.
4. Enfin, pourquoi cette fonction est-elle plus rapide qu'un parcours des éléments un par un? C'est un problème de *complexité*.

4. Correction de l'algorithme

Quand la fonction renvoie un entier positif, c'est à la ligne 12, ce qui signifie qu'on a effectivement trouvé `val` dans `liste_triee`, à la position renvoyée.

5. Complétude de l'algorithme

Pour prouver que cette fonction est complète, on doit utiliser un *invariant de boucle*.

Définition

Un *invariant de boucle* est une propriété \mathcal{P} dépendant éventuellement des variables du programme.

- \mathcal{P} doit être vraie avant l'entrée dans la boucle;
- \mathcal{P} doit rester vraie à chaque itération de boucle;
- à la fin de la boucle, \mathcal{P} doit nous permettre de conclure que la fonction « fait bien ce qu'elle doit faire ».

Dans notre cas voici l'invariant de boucle :

\mathcal{P} : « si `val` est dans `liste_triee` son indice est entre `gauche` et `droite` »

- avant l'entrée dans la boucle `while`, on a `gauche == 0` et `droite == len(liste_triee) - 1` donc \mathcal{P} est trivialement vérifiée;

- dans la boucle, si `liste_triee[milieu] == val` alors on renvoie `val` et la fonction s'arrête et donne bien le résultat attendu;
- sinon si `liste_triee[milieu] > val` alors puisque la liste est triée, la position de `val` ne peut être qu'entre gauche et `milieu-1`, or droite est actualisée avec cette valeur, et `P` reste vraie;
- de même si `liste_triee[milieu] < val`;
- En sortie de boucle \mathcal{P} est toujours vérifiée et puisque `gauche > droite` cela signifie que `val` n'est pas dans `liste_triee`.

On a donc prouvé la complétude de notre fonction.

6. Terminaison

Pour prouver qu'une boucle *tant que* se termine, *en théorie* on détermine un *variant* de boucle.

Définition

Un variant de boucle est un *entier positif* qui décroît strictement à chaque itération de boucle. On le choisit de sorte à ce que lorsqu'il atteint zéro (ou un, en tout cas une petite valeur) la boucle se termine.

Dans notre cas, le variant de boucle est l'entier `v` défini par `v = droite - gauche` : la condition du `while` est liée à `v` puisque `gauche <= droite` équivaut à `v >= 0`.

Pour montrer que `v` décroît strictement il suffit de montrer que ou bien `gauche` augmente strictement ou bien `droite` décroît strictement.

Or lors d'une itération, `m` est toujours entre `gauche` et `droite` (au sens large) et

- soit on trouve que `liste_triee[m]` vaut `val` et la boucle s'arrête;
- sinon ou bien `gauche` devient `m + 1` donc augmente strictement, ou bien `droite` devient `m - 1` donc décroît strictement.

Ainsi les valeurs de `v` décroissent strictement, donc finissent (si on ne trouve pas `val`) par atteindre zéro et la boucle se termine.

On dit qu'on a prouvé la *terminaison* de la fonction.

7. Complexité

On va ici évaluer le nombre d'étapes nécessaires au déroulement de la fonction.

On va raisonner dans le pire des cas : `val` n'appartient pas à la liste.

À chaque itération de boucle, le nombre de valeurs qui restent à examiner est au moins divisé par 2 et lorsque cette valeur vaut 1, c'est qu'on est à la dernière itération de boucle et on est sûr ou bien de trouver `val` à cet endroit, ou bien on sort de la boucle et on renvoie `-1`.

Ainsi, pour une liste triée de taille n , le nombre d'itérations de la boucle dans le pire des cas, c'est le plus petit entier k tel que 2^k dépasse n .

Pour une liste de longueur 2 on est sûrs d'arriver au résultat en 2 itérations, pour une liste de longueur 4, en 3 itérations et en généralisant, si la liste est de longueur 2^n , en $n + 1$ itérations.

Pour un tableau de longueur 1000, puisque $2^9 < 1000 < 2^{10}$, on est sûr d'arriver au résultat au plus en 10 itérations.

Définition

Soit n un entier naturel non nul, on appelle *logarithme en base 2 de n* l'unique réel x solution de

$$2^x = n$$

Ce nombre x est noté $\log_2(n)$.

Ce que l'on vient de prouver, c'est que pour une liste de taille n , la fonction `recherche_dichotomique` nécessitera au plus $E(\log_2(n)) + 1$ itérations pour déterminer si oui ou non une valeur appartient à cette liste (E représente la fonction *partie entière*).

Propriété

Soit une liste triée de longueur $n \in \mathbf{N}^*$.

Soit p le nombre de bits nécessaires pour écrire n en base 2.

La recherche dichotomique d'une valeur dans la liste nécessite **au plus** p accès à cette liste.

Pour cette raison la complexité de l'algorithme de recherche dichotomique est

dite *logarithmique*. C'est bien mieux que celle de la recherche simple.

Exercice 57 : efficacité de l'algorithme

1. Dans une liste triée de taille 10 000, en combien d'étapes l'algorithme de recherche dichotomique s'arrête-t-il *dans le pire des cas* ?
2. Même question pour une liste de taille 100 000 et pour une liste de taille 1 000 000.

Chapitre 15

Algorithmes gloutons

1. Une manière de procéder...

Définition : algorithme glouton

Un algorithme est dit *glouton* lorsque

- il procède étape par étape, avec une boucle;
- à chaque itération il essaye d'*optimiser* une grandeur (maximiser ou minimiser) en faisant un *choix*;
- les choix faits sont *définitifs* : ils ne sont jamais remis en question lors des itérations suivantes.

Exemple : rendu de monnaie

Lorsqu'on rend la monnaie en euros et qu'on veut rendre le moins de pièces (ou billets) possibles, on

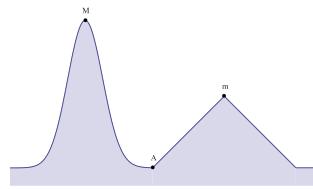
- procède pièce par pièce;
- choisit la pièce dont la valeur est la plus grande possible tout en restant inférieure ou égale au montant qu'il reste à rendre;
- on continue ainsi jusqu'à ce qu'il ne reste plus rien à rendre, sans jamais reprendre une pièce rendue auparavant.

Cette méthode est gloutonne et elle permet toujours de rendre la monnaie avec le moins de pièces possible (en tout cas lorsque le système monétaire est l'euro).

2. Qui n'est pas toujours optimale

Considérons un robot placé en A, qui veut monter le plus haut possible. S'il applique la méthode gloutonne suivante :

- à chaque seconde, tant que possible;
- regarder à droite ou à gauche sur une petite distance;
- aller dans la direction où la pente est la plus forte.



Alors il se retrouvera en m, et pas en M.

À retenir

- un algorithme glouton ne fournit pas toujours une solution optimale;
- pour s'assurer qu'il fournit une démonstration optimale, il faut le *démontrer*.

3. Des exemples optimaux

- le rendu de pièces en euros;
- l'écriture d'un entier naturel en binaire par la méthode des soustractions (qui correspond à un rendu de pièces qui ont des valeurs de 2^n);
- l'algorithme dit «des conférenciers»;

Nous en verrons d'autres en Terminale.

4. Des exemples non optimaux

- le problème du robot exposé précédemment;
- les méthodes gloutonnes pour résoudre le problème du «sac à dos».

Chapitre 16

L'algorithme des k plus proches voisins

Cet algorithme s'appelle *k Nearest Neighbors* en anglais, nous l'appellerons donc *kNN*.

1. Des questions concrètes

1. Une entreprise de vente d'articles en lignes collecte les informations de ses clients. Elle en a accumulé un grand nombre, tels que

- l'âge;
- le sexe;
- l'adresse;
- les revenus moyens mensuels;
- *et cætera*.

