

Cours d'informatique

Prépas 1

Table des matières

I. INTRODUCTION	3
A. QU'EST-CE QU'UN ORDINATEUR ?	3
1. QU'EST-CE QU'UN ORDINATEUR ?	3
2. ELEMENTS D'ARCHITECTURE D'UN ORDINATEUR	4
3. LE PROCESSEUR (CPU, CENTRAL PROCESS UNIT)	11
4. LES CIRCUITS LOGIQUES	15
B. CODAGE	19
1. ÉCRITURE D'UN ENTIER POSITIF	19
2. REPRESENTER LE SIGNE D'UN ENTIER EN BINAIRE	22
3. REPRESENTATION D'UN TEXTE EN MACHINE	26
C. SYSTEME D'EXPLOITATION C'EST QUOI ?	29
1. INTRODUCTION AUX SYSTEMES D'EXPLOITATION	29
2. QU'EST-CE QU'UN SYSTEME D'EXPLOITATION ?	30
3. DE MULTICS A UNIX	30
4. SYSTEMES PROPRIETAIRES VS SYSTEMES LIBRES	31
5. MICROSOFT	33
6. APPLE	34
D. UN LANGAGE DE PROGRAMMATION C'EST QUOI ?	35
1. QU'EST-CE QU'UN LANGAGE DE PROGRAMMATION ?	35
2. NIVEAU DE LANGAGE	35
3. INTERPRETATION ET COMPIILATION	36
4. PARADIGMES DE PROGRAMMATION	37
5. ENVIRONNEMENT DE DEVELOPPEMENT INTEGRÉ	38
6. CONSOLES INTERACTIVE	39

7. EDITEUR	41
8. DEBOGUEUR	42
II. ALGORITHMES ET PROGRAMMATION EN PYTHON	44
II.UN ALGORITHME	44
1. DEFINITION	44
2. LE LANGAGE	45
3. LES STRUCTURES ELEMENTAIRES	45
4. PROCEDURES, FONCTIONS ET RECURSIVITE	50
II.DEUX PROGRAMMATION EN PYTHON	52
A. TYPES SIMPLES ET EXPRESSIONS	53
1. EXPRESSIONS	53
2. ENTIERS	55
3. FLOTTANTS	55
4. BOOLEENS	56
B. VARIABLES	58
1. IDENTIFICATEURS	58
2. VARIABLES	58
3. INPUT	60
C. ÉCRIRE DES PROGRAMMES PYTHON DANS DES FICHIERS : UTILISATION DE IDLE	61
D. DEBOGAGE DU CODE AVEC IDLE	62
E. LES STRUCTURES CONDITIONNELLES	64
EXERCICE A	65
EXERCICE B	66
F. BOUCLE	67
1. BOUCLE CONDITIONNELLE WHILE (TANT QUE)	67
2. BOUCLE INCONDITIONNELLE FOR... (POUR...)	69
3. BREAK ET CONTINUE	70
EXERCICE	71
G. FONCTIONS	72
1. NOTIONS ET SYNTAXE DE BASE	73
2. VARIABLES LOCALES ET GLOBALES	77
3. TYPAGE DES ARGUMENTS, DU RETOUR - SIGNATURE D'UNE FONCTION	79
EXERCICE	79
H. LISTES	80
1. CONSTRUCTION DE LISTES	81

2. ACCES AUX ELEMENTS	82
3. MODIFICATION D'UN ELEMENT	83
4. SLICING (TRANCHAGE)	84
5. METHODES SUR LES LISTES	84
6. LISTES ET REFERENCES	85
7. LISTES DE LISTES	86
8. LA BOUCLE 'FOR' : PARCOURIR LES ELEMENT D'UN TABLEAU	87
EXERCICE	88

I. Introduction

A. Qu'est-ce qu'un ordinateur ?

1. Qu'est-ce qu'un ordinateur ?

Si on tente rapidement de définir ce qu'est un ordinateur au sens large du terme (y compris tablettes, smartphones...), on peut lister les éléments communs suivants :

- un ordinateur reçoit des informations par l'intermédiaire d'un utilisateur ou d'un réseau ;
- un ordinateur émet des informations via le réseau ou un de ses périphériques ;
- un ordinateur a besoin d'une source d'énergie pour fonctionner.

Néanmoins cette première tentative s'avère infructueuse, on peut penser à plusieurs contre-exemples qui satisfont ces trois critères et qui ne sont pas pour autant des ordinateurs :

- un réfrigérateur nécessite une source d'énergie, il reçoit des informations de la part de capteurs (température...), il en émet sous la forme de signaux lumineux électriques ;
- un interrupteur fonctionnant avec la luminosité ambiante reçoit de l'information par le biais de son capteur et transmet de l'information (ouvert-fermé), de plus le capteur peut nécessiter une source d'énergie pour fonctionner ;
- le système ABS d'aide au freinage d'urgence d'une voiture reçoit également de l'information (vitesse...) et en émet sous forme de pression hydraulique sur le système de freinage ;
- plus généralement, tout système électronique embarqué (système électronique et informatique autonome, ayant une tâche précise) vérifie ces trois critères. Mais l'utilisateur ne peut a priori pas détourner le système pour lui faire exécuter une autre tâche.

Tout cela montre que la définition de ce qu'est un ordinateur n'est pas une chose si triviale que cela. Tout ceci est lié à une des grandes problématiques du siècle dernier : qu'est-ce qu'un calcul ?

2. Eléments d'architecture d'un ordinateur

Nous commençons par décrire le matériel informatique constituant un ordinateur, et permettant son fonctionnement. Notre but est de donner une idée rapide, sans entrer dans le détail logique du fonctionnement du processeur (portes logiques) et encore moins dans le détail électronique caché derrière ce fonctionnement logique (amplificateurs opérationnels, transistors etc.)

2.1. MODELE DE VON NEUMANN

Pour commencer, interrogeons-nous sur la signification-même du terme « informatique »

Définition 1.1.1 (Informatique)

Le mot informatique est une contraction des deux termes information et automatique. Ainsi, l'informatique est la science du traitement automatique de l'information.

Il s'agit donc d'appliquer à un ensemble de données initiales des règles de transformation ou de calcul déterminées (c'est le caractère automatique), ne nécessitant donc pas de réflexion ni de prise d'initiative.

Définition 1.1.2 (Ordinateur)

Un ordinateur est une concrétisation de cette notion.

Il s'agit donc d'un appareil concret permettant le traitement automatique des données. Il est donc nécessaire que l'ordinateur puisse communiquer avec l'utilisateur, pour permettre l'entrée des données initiales, la sortie du résultat du traitement, et l'entrée des règles d'automatisation, sous la forme d'un programme.

Le modèle le plus couramment adopté pour décrire de façon très schématique le fonctionnement d'un ordinateur est celui décrit dans la figure 1.1, appelé *architecture de Von Neumann*. Dans ce schéma, les flèches représentent les flux possibles de données.

Note Historique 1.1.3 (von Neumann)

John von Neumann (János Neumann) est un scientifique américano-hongrois (Budapest, 1903 - Washington, D.C., 1957). Ses domaines de recherche sont très variés, de la mécanique quantique aux sciences économiques, en passant par l'analyse fonctionnelle, la logique mathématique et l'informatique. Il contribue au projet Manhattan, et notamment à l'élaboration de la bombe A, puis plus tard de la bombe H. Son nom reste attaché à la description de la structure d'un ordinateur, en 1945. C'est sur ce schéma qu'ont ensuite été élaborés les premiers ordinateurs.

Si les ordinateurs actuels sont souvent beaucoup plus complexes, leur schéma grossier reste cependant très proche du schéma de l'architecture de von Neumann.