En fonction des achats, réguliers ou non, elle a placé ses client.e.s dans des catégories telles que

- client.e fidèle;
- client.e à fidéliser;
- *et cætera*.

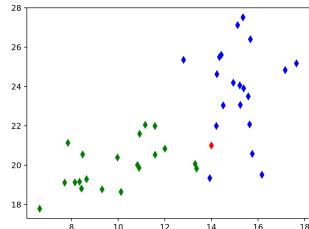
Un nouveau client se présente.

Connaissant son âge, son sexe, son adresse et ses revenus mensuels, dans quelle catégorie l'entreprise va-t-elle le placer ?

2. On a mesuré la largeur et la longueur des pétales et des sépales d'iris de 3 catégories (*iris setosa*, *iris versicolor* et *iris virginica*) On effectue des mesures sur une nouvelle fleur. Dans quelle catégorie va-t-on la placer ?

Pour répondre aux deux questions précédentes on utilise le même algorithme : *kNN*.

2. Un algorithme pour y répondre



On considère deux nuages de points, l'un vert et l'autre bleu. Le point rouge (appelons-le P) doit-il être considéré comme appartenant au nuage vert ou au nuage bleu ?

Pour répondre à cette question, on va

Méthode : kNN

- choisir un entier k impair (3, 5 ou 7 typiquement);
- calculer les distances entre P et tous les autres points;
- sélectionner les k points les plus proches de P;
- regarder leurs couleurs.

Puisque k est impair il n'y aura pas de situation d'*ex æquo* et, suivant la couleur majoritaire des « k plus proches voisins » de P, on pourra choisir celle de P.

C'est cela, l'algorithme *kNN*.

La seule chose qui change selon la situation, c'est la *distance* que l'on utilise. Avec les nuages de points, on utilise la distance euclidienne :

$$d(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Mais on peut utiliser une distance différente suivant la situation.

Par exemple, si on ne manipule non plus des points mais des quadruplets $A = (a_0, a_1, a_2, a_3)$, alors on peut poser

$$d(A, B) = \sqrt{(b_0 - a_0)^2 + (b_1 - a_1)^2 + (b_2 - a_2)^2 + (b_3 - a_3)^2}$$

ou encore

$$d(A, B) = |b_0 - a_0| + |b_1 - a_1| + |b_2 - a_2| + |b_3 - a_3|$$

Et d'ailleurs, si c'est la proximité de la première composante qui importe le plus, on peut définir

$$d(A, B) = 50 \times |b_0 - a_0| + |b_1 - a_1| + |b_2 - a_2| + |b_3 - a_3|$$

Pour plus de renseignements sur ce qu'est une distance, tu peux consulter Wikipédia.

Chapitre 17

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

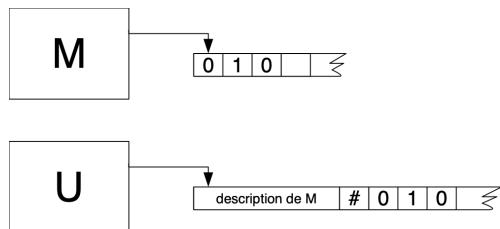
- un ensemble d'états \mathcal{E} dans lesquels la machine M pourra se trouver;

- un ensemble de *symbols* (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 58

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).



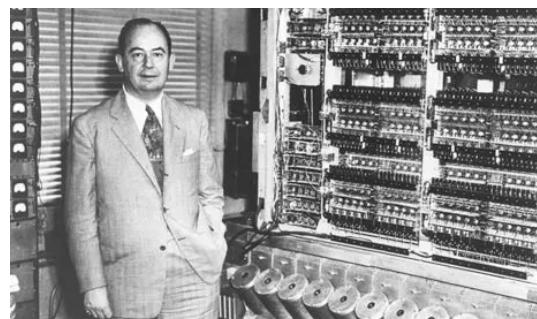
Une machine de Turing universelle.

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 peut 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 59

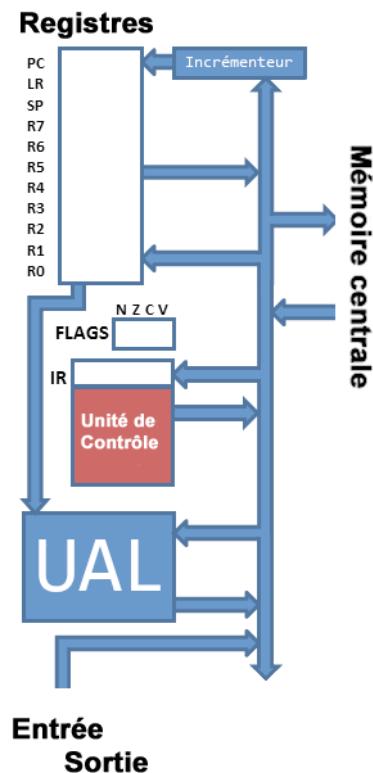
Entre un transistor des années 50 (quelques centimètres) et un transistor actuel, quel est le facteur de réduction de taille ?

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 60

Faire l'activité : Simulateur de CPU.

Chapitre 18

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
- \bar{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.

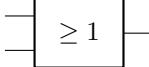
Parmi les notations que l'on rencontre pour noter « x et y » il y a

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

L'opérateur « ou »

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

x	y	x ou 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

– $x \text{ OR } y$

– $x \vee y$

– $x \mid\mid y$

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 $\text{not}(A \text{ and } B)$ et $(\text{not } A) \text{ or } (\text{not } B)$ 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 61

Montrer que $\text{not}(A \text{ or } B)$ et $(\text{not } A) \text{ and } (\text{not } B)$ sont équivalentes.

Exercice 62

On peut définir le « ou exclusif » noté xor comme ceci : $A \text{ xor } B$ 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 $A \text{ xor } B$ équivaut à $(A \text{ or } B) \text{ and } (\text{not}(A \text{ and } B))$.
3. Montrer que $A \text{ xor } B$ équivaut à $(A \text{ and } (\text{not } B)) \text{ or } ((\text{not } A) \text{ and } B)$.
4. Représenter $A \text{ xor } B$ avec un circuit logique, en utilisant les symboles de porte logique européens.

Exercice 63

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

$$A \downarrow B = \text{not}(A \text{ or } B)$$

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 $A \downarrow A = \text{not } A$.
3. En utilisant les exemples de la page précédente, en déduire que

$$(A \downarrow B) \downarrow (A \downarrow B) = A \text{ or } B$$

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 :

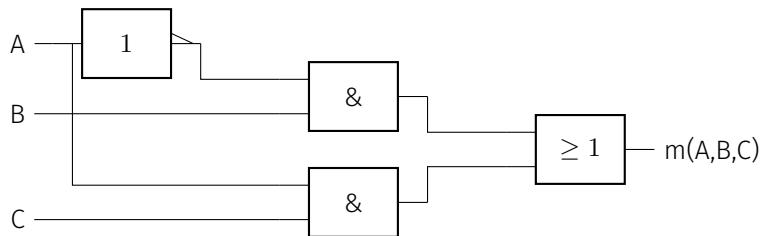
A	B	C	m(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

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

$$m(A,B,C) = (A \text{ et } C) \text{ ou } ((\text{non } A) \text{ et } B).$$

Le circuit logique modélisant m est le suivant :



Exercice 64

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 65

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	00	[NULL]	32	20	[SPACE]	64	40	@	96	60	'
1	01	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	02	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	03	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	04	[EOT - END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	05	[ENQ/URGENT]	37	25	%	69	45	E	101	65	e
6	06	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	07	[BELL]	39	27	,	71	47	G	103	67	g
8	08	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	09	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	0A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	0B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	0C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	0D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	0E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	0F	[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	[ENO 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 $(a \text{ XOR } b) \text{ XOR } b$.
- b. Bob connaît la chaîne de caractères utilisée par Alice pour chiffrer le message. Quelle opération doit-il réaliser pour déchiffrer son message ?

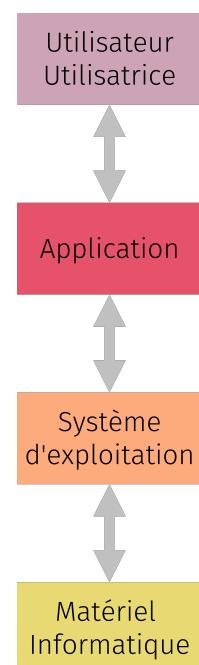
Chapitre 19

Systèmes d'exploitation

1. Qu'est-ce qu'un système d'exploitation ?

Un système d'exploitation (*Operating System* en anglais, abrégé OS) est un ensemble de programmes. Il a plusieurs fonctions :

- c'est un intermédiaire entre l'utilisateur et les applications qu'il utilise (qui sont aussi des programmes) et le matériel;
- c'est lui qui partage les *ressources* entre les différents programmes en train d'être exécutés voire entre les différents utilisateurs;
- c'est lui qui protège la machine des applications, et les applications les unes des autres;
- c'est lui qui gère l'adaptation entre l'*interface* (graphique par exemple) et le matériel.



Les ressources, ce sont entre autres :

- le matériel qui compose l'ordinateur proprement dit (CPU, mémoire, carte graphique...) et les *périphériques* (imprimante, clé usb, clavier);
- les fichiers;
- les éventuelles connexions à des réseaux.



Il existe de multiples OS.

- pour PC : WINDOWS et beaucoup de distributions de LINUX;
- pour APPLE : MACOS;
- pour iPhone et iPad : IOS;
- pour beaucoup de smartphones : ANDROID.

Les appareils connectés ainsi que les « box » des fournisseurs d'accès à Internet et les consoles de jeux disposent aussi de leurs OS. Certains OS sont *gratuits*, d'autres *payants*.

2. Le système de gestion des fichiers

Un *fichier* est un ensemble de données numériques réunies sous un même nom et enregistré sur un support de mémoire permanent appelé *mémoire de masse* (ce peut être un disque dur, un DVD-rom, une clé USB, une carte SD). Un *système de gestion de fichiers* est une façon de stocker les fichiers sur la mémoire de masse. La plupart des systèmes d'exploitation stockent les fichiers dans des *répertoires* organisés de manière hiérarchique selon une *arborescence* (voir plus bas).

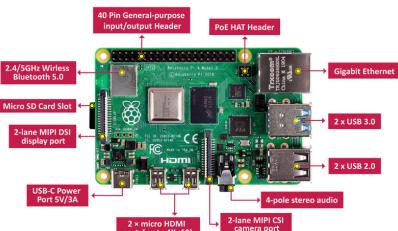
3. L'exemple de Linux et du shell

En 1970 naît UNIX, l'un des premiers système d'exploitation multi-utilisateurs.

Cependant ce système d'exploitation n'est pas *libre*, c'est à dire que son code source n'est pas à la disposition du public.

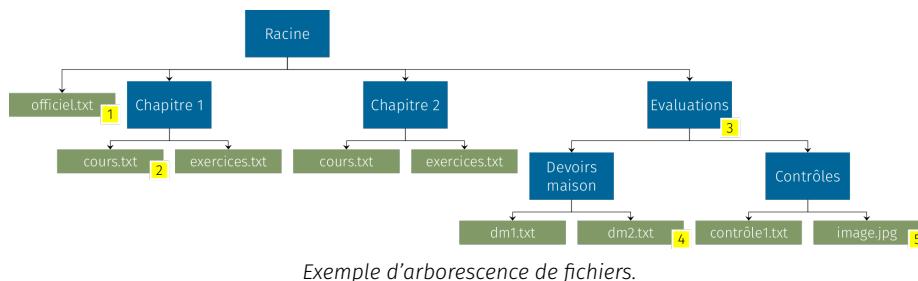
En 1991, un étudiant finlandais du nom de Linus Torvalds entreprend de créer un clone d'UNIX en le ré-écrivant totalement.

Ce nouveau système d'exploitation libre s'appelle LINUX et est désormais largement utilisé.



les Raspberry Pi tournent sous Linux

C'est la plupart du temps une version (appelée *distribution*) de LINUX que l'on installe sur les ordinateurs Raspberry Pi.



Dans un système de fichier tel que celui de LINUX, la racine se note / et les séparateurs (équivalents des flèches du graphique ci-dessus) aussi.
Il y a deux manières d'accéder aux fichiers.

Chemin absolu

On part de la racine et on écrit le chemin pour arriver à l'objet désiré.
Ainsi le chemin absolu de l'objet 1 est /officiel.txt et celui de l'objet 5 est /Evaluations/Contrôles/image.jpg

Exercice 66

Donner les chemins absolus des autres objets.

Chemin relatif

On part d'un emplacement précis (à l'intérieur d'un répertoire) et l'on indique le chemin pour aller à l'objet voulu. S'il faut remonter d'un cran dans l'arborescence on utilise la notation .. comme ceci :

- si l'on est dans le répertoire Evaluations et qu'on veut accéder à dm2.txt alors son chemin relatif est Devoirs maison/dm2.txt;
- si l'on est dans Contrôles et qu'on veut accéder au exercices du chapitre 1, le chemin relatif est ../../Chapitre 1/exercices.txt car il faut d'abord remonter l'arborescence de 2 crans.

Exercice 67

On est dans le répertoire Contrôles, donner les chemins relatifs des objets 1 à 5.

3.1. Le shell

Nous sommes habitués à un environnement graphique pour gérer les copies et déplacements de nos dossiers (et bien d'autre choses comme par exemple sélectionner des fichiers à compresser) mais ce n'est pas la seule manière. Dans tout système d'exploitation muni d'un environnement graphique, on peut trouver un *terminal* (aussi appelé *invite de commande*, ou *shell*) parce que

- c'est avec cette interface textuelle minimalistique que l'on interagissait avec l'OS avant;
- finalement quand on sait (très bien) s'en servir, on peut faire beaucoup plus de choses plus rapidement avec un shell et au clavier qu'avec un clavier, une souris et un environnement graphique.

Exercice 68

Faire l'activité *utiliser le shell*.

Chapitre 20

Unité et diversité des langages

1. Des éléments communs

Un langage de programmation sert à traduire des algorithmes pour les exécuter sur un ordinateur. Il est composé

- d'un alphabet (ensemble de lettres et de symboles);
- d'un vocabulaire (les *mots-clés* du langage);
- d'une grammaire (la *syntaxe* du langage).

Les notions suivantes sont communes à l'immense majorité des langages :

- une *instruction* est un ordre donné;
- une *variable* est un nom qui fait référence à une donnée manipulée par le programme et susceptible de changer au cours de celui-ci;
- une *constante* est un nom qui fait référence à une valeur immuable;
- un *type* qui sert à classifier une variable ou une constante et conditionne les opérations qu'il est possible d'effectuer;
- la *déclaration* consiste à renseigner le traducteur du programme sur la nature des données du programme (type, valeur).
- les *structures de contrôle* telles que le *test* et les *boucles*.
- une *fonction* (ou procédure, ou méthode) sert à isoler un fragment de programme pour pouvoir l'utiliser (éventuellement plusieurs fois) de manière paramétrée.

2. Des différences

2.1. Différences formelles

Tous les langages n'utilisent pas la même syntaxe. En examinant un même algorithme écrit dans plusieurs langages, on constate que

- L'affectation peut être signifiée par = (comme en PYTHON, BASIC, C, FORTRAN...) ou par := (ADA, ALGOL, Go pour partie...). On utilise aussi <- en CAML.
- Les *listes* (ou tableaux) sont très souvent indiquées à partir de zéro... Mais pas toujours (FORTRAN, LUA).
- Très souvent, les structures de contrôles sont délimitées par des accolades (C, JAVA, KOTLIN...) ou par des begin et des end (RUBY, PASCAL...). Parfois on utilise des variantes du end telles que done, END-IF ou NEXT pour signifier une fin de boucle « pour ».