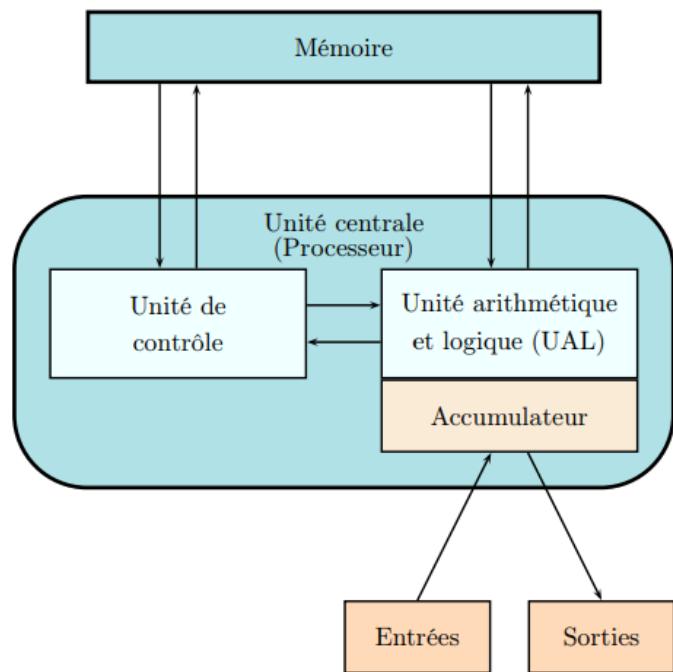
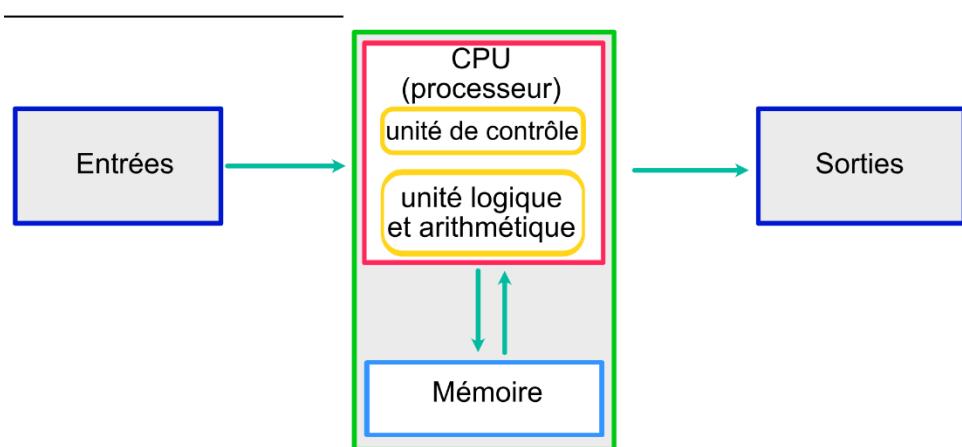


FIGURE 1.1 – Modèle d'architecture de Von Neumann



Note Historique 1.1.4 (architecture de von Neumann)

Le schéma d'un ordinateur (architecture de von Neumann) a été donné en 1945 par John von Neumann, et deux collaborateurs dont les noms sont injustement restés dans l'oubli : John W. Mauckly et John Eckert. John von Neumann lui-même attribue en fait l'idée de cette architecture à Alan Turing, mathématicien et informaticien britannique dont le nom reste associé à la notion de calculabilité (liée à la machine de Turing), ainsi qu'au déchiffrement de la machine Enigma utilisée par les nazis durant la seconde guerre mondiale.



- **Entrées - sorties**

Les entrées et sorties se font au moyen de périphériques spéciaux destinés à cet usage.

- Les périphériques d'entrée permettent à un utilisateur d'entrer à l'ordinateur des données, sous des formats divers : clavier, souris, scanner, webcam, manettes de jeu etc.
 - Les périphériques de sortie permettent de restituer des informations à l'utilisateur. Ils sont indispensables pour pouvoir profiter du résultat du traitement de l'information : écran, imprimante, haut-parleurs, etc.
 - Certains périphériques peuvent parfois jouer à la fois le rôle d'entrée et de sortie, comme les écrans tactiles. Ils peuvent aussi avoir des fonctions non liées aux ordinateurs, comme certaines photocopies, dont l'utilisation essentielle ne requiert pas d'ordinateur, mais qui peuvent aussi faire office d'imprimante et de scanner.
- **La mémoire** permet le stockage des données et des logiciels (programmes) utilisés pour les traiter. Ce stockage peut être :
 - définitif (mémoire morte, ou ROM, inscrite une fois pour toute, et non modifiable, à moins d'interventions très spécifiques),
 - temporaire à moyen et long terme (stockage de données et logiciels que l'utilisateur veut garder, au moins momentanément)
 - temporaire à court terme (données stockées à l'initiative du processeur en vue d'être utilisées ultérieurement : il peut par exemple s'agir de résultats intermédiaires, de piles d'instructions etc.)

L'architecture de Von Neumann utilise le même type de mémoire pour les données et les programmes, ce qui permet la modification des listes d'instructions (elles-mêmes pouvant être gérées comme des données). Ce procédé est à l'origine des boucles.

Nous reparlerons un peu plus loin des différents types de mémoire qui existent.

- **Le processeur** est le cœur de l'ordinateur. C'est la partie de l'ordinateur qui traite l'information. Il va chercher les instructions dans un programme enregistré en mémoire, ainsi que les données nécessaires à l'exécution du programme, il traduit les instructions (parfois complexes) du programme en une succession d'opérations élémentaires, exécutées ensuite par les unités de calcul (UAL(*unité arithmétique*)

et logique) et unité de calcul flottant). Il interagit aussi éventuellement avec l'utilisateur, suivant les instructions du programme.

Nous étudierons un peu plus loin le processeur de façon un peu plus précise, sans pour autant entrer dans les détails logiques associés aux traductions et aux exécutions.

- **Le transfert des données** (les flèches dans le schéma de la figure 1.1) se fait à l'aide de câbles transportant des impulsions électriques, appelés bus.
 - Un bus est caractérisé :
- par le nombre d'impulsions électriques (appelées **bit**) qu'il peut transmettre simultanément. Ce nombre dépend du nombre de conducteurs électriques parallèles dont est constitué le bus. Ainsi, un bus de 32 bits est constitué de 32 fils conducteurs pouvant transmettre indépendamment des impulsions électriques.
- par la fréquence des signaux, c'est-à-dire le nombre de signaux qu'il peut transmettre de façon successive dans un temps donné. Ainsi, un bus de 25 MHz peut transmettre 25 millions d'impulsions sur chacun de ses fils chaque seconde.

Ainsi, un bus de 32 bits et 25 MHz peut transmettre $25 \cdot 10^6 \cdot 32$ bits par seconde, soit $800 \cdot 10^6$ bit par seconde, soit environ 100 Mo (mégaoctet) par seconde (un octet étant constitué de 8 bit). Le « **environ** » se justifie par le fait que les préfixes kilo et méga ne correspondent pas tout-à-fait à 10^3 et 10^6 dans ce cadre, mais à $2^{10} = 1024$ et $2^{20} = 1024^2$.

- Les **bus** peuvent donc transmettre les données à condition que celles-ci soient codées dans un système adapté à ces impulsions électriques. La base 2 convient bien ici (1 = une impulsion électrique, 0 = pas d'impulsion électrique). Ainsi, toutes les données sont codées en base 2, sous forme d'une succession de 0 et de 1 (les bits). Ces bits sont souvent groupés par paquets de 8 (un octet). Chaque demi-octet (4 bits) correspond à un nombre allant de 0 à 15, écrit en base 2. Ainsi, pour une meilleure concision et une meilleure lisibilité, les informaticiens amenés à manipuler directement ce langage binaire le traduisent souvent en base 16 (système hexadécimal, utilisant les 10 chiffres, et les lettres de a à f).

Chaque octet est alors codé par 2 caractères en hexadécimal.

- Les **bus** se répartissent en 2 types : les bus parallèles constitués de plusieurs fils conducteurs, et permettant de transmettre 1 ou plusieurs octets en une fois ; et les bus séries, constitués d'un seul conducteur : l'information est transmise bit par bit.

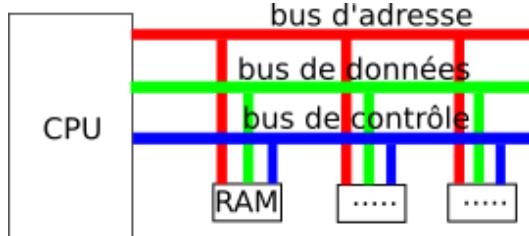
Paradoxalement, il est parfois plus intéressant d'utiliser des bus séries. En effet, puisqu'un bus série utilise moins de conducteur qu'un bus parallèle, on peut choisir, pour un même prix, un conducteur de bien meilleure qualité. On obtient alors, au même coût, des bus séries pouvant atteindre des débits égaux, voire supérieurs, à des bus parallèles.