Certains langages ont des syntaxes très similaires : le langage C, JAVA et JAVAS-CRIPT par exemple (notamment le fait qu'un point-virgule termine une ligne). D'autres ont une syntaxe et une mise en forme particulière, éloignée de tous les autres, comme le BASIC ou le COBOL.

2.2. Différences structurelles

Certains langages obligent à déclarer le type des variables lors de leur création. C'est le cas de C ou JAVA. D'autres obligent même à renseigner les variables et leur type avant toute chose, comme ADA, ALGOL, COBOL. Ce n'est pas le cas en PYTHON ou en RUBY.

Une autre différence majeure vient de la manière dont est traité le programme par le traducteur :

- En C ou en C++, le programme est transformé en *langage-machine* par un *compilateur*. L'intérêt est que l'on gagne en rapidité lors de l'exécution du programme.
- En PYTHON le programme est *interprété* lors de son exécution, ligne par ligne.
- Beaucoup de langages utilisent un procédé hybride : le langage utilise une machine virtuelle, et les programmes sont compilés dans le langage de cette machine virtuelle, qui sera ensuite exécuté. La compilation peut être faite avant et le résultat stocké dans un fichier, ou bien être faite à la volée (*Just In Time compilation*).

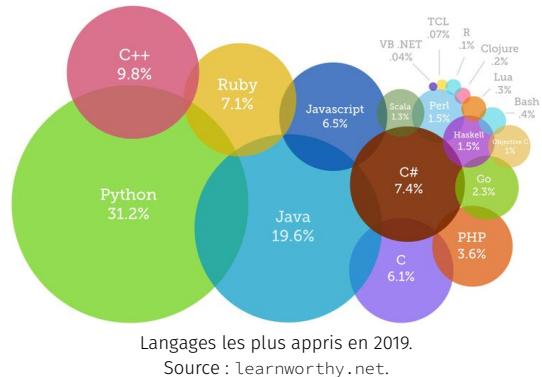
Chaque langage suit un ou plusieurs *paradigmes* de programmation qui change radicalement la manière d'écrire les programmes.

Chaque langage suit un ou plusieurs *paradigmes* de programmation qui change radicalement la manière d'écrire les programmes.

Nous avons vu la programmation *impérative* (séquentielle, linéaire) où le programme effectue une liste d'instructions pas-à-pas. Il existe la programmation *événementielle* lors de laquelle on précise à PROCESSING ce qu'il doit faire quand tel ou tel évènement se produit.

D'autres paradigmes de programmation telle la programmation *orientée objet* ou la programmation *fonctionnelle* sont très utilisés.

Enfin on distinguera des différences dans les contexte d'utilisation de chaque langages : il existe des langages généralistes, des langages qui sont censés être exécutés sur des « serveurs de pages web » tels PHP, d'autres qui (à la base) ont été conçus pour être exécutés au sein même d'un navigateur (JAVASCRIPT).



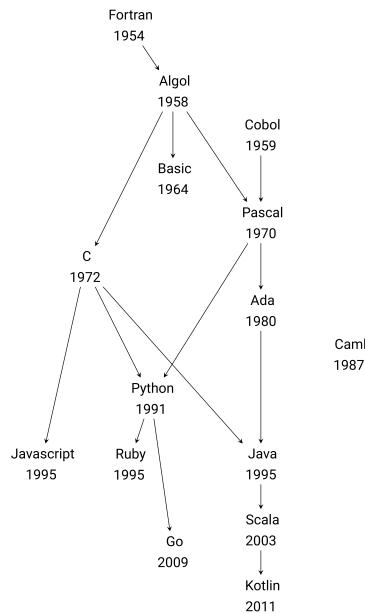
3. Évolution au cours du temps

Les progrès réalisés sur les analyseurs de code permettent de créer des langages avec une syntaxe de plus en plus épurée courte et pour lesquels il n'est pas obligé de déclarer le type de chaque variable.

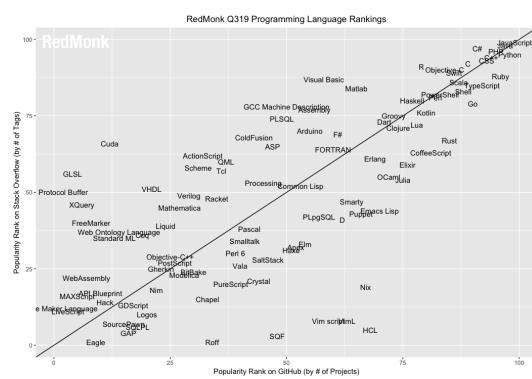
Les nouveaux langages s'inspirent souvent de leurs prédécesseurs.

On continue d'inventer de nouveaux langages : ceux-ci sont créés en fonction des besoins de l'époque.

De nos jours, les langages qui permettent de programmer facilement plusieurs tâches simultanées (comme par exemple récupérer des données sur INTERNET et en même temps traiter ces données) ont le vent en poupe.



Voici un graphe indiquant comment les langages anciens ont influencé les nouveaux.



Stack Overflow est un forum d'entraide à la programmation.

GitHub est un service d'hébergement de projets logiciels.

Le graphique suivant indique la popularité de différents langages sur ces deux sites.

SOURCE : redmonk.com.

Chapitre 21

Réseau - principes

Un *réseau*, c'est un ensemble d'entités qui *communiquent* :

- des fourmis qui envoient des informations par voie *chimique* (les phéromones);
- des individus qui s'envoient des colis postaux ou du courrier;
- des ordinateurs qui s'envoient des données.

Chacun des trois exemples ci-dessus est remarquable par son efficacité. Le premier est sans doute le fruit de l'évolution.

Le deuxième est le résultat de *conventions* et *processus* que les êtres humains ont produits : parmi ces conventions il y a les règles d'écriture d'une adresse postale et aussi le moyen de payer le transport du colis via un ou des timbres postaux et parmi les processus il y a l'acheminement du courrier par voie routière, ferrée, maritime ou aérien-ch-reseaux/imgne.

Quant au troisième, il est également le fruit de conventions et processus très aboutis que nous allons tenter de détailler dans ce chapitre, il fonctionne remarquablement bien et s'appelle Internet. Ce réseau, composé de *milliards* de machines qui communiquent entre elles et qui permet à deux ordinateurs séparés de milliers de kilomètres de s'échanger une quantité considérable d'informations en une fraction de seconde est bel et bien la plus grande structure que l'être humain ait construit à ce jour. Elle mérite bien qu'on s'attarde un peu sur son fonctionnement.

1. L'idée du modèle à couches empilées

La finalité d'un réseau, quel qu'il soit, c'est transporter de la manière la plus fiable possible des informations, lesquelles sont émises par une entité A, à destination d'une entité B, et seront transmises en passant très probablement par des entités intermédiaires.

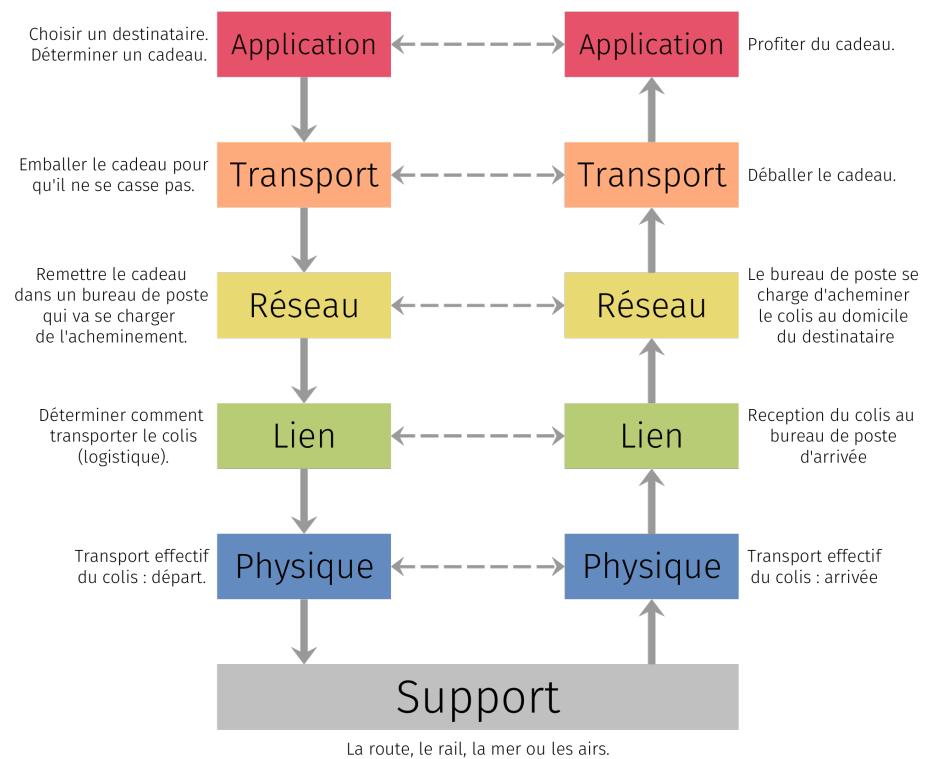
Le processus peut être variable mais il existe des mécanismes communs à la

plupart des moyens de communication : on cherche à décomposer en étapes et les similitudes apparaissent :

- A possède une information « brute »;
- l'information est « conditionnée » pour être envoyée;
- il existe un moyen fiable de savoir comment faire arriver l'information en B;
- en définitive, la transmission s'effectue par un moyen physique;
- du point de vue de B, les choses s'effectuent « à l'envers » : par moyen physique, A a réussi à lui faire parvenir une information « conditionnée » que B traite pour finalement disposer de l'information brute de A.

1.1. L'exemple du réseau postal

Les considérations précédentes sont très abstraites, nous allons donc envisager l'exemple d'une personne A qui souhaite envoyer un cadeau à B. L'information est le cadeau en lui même, le réseau est le réseau postal :



On a décomposé la communication en 5 couches, auxquelles on a donné des noms qui sont déjà liés au vocabulaire des réseaux informatiques. En voici le détail dans l'ordre chronologique :

– Du point de vue de A :

- **Couche 5 - Application :** A veut envoyer un cadeau à B;
- **Couche 4 - Transport :** le cadeau est emballé pour pouvoir être transporté et reçu intact. On y met aussi l'adresse postale de B;
- **Couche 3 - Réseau :** le bureau de poste détermine comment acheminer le colis : il a déjà l'adresse postale de B mais il trouve par lui-même la route qu'il fera suivre au colis;
- **Couche 2 - Lien :** La route est trouvée mais il faut déterminer comment transporter ce colis : si c'est par voie routière on doit savoir quels véhicules capables d'acheminer le cadeau sont disponibles maintenant ou bien quels sont ceux qui le seront très prochainement;
- **Couche 1 - Physique :** c'est le transport effectif du colis.

– Du point de vue de B :

- **Couche 1 - Physique :** c'est également le transport effectif du colis;
- **Couche 2 - Lien :** le colis arrive au bureau postal;
- **Couche 3 - Réseau :** le bureau de poste détermine comment acheminer le colis chez B;
- **Couche 4 - Transport :** le cadeau est arrivé chez B qui le déballe;
- **Couche 5 - Application :** B profite de son cadeau.

L'intérêt de la décomposition en couches est qu'elle sépare un problème complexe en une succession de tâches plus simples. Chaque couche n'interagit directement qu'avec ses couches voisines (ce sont les flèches en trait plein) de manière *relativement simple*. L'interaction de chaque couche chez A avec la couche de même niveau chez B est cependant plus complexe (ce sont les flèches en pointillés).

- **Couche 1 A - Couche 1 B :** c'est le trajet du véhicule qui transporte le cadeau;
- **Couche 2 A - Couche 2 B :** c'est la planification du trajet;
- **Couche 3 A - Couche 3 B :** c'est l'envoi du cadeau au niveau postal;
- **Couche 4 A - Couche 4 B :** c'est la phase emballage/déballage;
- **Couche 5 A - Couche 5 B :** c'est l'envoi du cadeau de A à B.

En général un colis postal est d'un seul tenant mais on peut très bien imaginer que le cadeau que A veut envoyer à B soit très volumineux (par exemple la collection des romans « Les Rougon-Macquart » d'Émile Zola qui comporte 20 livres qui se suivent). Lors de la phase d'emballage, plusieurs « paquets » sont

créés et envoyés séparément. B ne les reçoit pas obligatoirement dans l'ordre d'envoi mais ce n'est pas grave car il finit par avoir tous les paquets et remet les romans dans le bon ordre.

2. Un modèle informatique : TCP/IP

Un ordinateur A veut envoyer des données à un ordinateur B. On supposera que ces données sont *un fichier*. Le processus ressemble à l'exemple précédent. On va présenter le modèle le plus courant dont le nom est TCP/IP et qu'on peut également modéliser avec 5 couches empilées.

On appelle *protocole* tout programme utilisé par une couche.

2.1. Du point de vue de l'émetteur

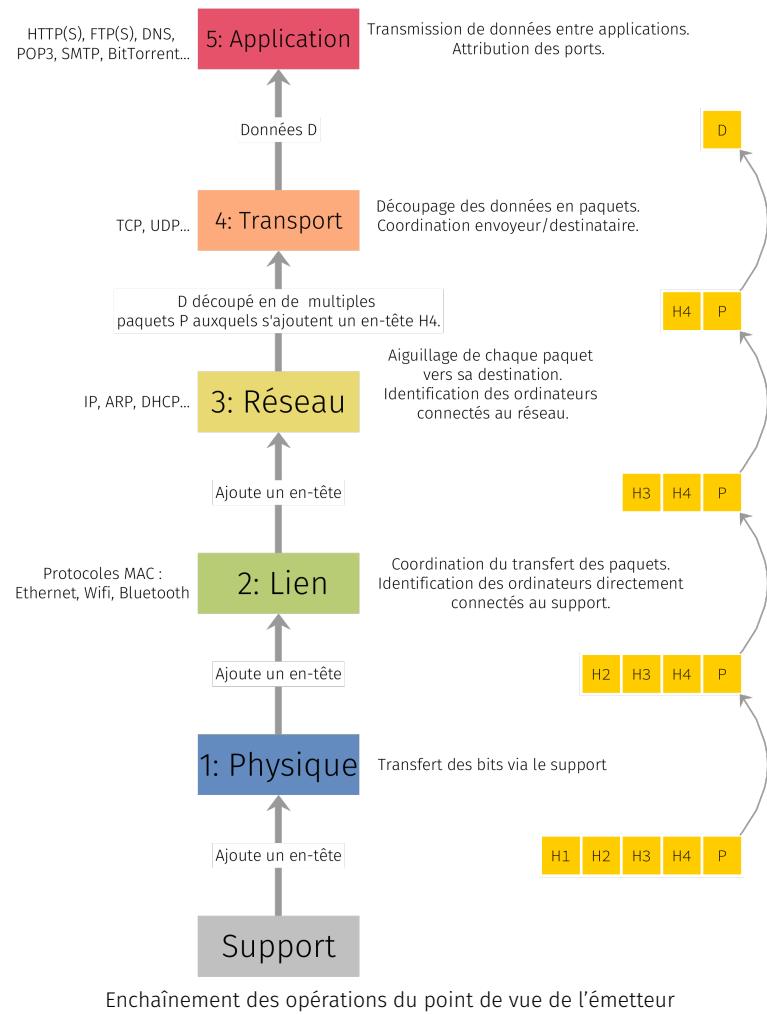
Ce qu'il faut retenir c'est que lors de la transmission

- très tôt dans le processus, le fichier est découpé en paquets;
- chaque couche (hormis la couche application) ajoute son propre en-tête aux données reçues par la couche supérieure et destiné à la même couche pour le destinataire. Ce procédé s'appelle *encapsulation*.