- Un ordinateur utilise **des bus à 3 usages essentiellement** :

- **le bus d'adresse**, dont la vocation est l'adressage en mémoire (trouver un endroit en mémoire).

C'est un bus unidirectionnel.

- **les bus de données**, permettant la transmission des données entre les différents composants. Ce sont des bus bidirectionnels.
- **les bus de contrôle**, indiquant la direction de transmission de l'information dans un bus de données.



données.

- **La carte-mère** est le composant assurant l'interconnexion de tous les autres composants et des périphériques (via des ports de connexion). Au démarrage, elle lance le **BIOS** (Basic Input/Output system), en charge de repérer les différents périphériques, de les configurer, puis de lancer le démarrage du système via le chargeur d'amorçage (boot loader).

2.2. MEMOIRES

Nous revenons dans ce paragraphe sur un des composants sans lequel un ordinateur ne pourrait rien faire : la mémoire.

La mémoire est caractérisée :

- par sa taille (nombre d'octets disponibles pour du stockage). Suivant le type de mémoire, cela peut aller de quelques octets à plusieurs Gigaoctets ;
- par sa volatilité ou non, c'est-à-dire le fait d'être effacée ou non en absence d'alimentation électrique.
- par le fait d'être réinscriptible ou non (mémoire morte, mémoire vive).

Nous énumérons ci-dessous différents types de mémoire qu'on peut rencontrer actuellement. Du fait de l'importance de la mémoire et des besoins grandissants en capacité de mémoire, les types de mémoire sont en évolution constante, aussi bien par leur forme que par les techniques ou principes physiques utilisés.

Nous ne pouvons en donner qu'une photographie instantanée, et sans doute déjà périmée et loin d'être exhaustive.

- **Mémoire morte (ROM, read-only memory)**

Il s'agit de mémoire non volatile, donc non reprogrammable. Ce qui y est inscrit y est une fois pour toutes, ou presque. Le BIOS, permettant le lancement de l'ordinateur, est sur la ROM de l'ordinateur.

Il s'agit plus spécifiquement d'une EPROM (Erasable programmable Read-Only Memory). Comme son nom l'indique, une EPROM peut en fait être effacée et reprogrammée, mais cela nécessite une opération bien particulière (elle doit être flashée avec des ultraviolets).

- **Mémoire vive (RAM, random access memory)**

- Le nom de la RAM provient du fait que contrairement aux autres types de stockages existant à l'époque où ce nom a été fixé (notamment les cassettes magnétiques), la lecture se fait par accès direct (random), et non dans un ordre déterminé. Le nom est maintenant un peu obsolète, la plupart des mémoires, quelles qu'elles soient, fonctionnant sur le principe de l'accès direct.
- La mémoire vive est *une mémoire volatile*, utilisée par l'ordinateur pour le traitement des données, lorsqu'il y a nécessité de garder momentanément en mémoire un résultat dont il aura à se resservir plus tard. Elle est d'accès rapide, mais peu volumineuse. Elle se présente généralement sous forme de barrettes à enficher sur la carte-mère.
- Physiquement, il s'agit de quadrillages de condensateurs, qui peuvent être dans 2 états (chargé = 1, déchargé = 0). Ces condensateurs se déchargent naturellement au fil du temps. Ainsi, pour garder un condensateur chargé, il faut le recharger (rafraîchir) à intervalles réguliers. Il s'agit du cycle de rafraîchissement, ayant lieu à des périodes de quelques dizaines de nanosecondes. Par conséquent, en l'absence d'alimentation électrique, tous les condensateurs se déchargent, et la mémoire est effacée.

- **Mémoires de masse**

Ce sont des mémoires de grande capacité, destinées à conserver de façon durable de grosses données (bases de données, gros programmes, informations diverses...) De par leur vocation, ce sont nécessairement des mémoires non volatiles (on ne veut pas perdre les données lorsqu'on éteint l'ordinateur !). Par le passé, il s'agissait de bandes perforées, puis de cassettes, de disquettes etc. Actuellement, il s'agit plutôt de disques durs, de bandes magnétiques (fréquent pour les sauvegardes régulières), de CD, DVD, ou de mémoires flash (clé USB par exemple).

- **Mémoires flash**

Les **mémoires flash** (clé USB par exemple) que nous venons d'évoquer ont un statut un peu particulier.

Techniquement parlant, il s'agit de mémoire morte (**EEPROM** : electrically erasable programmable read-only memory), mais qui peut être flashée beaucoup plus facilement que les EPROM, par un processus purement électrique. Ce flashage fait partie du fonctionnement même de ces mémoires, ce qui permet de les utiliser comme des mémoires réinscriptibles et modifiables à souhait.

Une caractéristique très importante de la mémoire est son temps d'accès, qui représente un facteur limitant du temps de traitement de données. Ce temps d'accès est bien entendu dépendant du type de mémoire, ainsi que de sa position par rapport au processeur : même si elle est très grande, la vitesse de circulation des impulsions électriques n'est pas nulle, et loin d'être négligeable dans la situation présente. Mais le processeur lui-même et les composantes auxquels il est rattaché subissent le même principe de miniaturisation en vue de l'augmentation de l'efficacité. Ainsi, il y a physiquement peu de place près du processeur.

Par ailleurs, le temps d'accès dépend beaucoup de la technologie utilisée pour cette mémoire. Les mémoires électroniques, composées de circuits bascules (ou circuits bistables), sont rapides d'accès, tandis que les mémoires à base de transistors et de condensateurs (technologie usuelle des barrettes de RAM) sont plus lentes d'accès (temps de charge + temps de latence dû à la nécessité d'un rafraîchissement périodique, du fait de la

décharge naturelle des condensateurs). Les mémoires externes nécessitant un processus de lecture sont encore plus lentes (disques durs, CD...)

Pour cette raison, les mémoires les plus rapides sont aussi souvent les moins volumineuses, et les plus onéreuses (qualité des matériaux + coût de la miniaturisation), le volume étant d'autant plus limité que d'un point de vue électronique, un circuit bistable est plus volumineux qu'un transistor.



On représente souvent la **hiérarchie des mémoires** sous forme d'un triangle (figure 1.2)

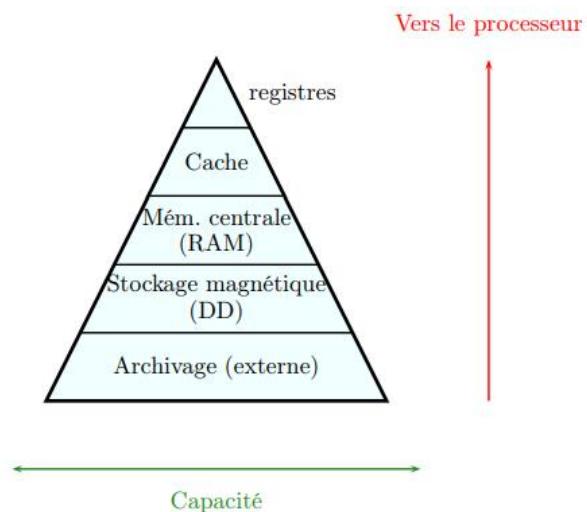


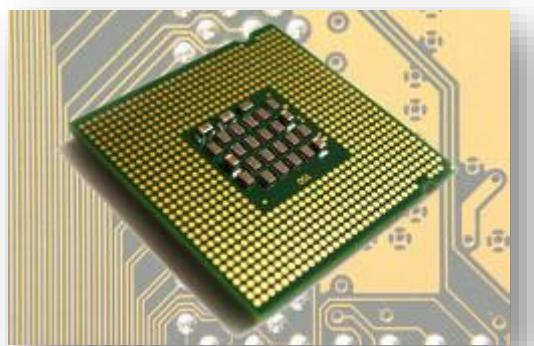
FIGURE 1.2 – Hiérarchie des mémoires

Les registres sont des mémoires situées dans le processeur, ne dépassant souvent pas une dizaine d'octets, et la plupart du temps à usages spécialisés (registres d'entiers, de flottants, d'adresses, compteur ordinal indiquant l'emplacement de la prochaine instruction, registres d'instruction...). Les données stockées dans ces registres sont celles que le processeur est en train d'utiliser, ou sur le point de le faire. Le temps d'accès est très court. Ces mémoires sont le plus souvent constitués de circuits bascule (voir plus loin)

La mémoire cache se décompose souvent en deux parties, l'une dans la RAM, l'autre sur le disque dur. Pour la première, il s'agit de la partie de la RAM la plus rapide d'accès (SRAM). La mémoire cache est utilisée pour stocker momentanément certaines données provenant d'ailleurs ou venant d'être traitées, en vue de les rapprocher, ou les garder près de l'endroit où elles seront ensuite utiles, afin d'améliorer par la suite le temps d'accès. Ainsi, il arrive qu'avant de parvenir au processeur, les données soient d'abord rapprochées sur la mémoire cache.