Couche 5 - Application : Cette couche est chargée d'envoyer les données d'un programme sur l'ordinateur A à un programme sur l'ordinateur B. Puisque plusieurs programmes peuvent utiliser la couche transport en même temps, il existe 65536 *ports* et chaque application va utiliser un ou plusieurs ports particuliers. Les numéros de ports sont attribués par convention mais il est possible de les changer.

Les protocoles les plus connus sont

- HTTP (*HyperText Transfer Protocol*), protocole de transfert hypertexte, sur le port 80 ou 443 pour sa version sécurisée HTTPS;
- FTP (*File Transfer Protocol*), protocole de transfert de fichiers, sur les ports 20 et 21, et 990 pour sa version sécurisée FTPS;
- SMTP (*Simple Mail Transfer Protocol*), chargé d'envoyer les emails et POP3 (*Post Office Protocol*) de les récupérer;
- DNS (*Domain Name System*), chargé de traduire un nom de site en adresse IP;
- BitTorrent, chargé de récupérer et diffuser des fichiers.



Couche 4 - Transport : Cette couche va découper les données fournie par la couche 5 en paquets (d'une taille de l'ordre du kilooctet). Elle ajoute également son propre en-tête à chaque paquet.

Les protocoles les plus connus sont

- TCP (*Transfert Control Protocol*), protocole de contrôle des transmissions, qui est fiable, fonctionne sur le principe d'une connexion avec accusé de réception.
- UDP (*User Datagram Protocol*), non fiable en théorie, sans connexion ni accusé de réception, mais plus rapide.

On utilise TCP lorsque les données à recevoir sont sensibles (téléchargement d'un fichier). UDP sera préféré dans le cas du *streaming* : le protocole est plus rapide et permet par exemple de visionner un film avec une bonne qualité, et si des données sont perdues, ce n'est pas grave, on passe aux données suivantes,

cela ne perturbe pas trop le film (on ne va pas, comme ce serait le cas pour TCP, attendre que les données soient arrivées, et faire des pauses).

Couche 3 - Réseau : Cette couche est responsable de l'aiguillage des paquets vers l'adresse de destination. Elle encapsule les paquets de la couche 4 et y ajoute son propre en-tête (en général un en-tête IP). Du point de vue de cette couche, les paquets sont à envoyer à un ordinateur muni d'une adresse IP (nombre attribué par ce protocole).

Pour que les paquets trouvent leur chemin de l'ordinateur A à l'ordinateur B (qui peut être à plusieurs milliers de kilomètres de A), de *protocoles de routage sont utilisés*, les machines dédiées au routage s'appellent des *routeurs*. Un routeur n'utilise que les couches 1,2 et 3 du modèle.

Il faut retenir que chaque paquet suit son propre chemin, indépendamment des autres et qu'il a un « temps de vie » (pour éviter, s'il est perdu, de circuler sur le réseau pendant des années).

Parmi les protocoles utilisés par cette couche on trouve

- IP (*Internet Protocol*), décrit ci-dessus;
- ARP (*Address Resolution Protocol*) qui permet de trouver l'adresse *physique* de l'ordinateur connecté à l'adresse IP. Ce protocole est utilisé lorsque l'information est arrivée à B et « remonte les couches ».

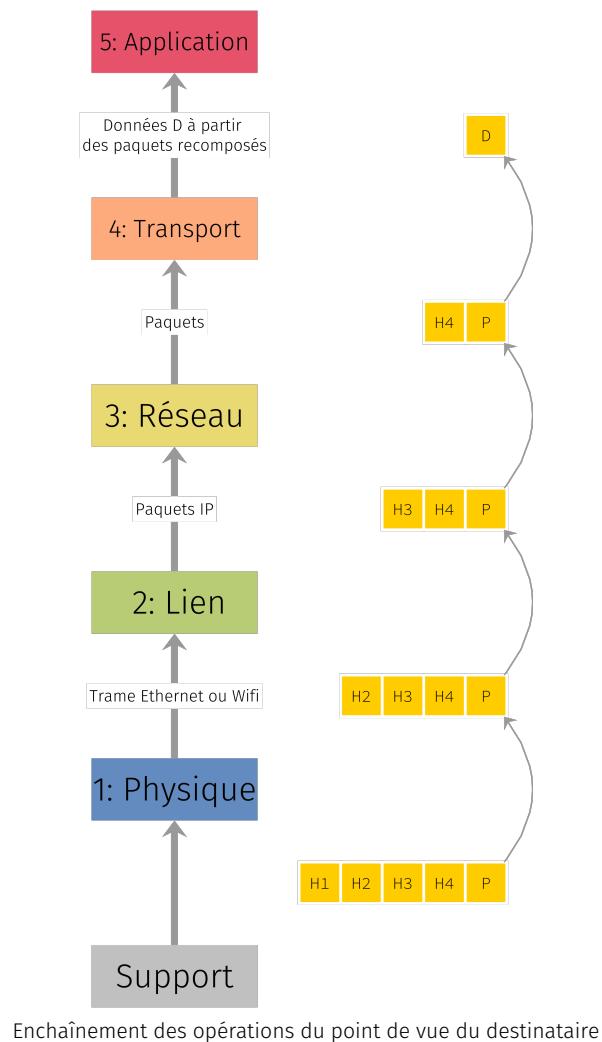
Couche 2 - Lien : Elle est chargée de coordonner le transfert des données ainsi que le « temps de parole » de chaque machine connectée au support physique (si toutes les machines connectées à un même support émettent des données en même temps, il risque d'y avoir des *collisions*). Elle encapsule les paquets de la couche supérieure dans ce qu'on appelle des *trames* (trames Ethernet par exemple).

Du point de vue de la couche lien, la destination d'un paquet est une adresse physique appelée adresse MAC (Media Access Control), déterminée par la carte réseau de la machine destinataire. Les protocoles utilisés par cette couche sont Ethernet (réseau filaire), Wi-Fi (réseau via les ondes).

Couche 1 - Physique : Elle est chargée de la transmission effective des trames (qu'elle encapsule également) d'un bout à l'autre du support physique.

2.2. Du point de vue du récepteur

Il faut retenir que chaque couche dépaquette le paquet (ou trame) qui lui est adressé en enlevant l'en-tête correspondant et passe le relais à la couche du dessus.



3. Le matériel

3.1. Couche physique

Liaison filaire

Les informations peuvent être transmises via des câbles. Le plus utilisé est le câble Ethernet. Son nom est UTP-CAT5 ou UTP-CAT6, la différence étant que le second permet un débit dix fois plus grand que le premier (1 Gbit/s contre 100 Mbit/s).

Parmi les autres types de liaisons filaires, on compte la liaison par câble téléphonique (qui permet par exemple de se connecter à Internet par ADSL) et la liaison par câble optique.



Bluetooth



C'est une technologie utilisant les ondes radios pour permettre la communication entre les équipements électroniques (imprimantes, téléphones, scanners, système audio portatif ou dans un véhicule...) à courte distance. Ses fonctionnalités sont assez limitées en terme de mise en réseau.

Wi-Fi

Cette technologie utilise également les ondes radio. Son nom de norme est IEEE 802.11 et c'est le moyen de transmission des données sans fil le plus utilisé.



Répéteur et concentrateur

Lorsqu'un signal parcourt le support physique, son intensité s'atténue avec la distance. Un *répéteur* régénère le signal perçu avec plus d'intensité pour pallier ce problème.

Le *concentrateur* (hub) est moins utilisé de nos jours. C'est une version « multiprise » du répéteur : quand il reçoit un signal sur un des ses branchements, il les recopie sur tous les autres branchements sans se soucier de l'éventuel destinataire du signal.

Carte réseau



Que ce soit une clé USB Wi-Fi ou une carte réseau interne, c'est la même chose : ce composant est indispensable pour connecter un ordinateur à un réseau.

Chaque carte réseau possède une *adresse MAC* : c'est l'adresse physique de la carte, elle permet d'identifier l'ordinateur de manière unique.

3.2. Couche lien

Le *concentrateur* (*switch* en Anglais) est un équipement à plusieurs branchements (au moins 2) appelés *ports* (ne pas confondre avec la notion de port utilisé par une application).

Son rôle est de rediriger une trame reçue vers l'ordinateur de destination.



3.3. Couche réseau



Le *routeur* (*router* en Anglais) permet d'effectuer le routage des paquets et de les faire transiter d'une partie du réseau vers une autre (par exemple d'un réseau local à un autre, nous verrons cela plus tard).

3.4. Couches supérieures

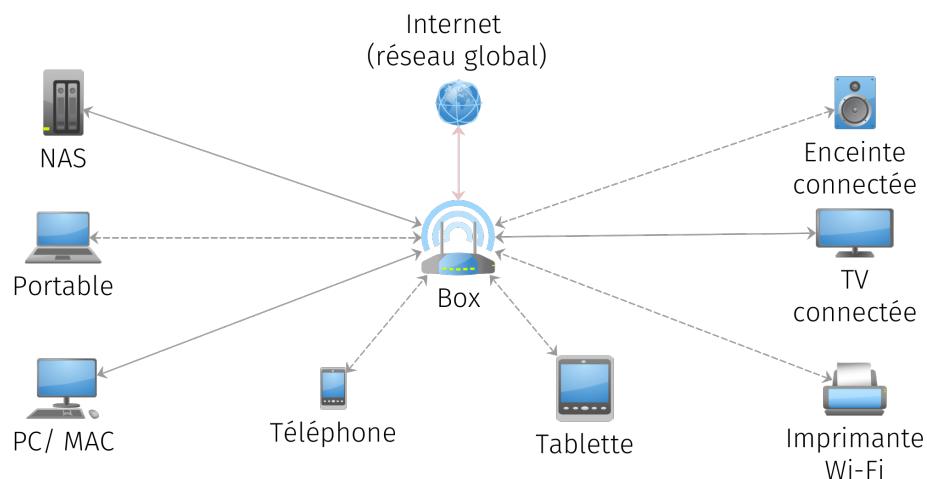
C'est un ordinateur qui exécute les protocoles des couches transport et application. Cela peut être un téléphone, une tablette, un ordinateur portable ou de bureau ou bien encore un objet connecté tel qu'une enceinte Bluetooth.

Chapitre 22

Le réseau local

Un *réseau local*, ou LAN (Local Area Network) est de taille modeste. Les équipements qui y sont connectés s'envoient des données entre eux sans passer par internet. Un bon exemple de LAN est le *réseau domestique* d'une maison :

- la box, fournie par l'opérateur (Orange, Free, Bouygues...) fait office de switch et de routeur;
- les téléphones portables, tablettes, ordinateurs, imprimantes et autres s'y intègrent.



1. Les adresses IP privées et publiques

On a vu dans le chapitre précédent l'importance de l'adresse IP d'un ordinateur dans le rôle de la communication.

Définition : adresse IPv4

Une adresse IP (version 4) est la donnée de 4 octets. On les note séparés par des points. Il y a 5 classes d'adresse IP, notées A, B, C, D et E et les 3

premières classes disposent d'adresses IP (on abrègera en IP) publiques et privées.

Exemples

- 74.125.21.138 est à ce jour une IP permettant d'accéder au site du moteur de recherche de Google. C'est une IP *publique*, accessible par tout le monde *via* Internet.
- 192.168.1.47 est l'IP de l'ordinateur sur lequel j'écris ces lignes. C'est une IP *privée* : elle n'est accessible que par les ordinateurs de mon propre réseau domestique. Elle n'est d'ailleurs pas *unique* non plus dans le sens où d'autres réseaux domestiques utilisent cette IP.
- lorsque je veux voir mon IP publique, je vais par exemple consulter <http://whatismyip.host/> et je trouve une adresse différente : 83.199.117.xx (permettez moi de garder mon adresse IP secrète).

Une adresse IP publique, c'est un peu comme une adresse postale publique : elle identifie de manière unique une machine (qui peut être une box jouant le rôle de routeur vers un réseau domestique).

Une adresse IP privée, c'est un peu comme le numéro et le nom de la rue sans la ville ni le pays : il y a sans doute beaucoup d'adresse au 24 rue des oliviers. Si le réseau postal ne concerne que la ville de Rennes, cette adresse est suffisante, mais si je cherche à envoyer du courrier au 24 rue des oliviers, cela ne marchera pas.

Remarques

- 4 octets pour une adresse IP, c'était bien il y a 30 ans, ça l'est beaucoup moins de nos jours!
- $2^{32} = 4\,294\,967\,296$, donc vu le nombre de machines croissant en fonctionnement simultané sur Terre, il est impossible d'attribuer une IP unique à chaque ordinateur connecté à Internet, d'où l'importance du LAN.
- Pour pallier le problème, une norme IP version 6, plus performante, sur 128 bits au lieu de 32, a vu le jour mais peine encore à s'imposer.

Voici les plages d'IP publiques et privées selon les classes :

- **Classe A** : de l'adresse IP 0.0.0.0 à 126.255.255.255.

Adresses privées : de 10.0.0.0 à 10.255.255.255 (avec 16 millions d'adresses possibles au sein d'un réseau local).

- **Classe B :** de l'adresse IP 128.0.0.0 à 191.255.255.255.

Adresses privées : 172.16.0.0 à 172.31.255.255 (avec 65535 adresses possibles au sein d'un réseau local).

- **Classe C :** de l'adresse IP 192.0.0.0 à 223.255.255.255.

Adresses privées C : 192.168.1.0 à 192.168.255.255 (255 adresses possibles dans un réseau local).

- **Classe D (réservée) :** de l'adresse IP 224.0.0.0 à 239.255.255.255.

- **Classe E (réservée) :** de l'adresse IP 240.0.0.0 à 255.255.255.255.

Exemple

Mon adresse IP publique est une IP de classe A, mon IP privée est de taille C, ce que je comprends parfaitement puisque mon réseau domestique ne contiendra qu'une dizaine de terminaux tout au plus.

2. L'exemple du réseau local traditionnel

Lorsqu'on met en place un réseau local, on commence par déterminer sa taille. Il y a beaucoup de chances qu'on ait moins de 256 machines à connecter donc on va choisir une IP publique de classe C :

- je choisis de prendre pour *adresse réseau* 192.168.1.0, ce n'est pas une IP attribuée à une machine, elle désigne mon réseau local;
- le *masque de sous-réseau* par défaut est 255.255.255.0, ce qui signifie que les 3 premiers octets des machines de mon réseau sont « bloqués » et donc que les machines vont avoir des adresses du type 192.168.1.xx;
- je peux attribuer des IP aux machines que je veux connecter, par exemple 192.168.1.1 pour la première, et cætera;
- la dernière IP 192.168.1.255 est interdite, elle est réservée pour adresser un message à l'ensemble des machines du réseau (on appelle ceci *broadcast*).

Remarque

Prenons le cas d'un foyer qui utilise le FAI Orange. Par défaut le réseau local est 192.168.1.0. avec pour masque 255.255.255.0. La LiveBox, qui fait office de switch et de routeur, a pour adresse 192.168.1.1. C'est cette adresse qui est utilisée comme passerelle pour accéder à internet :

```
C:\Users\UGLi>ipconfig
Configuration IP de Windows

Carte Ethernet :

    Adresse IPv4 . . . . . : 192.168.1.47
    Masque de sous-réseau. . . . . : 255.255.255.0
    Passerelle par défaut. . . . . : fe80::5664:d9ff:febf:70c6%5
                                    192.168.1.1
```

La commande ipconfig de Windows (ifconfig sous Linux et Mac) me permet de retrouver ces informations ainsi que mon IP privée.