3. Le processeur (cpu, central process unit)

C'est le **coeur** de l'ordinateur. C'est lui qui réalise les opérations. Dans ce paragraphe, nous nous contenterons d'une description sommaire du principe de fonctionnement d'un processeur. Nous donnerons ultérieurement un bref aperçu de l'agencement électronique (à partir de briques électroniques élémentaires, traduisant sur les signaux électroniques les fonctions booléennes élémentaires) permettant de faire certaines opérations. Nous n'entrerons pas dans le détail électronique d'un processeur.



Composition d'un processeur

Le processeur est le calculateur de l'ordinateur. Il lit les instructions, les traduit en successions d'opérations élémentaires qui sont ensuite effectuées par l'unité arithmétique et logique (UAL), couplée (actuellement) à une unité de calcul en virgules flottantes (format usuellement utilisé pour les calculs sur les réels).

Il est constitué d' :

- **une unité de traitement**, constituée d'une **UAL** (unité arithmétique et logique), d'un registre de données (mémoire destinée aux données récentes ou imminentes), et d'un accumulateur (l'espace de travail). Le schéma global est celui de la figure 1.3.

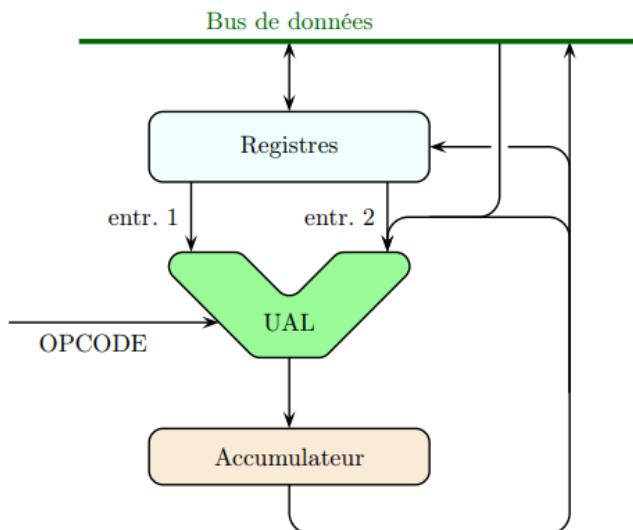


FIGURE 1.3 – Schéma d'une unité de traitement

Ce schéma correspond au cas d'une machine à plusieurs adresses (l'UAL peut traiter simultanément des données situées à plusieurs adresses). Il existe également des machines à une seule adresse : la deuxième donnée nécessaire aux calculs étant alors celle stockée dans l'accumulateur. Cela nécessite de décomposer un peu plus les opérations élémentaires (notamment dissocier le chargement de l'accumulateur du calcul lui-même).

Le code de l'opération (OPCODE) correspond à une succession de bits indiquant la nature de l'opération à effectuer sur l'entrée, ou les entrées (par exemple, un certain code correspond à l'addition, un autre à la multiplication, d'autres aux différents tests booléens etc.)

Une instruction élémentaire doit donc arriver à l'unité de traitement sous la forme d'un code (une succession de bits, représentés par des 0 et des 1, correspondant en réalité à des impulsions électriques), donnant d'une part le code de l'opération (OPCODE), d'autre part les informations nécessaires pour trouver les entrées auxquelles appliquer cette opération (la nature de ces informations diffère suivant le code de l'opération : certains opérations demandent 2 adresses, d'autres une seule adresse et la donnée d'une valeur « *immédiate* », d'autres encore d'autres formats).

Ce codage des instructions diffère également d'un processeur à l'autre. Par exemple, dans un processeur d'architecture MIPS, l'opération est codée sur 6 bits.

L'opération 100000 demande ensuite la donnée de 3 adresses a_1 , a_2 et a_3 . Cette opération consiste en le calcul de $a_2 + a_3$, le résultat étant ensuite stocké à l'adresse a_1 :

100000	a_1	a_2	a_3	$\$a_1 \leftarrow \$a_2 + \$a_3$
--------	-------	-------	-------	----------------------------------

Dans cette notation, $\$a$ représente la valeur stockée à l'adresse a . L'opération 001000 est également une opération d'addition, mais ne prenant que deux adresses a_1 et a_2 , et fournissant de surcroît une valeur i directement codée dans l'instruction (valeur immédiate). Elle réalise l'opération d'addition de la valeur i à la valeur stockée à l'adresse a_2 , et va stocker le résultat à l'adresse a_1 :

001000	a ₁	a ₂	i	
--------	----------------	----------------	---	--

$$\$a_1 \leftarrow \$a_2 + i$$

L'UAL est actuellement couplée avec une unité de calcul en virgule flottante, permettant le calcul sur les réels (ou plutôt leur représentation approchée informatique, dont nous reparlerons plus loin). Un processeur peut avoir plusieurs processeurs travaillant en parallèle (donc plusieurs UAL), afin d'augmenter la rapidité de traitement de l'ordinateur ;

- **Une unité de contrôle**, qui décode les instructions d'un programme, les transcrit en une succession d'instructions élémentaires compréhensibles par l'unité de traitement (suivant le code évoqué précédemment), et envoie de façon bien coordonnée ces instructions à l'unité de traitement. La synchronisation se fait grâce à l'horloge : les signaux d'horloge (impulsions électriques) sont envoyés de façon régulière, et permettent de cadencer les différentes actions. De façon schématique, la réception d'un signal d'horloge par un composant marque l'accomplissement d'une étape de la tâche qu'il a à

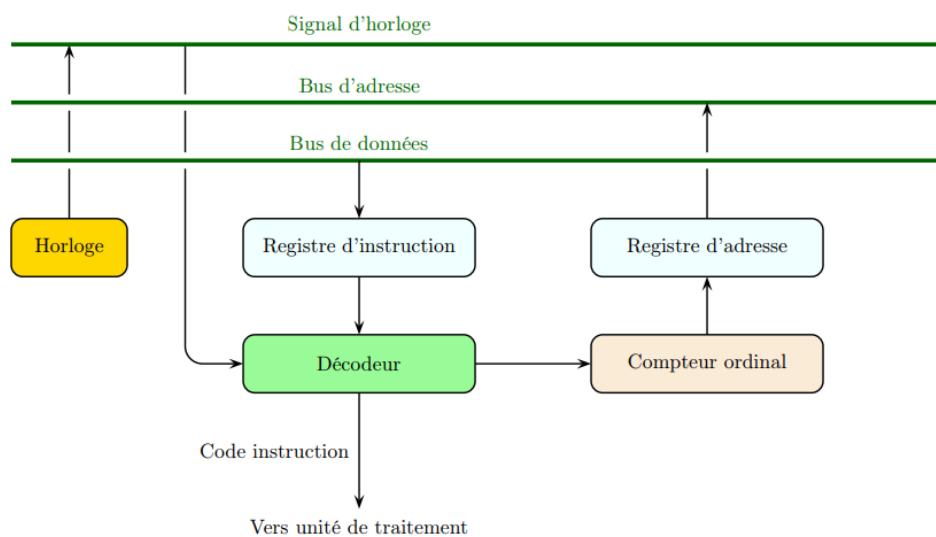


FIGURE 1.4 – Schéma d'une unité de contrôle

faire. Les différentes étapes se succèdent donc au rythme des signaux d'horloge. Une unité de contrôle peut être schématisée à la manière de la figure 1.4.

Le registre d'instruction contient l'instruction (non élémentaire) en cours, le compteur ordinal contient ce qu'il faut pour permettre de trouver l'adresse de l'instruction suivante, et le registre d'adresse contient l'adresse de l'instruction suivante. Le décodeur décode l'instruction contenue dans le registre d'adresse et la traduit en une succession d'instructions élémentaires envoyées à l'unité de traitement au rythme de l'horloge. Lorsque l'instruction est entièrement traitée, l'unité va chercher l'instruction suivante, dont l'adresse est stockée dans le registre d'adresse, et la stocke dans le registre d'instruction. Le décodeur actualise le compteur ordinal puis l'adresse de l'instruction suivante.

Décomposition de l'exécution d'une instruction

On peut schématiquement décomposer l'exécution d'une instruction par l'unité de traitement en 4 étapes :

- ❖ Réception de l'instruction codée (FETCH) : L'instruction, provenant de l'unité de contrôle, est reçue.

- ❖ Décodage de l'instruction (DECODE) : Les différentes données sont envoyées aux composants concernés, en particulier l'OPCODE est extrait et envoyé à l'UAL (Unité Arithmétique et Logique).
- ❖ Exécution de l'instruction (EXECUTE) : L'instruction est exécutée par l'UAL.
- ❖ Écriture du résultat dans une mémoire (WRITEBACK) : Le résultat est écrit dans une mémoire.

Étape	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Instr. 1	FE	DE	EX	WR												
Instr. 2					FE	DE	EX	WB								
Instr. 3									FE	DE	EX	WB				
Instr. 4													FE	DE	EX	WB

Si nous avons 4 instructions à exécuter successivement, cela nous donne donc 16 étapes successives :

On se rend compte que l'UAL n'est utilisée qu'aux étapes 3, 7, 11 et 15 (EXECUTE), n'exploitant pas pleinement ses capacités. En supposant les instructions indépendantes les unes des autres, et en négligeant les dépendances d'adresses, on peut imaginer le fonctionnement suivant, sans attendre la fin de l'instruction précédente pour commencer la suivante :

Étape	1	2	3	4	5	6	7
Instr. 1	FE	DE	EX	WR			
Instr. 2		FE	DE	EX	WB		
Instr. 3			FE	DE	EX	WB	
Instr. 4				FE	DE	EX	WB

On dit que le processeur est muni d'un pipeline s'il permet d'effectuer ces opérations en parallèle de façon décalée. Un pipeline doit aussi gérer les problèmes de dépendance, nécessitant parfois la mise en attente d'une instruction pour attendre le résultat d'une autre. Dans cet exemple, nous avons un pipeline à 4 étages (nombre d'étapes dans la décomposition de l'exécution d'une instruction, correspondant aussi au nombre maximal d'instructions pouvant être traitées simultanément). Des décompositions plus fines peuvent aboutir à des pipelines à plus d'étages (une dizaine).

De plus, certains processeurs possèdent plusieurs coeurs, permettant l'exécution simultanée d'instructions. Supposons un processeur à 2 coeurs avec un pipeline à 4 étages pour exécuter 4 instructions indépendantes :

Étape	1	2	3	4	5
Instr. 1	FE 1	DE 1	EX 1	WR 1	
Instr. 2	FE 2	DE 2	EX 2	WB 2	
Instr. 3		FE 1	DE 1	EX 1	WB 1
Instr. 4		FE 2	DE 2	EX 2	WB 2

On parle dans ce cas d'architecture en parallèle. Le parallélisme demande également un traitement spécifique des dépendances.

Puissance de calcul d'un processeur

La puissance d'un processeur est calculée en FLOPS (Floating Point Operation Per Second). Ainsi, il s'agit du nombre d'opérations qu'il est capable de faire en une seconde sur le format flottant (réels). Pour donner des ordres de grandeur, voici l'évolution de la puissance sur quelques années références :

- 1964 : 1 mégaFLOPS (10^6)
- 1997 : 1 téraFLOPS (10^{12})
- 2008 : 1 pétaFLOPS (10^{15})
- 2013 : 30 pétaFLOPS

On estime que l'exaFLOPS (10^{18}) devrait être atteint vers 2020. Ces puissances concernent les super-calculateurs. La puissance des ordinateurs personnels est bien inférieure. Par exemple, en 2013, un ordinateur personnel avait en moyenne un processeur de 200 gigaFLOPS, comparable à celle des super-calculateurs de 1995.

4. Les circuits logiques

Le transistor est l'élément de base des circuits logiques. Un circuit logique permet de réaliser une opération booléenne. Ces opérations booléennes sont directement liées à l'algèbre de Boole, développée par le mathématicien britannique George Boole (1815-1864). L'étude approfondie de l'algèbre de Boole dépasse le cadre de ce cours, mais vous devez savoir qu'un circuit logique reçoit en entrée un ou plusieurs signaux électriques (chaque entrée étant dans un état "**haut**" (symbolisé par un "1") ou un état "**bas**" (symbolisé par un "0")) et produit en sortie un ou plusieurs signaux électriques (chaque sortie étant également dans un état "**haut**" ou "**bas**").

Il existe deux catégories de circuits logiques :

- **Les circuits combinatoires** : les états en sortie dépendent uniquement des états en entrée.
- **Les circuits séquentiels** : les états en sortie dépendent des états en entrée ainsi que du temps et des états antérieurs.

Dans la suite de ce cours, nous nous intéresserons principalement aux circuits combinatoires.

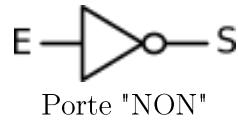
Étant donné que l'information circule sous forme d'impulsions électriques, la base de représentation privilégiée est la base 2. Mais comment, en pratique, effectuer les calculs élémentaires sur ces représentations en base 2 ? Autrement dit, comment, à partir d'impulsions électriques correspondant à deux arguments, récupérer les impulsions électriques correspondant, par exemple, à la somme ? Le but de cette section est de fournir quelques éléments de réponse basés sur l'étude des portes logiques, sans entrer dans le détail électronique de ces portes, et en se concentrant sur un exemple d'opération : le cas de l'addition.

La porte NON (NOT)

Le plus simple des circuits combinatoires est la porte "**NON**" ("**NOT**" en anglais) qui inverse l'état en entrée : si l'entrée de la porte est dans un état "**bas**" alors la sortie sera dans un état "**haut**" et vice versa. Si on symbolise l'état "**haut**" par un "1" et l'état "**bas**" pour un "0", on peut obtenir ce que l'on appelle la table de vérité de la porte "**NON**" :

E (Entrée)	S (Sortie)
1	0
0	1

La porte "NON" est symbolisée par le schéma suivant :



✚ La porte OU (OR)

La porte "OU" a deux entrées (E1 et E2) et une sortie S

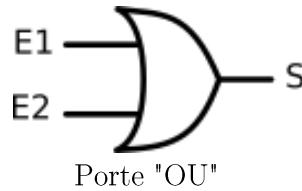


Table de vérité porte "OU" :

E1	E2	S
0	0	0
0	1	1
1	0	1
1	1	1

✚ La porte ET (AND)

La porte "ET" ("AND") a deux entrées (E1 et E2) et une sortie S

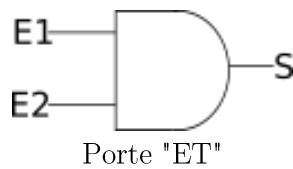


Table de vérité porte "ET" :

E1	E2	S
0	0	0
0	1	0
1	0	0
1	1	1

➊ La porte OU EXCLUSIF (XOR)

La porte "OU EXCLUSIF" ("XOR") a deux entrées (E1 et E2) et une sortie S



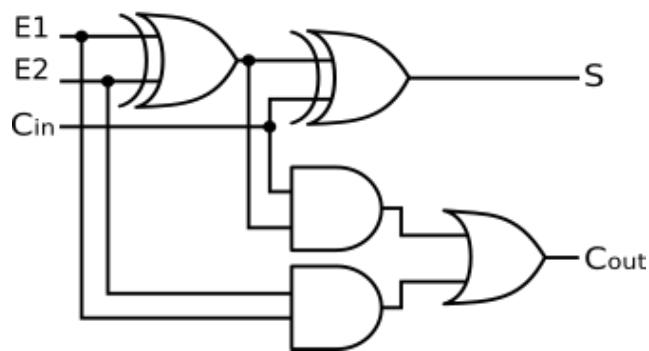
Porte "OU EXCLUSIF"

Table de vérité porte "XOR" :

E1	E2	S
0	0	0
0	1	1
1	0	1
1	1	0

➋ L'additionneur

En combinant les portes logiques, on obtient des circuits plus complexes. Par exemple en combinant 2 portes "OU EXCLUSIF", 2 portes "ET" et une porte "OU" on obtient un additionneur :



Additionneur

Comme son nom l'indique, l'additionneur permet d'additionner 2 bits (E1 et E2) en tenant compte de la retenue entrante ("Cin" "carry in" en anglais). En sortie on obtient le résultat de l'addition (S) et la retenue sortante ("Cout").