Exercice 69

Utilise l'invite de commande de ton système d'exploitation (touche windows et taper cmd pour Windows, Terminal sous MacOS) et trouve l'IP de ton ordinateur, le masque de sous-réseau et l'adresse de la passerelle.

3. Un outil de simulation réseau : Filius

Filius est un logiciel libre fonctionnant sur tous les systèmes d'exploitation et permettant de simuler le fonctionnement d'un petit réseau. On peut

- ajouter du matériel : ordinateur, switch, routeur et modem;
- connecter les éléments précédents;
- configurer ces éléments;
- installer de petites applications sur les ordinateurs connectés pour transmettre ou recevoir des données sur le(s) réseau(x).
- visualiser les échanges de données.

Nous utiliserons Filius pour observer ce que nous avons appris des réseaux.

Chapitre 23

Contrôles de transmission

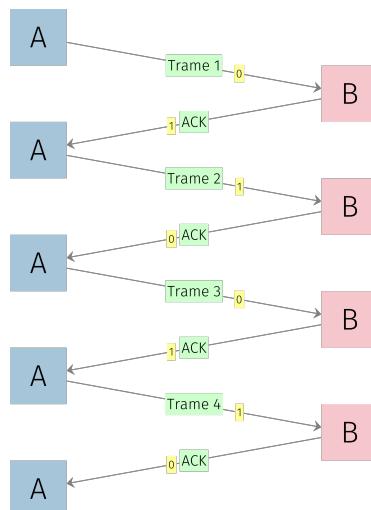
1. Le protocole du bit alterné

Nous allons ici voir un modèle de *contrôle de perte de données* appelé *protocole du bit alterné*. Ce protocole a (ou plutôt avait car il a été remplacé par un protocole plus performant) lieu au sein de la couche 2 (couche lien) et permet de vérifier que les trames d'un ordinateur A sont bien reçues par un ordinateur B.

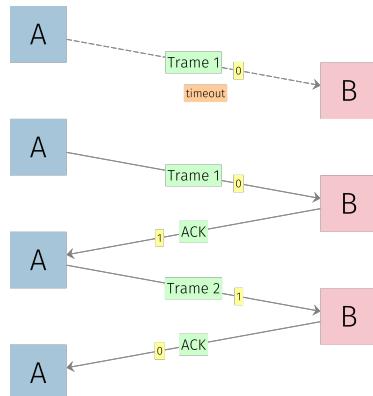
Le principe est très simple, il utilise les *acquittements* et les *flags* : lorsque A envoie une trame, il attend un accusé de réception (*acquittement, acknowledgement* en Anglais) de la part de B dans un temps imparti.

À ceci s'ajoute un bit de contrôle, appelé *flag* en Anglais, qui alterne suivant le modèle suivant :

- la communication commence avec le *flag* à 0, A envoie une première trame avec le *flag*;
- B reçoit la trame et accorde réception en envoyant une trame d'*acquittement* notée ACK. le *flag* est changé à 1;
- A reçoit ACK avec le flag 1 et envoie donc la 2^e trame avec ce *flag* 1;
- et ainsi de suite : Lorsque A reçoit une trame de B, elle garde la valeur du *flag* pour la prochaine trame qu'elle envoie. B, quant à lui change toujours le *flag* entre le moment où il reçoit et celui où il émet.



Ce protocole permet d'éviter la perte de trames dans les cas suivants :



1.1. Perte de trame du côté de A

A envoie la première trame et celle-ci se perd, au bout du temps imparti, ne reçoit rien.

C'est ce qu'on appelle un *timeout* en Anglais.

A renvoie donc sa trame comme si de rien n'était.

1.2. Perte de trame du côté de B

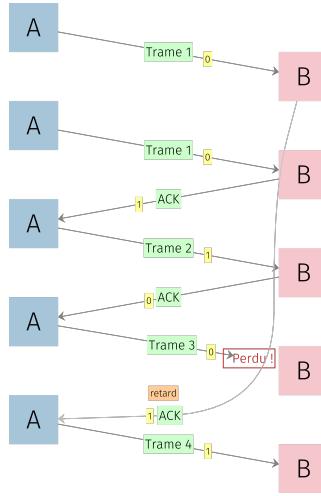
A envoie la première trame et celle-ci arrive à B, qui renvoie un ACK avec un *flag* à 1, et s'attend donc à recevoir une prochaine trame avec un *flag* à 1.

Cette trame ACK se perd donc du point de vue de A, il y a *timeout* et donc il renvoie la même trame avec le *flag* à 0. B se rend compte que quelque chose ne va pas, et renvoie donc l'ACK précédent, avec son *flag* à 1. La communication continue normalement.

Ce protocole présente des insuffisances comme le montre l'exercice suivant

Exercice 70

Analyse le schéma suivant et explique pourquoi il y a perte d'information.



2. Déroulement d'une communication TCP

On rappelle que TCP est un protocole de la couche 4 (couche transport) dont les caractéristiques principales sont les suivantes :

- il commence par établir une connexion entre les deux machines;
 - il découpe les données en paquets;
 - il s'assure de la bonne réception des données au moyen d'*accusés de réception*;
 - il met fin à la connexion.

L'exercice suivant va nous permettre d'examiner une exemple de communication TCP en détail.

Exercice 71

Reprendre le fichier Filius de l'exercice 5 (serveur web avec DNS) de la feuille de TP sur Filius

1. En mode simulation, faire un clic droit sur 192.168.2.1 et afficher les échanges de données.
 2. Normalement il n'y a encore eu aucune communication réseau donc la fenêtre d'échange est vide.
Sur le navigateur web installé sur 192.168.2.1, entrer `monsite.com` et observer la fenêtre d'échange de données *du point de vue de 192.168.2.1*.

Sur le navigateur web installé sur 192.168.2.1, entrer monsite.com et observer la fenêtre d'échange de données *du point de vue de 192.168.2.1*:

On observe 29 trames. Il est possible de cliquer sur chacune d'entre elles pour visualiser son contenu. Voici le contenu de la première :

No.: 1 / Date: 17:59:05.327
Réseau
└─ Source: 3E:A3:A9:87:40:7C
└─ Destination: FF:FF:FF:FF:FF:FF
└─ Commentaire: 0x806
Internet
└─ Source: 192.168.2.1
└─ Destination: 192.168.2.254
└─ Protocole: ARP
└─ Commentaire: Recherche de l'adresse MAC associée à 192.168.2.254, 192.168.2.1: 3E:A3:A9:87:40:7C

Il nous indique que 192.168.2.1 essaie de déterminer l'adresse MAC du routeur. En effet, 192.168.2.1 doit interroger le serveur DNS, situé en 192.168.3.1, pour obtenir l'adresse IP associée à monsite.com, et puisque 192.168.3.1 , n'est pas dans le même réseau que 192.168.2.1, celui-ci utilise la passerelle (le routeur).

La trame suivante est la réponse ARP et la communication se poursuit.

1. Regarder la source, la destination et le contenu des trames 3 et 4. À quoi correspondent-elles ?
2. On s'intéresse au début de la connexion TCP de 192.168.1.2.1 à 192.168.3.1 : ce sont les trames 5,6 et 7, qui constituent ce qu'on appelle en Anglais un *Three-way handshake*. Rechercher ce terme sur Wikipédia et interpréter ensuite les 3 trames.
3. Les trames 8 à 25 constituent l'échange de données en lui-même. Il y a deux grandes étapes. Lesquelles ?
4. Que représentent les trames 26 à 29 ? Détailler le procédé.

Partie III

Le Web