Établir la table de vérité de l'additionneur en complétant le tableau ci-dessous

E1	E2	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

En combinant plusieurs fois le type de circuit décrit ci-dessus, on obtient des additionneurs capables d'additionner des nombres sur X bits.

Une chose est très importante à bien comprendre : à la base nous avons le transistor, une combinaison de transistor (sous forme de circuit intégré) permet d'obtenir des circuits logiques, la combinaison de circuits logiques permet d'obtenir des circuits plus complexes (exemple : l'additionneur), et ainsi de suite...

Au sommet de cet édifice (on pourrait parler de poupée russe), nous allons trouver la mémoire vive (RAM) et le microprocesseur (CPU).

B. CODAGE

1. ÉCRITURE D'UN ENTIER POSITIF

a. Le système décimal

Pour comprendre le fonctionnement du binaire et des systèmes de comptage en général (bases), une réintroduction à la base 10 est nécessaire. Tout le monde sait compter en base 10 (décimal), mais comment cela fonctionne-t-il ?

La base 10 utilise dix chiffres (0 à 9). Si l'on veut aller au-delà de 9, il faut changer de rang (unités, dizaines, centaines, etc.). Par exemple, pour écrire 19, le rang des unités est saturé avec le chiffre 9, il faut donc incrémenter le rang des dizaines et réinitialiser le rang des unités. Un nombre entier est donc composé de rangs où chaque rang vaut le rang précédent multiplié par l'indice de la base (10). Par exemple, 185 en base 10 se décompose comme suit :

$$185 = 1 * 10^2 + 8 * 10^1 + 5 * 10^0$$

b. Le binaire

Introduction

Le binaire est le système de comptage des ordinateurs car les circuits électroniques utilisent des signaux électriques, représentés simplement par 0 (pas de courant) et 1 (courant). En binaire, chaque rang (bit) ne peut avoir que deux valeurs : 0 ou 1. Chaque rang en binaire correspond à une puissance de 2. Voici comment compter en binaire jusqu'à 10 :

Nombre en décimal	Nombre en binaire	Le pourquoi du comment
0	0	Pour l'instant, ça va.
1	1	Là encore, c'est simple.
2	10	Le premier rang ayant été rempli, on passe au suivant !
3	11	On re-remplit le rang 1.
4	100	Le rang 2 est plein, le rang 1 aussi, qu'à cela ne tienne, on passe au suivant.
5	101	On continue en suivant la même méthode.

6	110	
7	111	
8	1000	On commence le rang 4.
9	1001	
10	1010	On continue comme tout à l'heure.
		...

Il est important de retenir qu'il faut entamer le rang suivant quand l'actuel est plein.

Activité 1

- Combien de valeurs peut-on coder avec 1 bit ?
- Combien de valeurs peut-on coder avec 2 bits ?
- Combien de valeurs peut-on coder avec 3 bits ?
- Combien de valeurs peut-on coder avec n bits ?

c. Conversion décimale vers binaire

Pour convertir des nombres décimaux en binaire, la méthode euclidienne est souvent utilisée. Elle consiste à diviser le nombre par 2, noter le reste, et réitérer avec le quotient jusqu'à obtenir un quotient de 0. Les restes, lus de bas en haut, forment le nombre binaire.

Exemple :

185 en base 10 vaut 10111001 en binaire.

Activité 2

- Convertir 42 en base 2

d. Conversion binaire vers décimal

Pour convertir un nombre binaire en décimal, chaque rang binaire est multiplié par la puissance de 2 correspondante, puis les résultats sont additionnés.

Exemple

11010011 en binaire vaut 211 en décimal.

$$11010011_{\text{binair}} = 1 * 1 + 1 * 2 + 0 * 4 + 0 * 8 + 1 * 16 + 0 * 32 + 1 * 64 + 1 * 128 = 211_{\text{décimal}}$$

Activité 3

- Convertir 10011001 en base 10

L'hexadécimal

Introduction

L'hexadécimal (base 16) est souvent utilisé en informatique car il permet de représenter des informations sur 8 bits avec seulement 2 chiffres. En hexadécimal, on utilise les chiffres 0 à 9 et les lettres A à F.

Binaire (base 2)	Décimal (base 10)	Hexadécimal (base 16)
0	0	0
1	1	1
10	2	2
11	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

e. Conversion binaire vers hexadécimal (et vice versa)

Pour convertir un nombre binaire en hexadécimal, on regroupe les bits par groupes de 4. Inversement, chaque chiffre hexadécimal se convertit en un groupe de 4 bits.

Exemple :

11011001 en binaire vaut D9 en hexadécimal.

Activités

- Activité 4 : Convertir 111011 en base 16
- Activité 5 : Convertir 101111011001 en base 16
- Activité 6 : Convertir 1A3E en base 2

f. Conversion décimale vers hexadécimal

La méthode est similaire à la conversion vers le binaire, mais on divise par 16.

Exemple :

185 en base 10 vaut B9 en hexadécimal.

Activité 7

- Convertir 1387 en base 16

g. Conversion hexadécimale vers décimal

Pour convertir un nombre hexadécimal en décimal, on utilise des puissances de 16. Chaque chiffre hexadécimal est multiplié par la puissance de 16 correspondante, puis les résultats sont additionnés.

Exemple :

12B7 en hexadécimal vaut 4791 en décimal.

$$\begin{aligned}12B7_{16} &= 1 \times 16^3 + 2 \times 16^2 + 11 \times 16^1 + 7 \times 16^0 \\&= 1 \times 4096 + 2 \times 256 + 11 \times 16 + 7 \\&= 4096 + 512 + 176 + 7 \\&= 4791\end{aligned}$$

Activité 8

- Convertir 1B87 en base 10

h. Généralisation pour une base b

Nous avons vu ci-dessus comment convertir un nombre en base 10 en un nombre en base 16. Pour convertir un nombre n (en base 10) en sa représentation en base b , il faut suivre l'algorithme suivant :

- On appelle D la liste des chiffres (le résultat).
- Tant que $n > 0$
 - Faire la division euclidienne de n par b .
 - Ajouter le reste à D
 - Mettre le quotient dans n
- Inverser l'ordre de D

Soit x la représentation du nombre en base 10, r la représentation du nombre en base b , r étant écrit en utilisant n chiffres : C_0, C_1, \dots, C_{n-1} , nous avons :

$$r = C_{n-1} C_{n-2} \dots C_2 C_1 C_0$$

$$x = \sum_{i=0}^{n-1} C_i b^i = C_{n-1} b^{n-1} + C_{n-2} b^{n-2} + \dots + C_2 b^2 + C_1 b^1 + C_0 b^0$$

2. REPRESENTER LE SIGNE D'UN ENTIER EN BINAIRE

Jusqu'ici nous n'avons vu que les nombres entiers positifs, Comme le binaire n'autorisent que deux symboles, « 0 » ou « 1 », il fallait trouver un moyen de représenter le signe des entiers **relatifs** (positifs et négatifs).

La solution la plus évidente a été utilisée pour les premiers ordinateurs : utiliser le bit de poids fort pour le signe avec la convention suivante :

- Valeur « **1** » pour les entiers **négatifs**
- Valeur « **0** » pour les entiers **positifs**

Le premier bit représente donc le signe et les bits suivants représentent la valeur absolue du nombre.

Un exemple : on représente l'entier 5 sur 8 bits par 00000101, -5 serait donc représenté par 10000101

Activité 1

En utilisant la méthode décrite ci-dessus, représentez -15 (représentation sur 8 bits)

Ainsi, le nombre binaire relatif codé sur un octet suivant (le rang est en dessous) :

$$\begin{aligned}(10010100) &= -(24+22) = -20 \\(76543210) &\end{aligned}$$

On voit ici que le rang de poids fort (7) est utilisé pour le signe et vaut « **1** » donc le nombre est négatif.

Une fois le signe établi, on repère les « **1** » dans le reste du nombre et on additionne les 2 puissances « le rang du **1** » pour savoir de quel nombre il s'agit.

Cette méthode de codage a un inconvénient majeur : deux nombres représentent le zéro (sur 4bit par exemple : $(1000)_2 = (0000)_2 = (0)_{10}$). Pour traiter cela, il faut des circuits différents pour l'addition et la soustraction et cette méthode coûteuse en ressource a été remplacée par une méthode plus efficace : le complément à deux.

Le complément à deux

Avant de représenter un entier relatif, il est nécessaire de définir le nombre de bits qui seront utilisés pour cette représentation (**souvent 8, 16 , 32 ou 64 bits**)

Pour le complément à deux, on continue à regarder le rang de poids fort :

- Valeur « **0** » pour les entiers **positifs** : on prend les bits suivants pour déterminer la valeur du nombre.
- Valeur « **1** » pour les entiers **négatifs** :
 - On commence par représenter en binaire la valeur absolue du nombre.
 - On inverse ensuite tous les bits de ce nombre, cette opération est appelée « complément à 1 ».

- Enfin on ajoute (par addition binaire, voir plus loin) « 1 » au nombre ainsi obtenu.

signe $(0101)_2 = (+5)_{10}$
valeur absolue
(ici $2^2 + 2^0 = 5$)

(-5)₁₀
On écrit +5 en base 2 : 0101
On inverse les bit : 1010
On ajoute 1 : 1011
(-5)₁₀ = (1011)₂

Pour connaître la valeur d'un nombre binaire entier négatif (qui commence donc par « 1 ») on procède exactement de la même façon **dans le même ordre** : inversion des bits puis ajout (addition binaire) de « 1 » pour connaître la valeur absolue positive :

que vaut (1011)₂

On inverse les bit : 0100

On ajoute 1 : 0101 → $(0101)_2 = (5)_{10}$

donc (1011)₂ = (-5)₁₀

Prenons en suite un autre exemple : déterminons la représentation de -12 sur 8 bits

- Commençons par représenter 12 sur 8 bits (sachant que pour représenter 12 en binaire seuls 4 bits sont nécessaires, les 4 bits les plus à gauche seront à 0) : 00001100
- Inversons tous les bits (les bits à 1 passent à 0 et vice versa) : 11110011

$$\begin{array}{r} \textcolor{red}{11} \\ 11110011 \\ + 00000001 \\ \hline \end{array}$$

- Ajoutons 1 au nombre obtenu à l'étape précédente : $\begin{array}{r} \textcolor{red}{11} \\ 11110011 \\ + 00000001 \\ \hline 11110100 \end{array}$ les retenues sont notées en rouge
- La représentation de -12 sur 8 bits est donc : 11110100

Comment peut-on être sûr que 11110100 est bien la représentation de -12 ?

Nous pouvons affirmer sans trop de risque de nous tromper que $12 + (-12) = 0$, vérifions que cela est vrai pour notre représentation sur 8 bits.

$$\begin{array}{r} \textcolor{red}{1111} \\ 00001100 \\ + 11110100 \\ \hline 00000000 \end{array}$$

Dans l'opération ci-dessus, nous avons un 1 pour le 9e bit, mais comme notre représentation se limite à 8 bits, il nous reste bien 00000000.

Activité 2

En utilisant le complément à 2, représentez -15 (représentation sur 8 bits)

Il faut noter qu'il est facile de déterminer si une représentation correspond à un entier positif ou un entier négatif : si le bit de poids fort est à 1, nous avons affaire à un entier négatif, si le bit de poids fort est à 0, nous avons affaire à un entier positif.

Avec cette méthode de codage, une seule valeur représente 0 : sur 4 bits par exemple ce sera $(0000)_2 = (0)_{10}$

Voici les valeurs sur 3 bits en complément à deux

Binaire	décimal	binaire	décimal
000	0	100	-4
001	1	101	-3
010	2	110	-2
011	3	111	-1

La limite de cette méthode de codage des nombres relatifs réside dans un risque **d'erreurs de calcul** si le codage se fait sur un nombre trop restreint de bits.

Par exemple, sur 3 bits, si on effectue l'opération $(3+1)_{10} = (011+1=100)_2 = (-4)_{10}$!

De même, l'opération $(-4-1)_{10} = (100+111=(1)011)_2 = (3)_{10}$! Sur 4 bits, ces deux opérations ne présentaient pas d'erreur de calcul.

Dans les deux cas, on a dépassé les limites du codage possible sur 3bit, on parle alors de « débordement ». Pour éviter ce phénomène qui peut entraîner de graves erreurs de calcul, on a intérêt à **coder les entiers sur un nombre de bits plus grand** que la plus grande valeur qu'on va utiliser de façon que le bit de poids fort reste réservé au seul signe.

Vous pouvez trouver un convertisseur en ligne pour vérifier vos calculs sur ce site : <https://sebastienguillon.com/test/javascript/convertisseur.html>

Activité 3

Représentez sur 8 bits l'entier 4 puis représentez, toujours sur 8 bits, l'entier -5. Additionnez ces 2 nombres (en utilisant les représentations binaires bien évidemment), vérifiez que vous obtenez bien -1.

Activité 4

Quel est le plus petit entier négatif que l'on peut représenter sur 8 bits ?

Activité 5

Quel est le plus grand entier positif que l'on peut représenter sur 8 bits ?

Dans le "Activité 4" vous avez dû normalement trouver 10000000 (soit -128) et 01111111 (soit 127) dans le "Activité 5". Plus généralement, nous pouvons dire que pour une représentation sur n bits, il sera possible de coder des valeurs comprises entre -2^{n-1} et $+2^{n-1} - 1$

Activité 6

Quelles sont les bornes inférieure et supérieure d'un entier relatif codé sur 16 bits ?

3. REPRESENTATION D'UN TEXTE EN MACHINE

a. INTRODUCTION

Nous savons qu'un ordinateur est uniquement capable de traiter des données binaires. Comment sont donc codés les textes dans un ordinateur ? Ou plus précisément, comment sont codés les caractères dans un ordinateur ?

Le codage d'un texte sur ordinateur consiste donc à transformer chaque lettre en son code binaire afin de pouvoir effectuer le stockage et le traitement dans un ordinateur. Afin de simplifier l'écriture, on peut ensuite convertir des paquets de 8 bits (octets) en écriture hexadécimale, comme vu dans le chapitre

"REPRÉSENTATION DES DONNÉES : TYPES ET VALEURS DE BASE / ÉCRITURE D'UN ENTIER POSITIF".

b. ASCII

Avant 1960, de nombreux systèmes de codage de caractères existaient, ils étaient souvent incompatibles entre eux. En 1960, l'organisation internationale de normalisation (ISO) décide de mettre un peu d'ordre dans ce bazar en créant la norme ASCII (*American Standard Code for Information Interchange*). À chaque caractère est associé un nombre binaire sur 8 bits (1 octet). En fait, seuls 7 bits sont utilisés pour coder un caractère, le 8e bit n'est pas utilisé pour le codage des caractères. Avec 7 bits, il est possible de coder jusqu'à 128 caractères, ce qui est largement suffisant pour un texte écrit en langue anglaise (pas d'accents et autres lettres particulières).

Comme vous pouvez le constater dans le tableau ci-dessus, au "A" majuscule correspond le code binaire 1000001₂ (65₁₀ ou 41₁₆).

ASCII TABLE

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

Activité 1

Quel est le code binaire du "a" minuscule en ASCII?

Comme vous pouvez le constater, certains codes ne correspondent pas à des caractères (de 0 à 32₁₀), nous n'aborderons pas ce sujet ici.

c. ISO-8859-1

La norme ASCII convient bien à la langue anglaise, mais pose des problèmes dans d'autres langues, par exemple le français. En effet, l'ASCII ne prévoit pas d'encoder les lettres accentuées. C'est pour répondre à ce problème qu'est née la norme ISO-8859-1. Cette norme reprend les mêmes principes que l'ASCII, mais les nombres binaires associés à chaque caractère sont codés sur 8 bits, ce qui permet d'encoder jusqu'à 256 caractères. Cette norme va être principalement utilisée dans les pays européens puisqu'elle permet d'encoder les caractères utilisés dans les principales langues européennes (la norme ISO-8859-1 est aussi appelée "latin1" car elle permet d'encoder les caractères de l'alphabet dit "latin").

Problème : il existe beaucoup d'autres langues dans le monde qui n'utilisent pas l'alphabet dit "latin", par exemple le chinois ou le japonais ! D'autres normes ont donc dû voir le jour, par exemple la norme "GB2312" pour le chinois simplifié ou encore la norme "JIS_X_0208" pour le japonais.

Cette multiplication des normes a très rapidement posé problème. Imaginons un français qui parle le japonais. Son traitement de texte est configuré pour reconnaître les caractères de l'alphabet "latin" (norme ISO-8859-1). Un ami japonais lui envoie un fichier texte écrit en japonais. Le français devra modifier la configuration de son traitement afin que ce dernier puisse afficher correctement l'alphabet japonais. S'il n'effectue pas ce changement de configuration, il verra s'afficher des caractères ésotériques.

d. **Unicode**

Pour éviter ce genre de problème, en 1991 une nouvelle norme a vu le jour : Unicode.

Unicode a pour ambition de rassembler tous les caractères existant afin qu'une personne utilisant Unicode puisse, sans changer la configuration de son traitement de texte, à la fois lire des textes en français ou en japonais.

Unicode est uniquement une table qui regroupe tous les caractères existant au monde, il ne s'occupe pas de la façon dont les caractères sont codés dans la machine. Unicode accepte plusieurs systèmes de codage : UTF-8, UTF-16, UTF-32. Le plus utilisé, notamment sur le Web, est UTF-8.

Pour encoder les caractères Unicode, UTF-8 utilise un nombre variable d'octets : les caractères "classiques" (les plus couramment utilisés) sont codés sur un octet, alors que des caractères "moins classiques" sont codés sur un nombre d'octets plus important (jusqu'à 4 octets). Un des avantages d'UTF-8 est qu'il est totalement compatible avec la norme ASCII : Les caractères Unicode codés avec UTF-8 ont exactement le même code que les mêmes caractères en ASCII.

Activité 2

Quel est le code binaire du "b" minuscule Unicode codé avec UTF-8 ?

e. **Passer d'un codage à un autre**

Dans la norme UTF-8, les bits de poids fort du premier octet (à gauche) vont indiquer le nombre d'octets utilisés pour le caractère en mettant un « 1 » pour chaque octet à partir de 2. Tous les octets suivants commencent par la séquence « 10 » :

- Codage sur 1 octet : « 0x xx xx xx » correspond à la table ASCII de base sur 7 bits.
- 2 octets : « 11 0x xx xx 10 xx xx xx »
- 3 octets : « 11 10 xx xx 10 xx xx xx 10 xx xx xx »
- 4 octets : « 11 11 0x xx 10 xx xx xx 10 xx xx xx 10 xx xx xx »

Les codages sur 2, 3 ou 4 octets permettent de couvrir tous les caractères connus.

Le numéro correspondant à chaque caractère est généralement donné en hexadécimal avec l'écriture « U+xxxx » où « xxxx » est un nombre hexadécimal avec les plages suivantes :

- 1 octet : U+0000 à U+007F
- 2 octets : U+0080 à U+07FF

- 3 octets : U+0800 à U+FFFF
- 4 octets : U+10000 à U+10FFFF

En UTF-8, le caractère mathématique « \oplus » est encodé avec « U+2295 ». En HTML, il est possible d'utiliser l'écriture « ⊕ » pour coder ce caractère dans une page web (à condition d'avoir déclaré le charset UTF-8 comme vu plus haut).

L'intérêt d'utiliser UTF-8 est donc d'assurer une plus grande compatibilité aux documents textes. Il faut toutefois toujours faire attention à l'encodage utilisé par défaut, car un texte encodé en ISO-8859-1 (encore utilisé par certains « anciens » logiciels) ne sera pas affiché correctement en UTF-8.

C. Système d'exploitation c'est quoi ?

1. Introduction aux systèmes d'exploitation

Exprimer un algorithme dans un langage de programmation a pour but de le rendre exécutable par une machine dans un contexte donné. La découverte de l'architecture des machines et de leur système d'exploitation constitue une étape importante. Les circuits électroniques sont au cœur de toutes les machines informatiques. Les réseaux permettent de transmettre l'information entre machines. Les systèmes d'exploitation gèrent et optimisent l'ensemble des fonctions de la machine, de l'exécution des programmes aux entrées-sorties et à la gestion d'énergie. On étudie aussi le rôle des capteurs et actionneurs dans les entrées-sorties clavier, interfaces graphiques et tactiles, dispositifs de mesure physique, commandes de machines, etc.

Activité 1

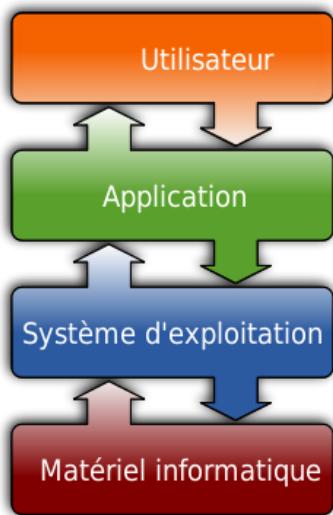
Résumez le contenu de la vidéo ci-dessous en rédigeant, à l'aide d'un traitement de texte, un texte de quelques lignes. Vous pourrez agrémenter votre texte avec un schéma (par exemple à l'aide d'un logiciel de dessin vectoriel comme Inkscape).

<https://youtu.be/4OhUDAtnAUo?si=--acyErE-l2qncGg>



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

Un système d'exploitation (Operating System en anglais) est un ensemble de logiciels qui permettent de faire fonctionner d'autres logiciels en exploitant les ressources proposées par un ordinateur (RAM, CPU, disques...). Les logiciels n'ont pas vraiment à gérer les ressources matérielles, le système d'exploitation s'en charge pour eux.



Il est à la fois :

- une *machine virtuelle* qui présente une interface simplifiée d'accès aux ressources (processeur, mémoire, périphériques d'entrée/sortie, réseau ...) pour les autres programmes et pour l'utilisateur
- un *chef d'orchestre* et un *administrateur* :
- c'est le premier programme exécuté au démarrage de l'ordinateur
- il gère l'accès concurrent aux ressources par les différents programmes (ordonnancement de l'utilisation du processeur par les programmes en cours d'exécution ou processus, sécurisation de la mémoire) ou utilisateurs (droits d'accès du système de fichiers).



3. De Multics à UNIX

Les tout premiers ordinateurs n'avaient pas vraiment de système d'exploitation (les logiciels devaient gérer la partie matériel). Il faut attendre 1965 pour voir arriver le premier système d'exploitation multitâche (capable d'exécuter plusieurs programmes en "même temps") et multi-utilisateur : Multics.

Le système d'exploitation UNIX voit le jour à la toute fin des années 60. Il a été conçu par Ken Thompson et Dennis Ritchie des laboratoires Bell. Le système d'exploitation Multics ne fonctionnait que sur des ordinateurs extrêmement chers, l'idée de Thompson et Ritchie était de concevoir un système d'exploitation pour les

ordinateurs un peu moins onéreux (mais on ne pouvait tout de même pas parler à cette époque d'informatique "grand public", les ordinateurs étaient encore réservés aux centres de recherche et aux grandes entreprises). Comme nous le verrons un peu plus loin, le système UNIX est encore utilisé aujourd'hui.

Activité 2

Résumez le contenu de la vidéo ci-dessous en rédigeant, à l'aide d'un traitement de texte, un texte de quelques lignes.

<https://youtu.be/Za6vGTLp-wg>



4. Systèmes propriétaires vs systèmes libres

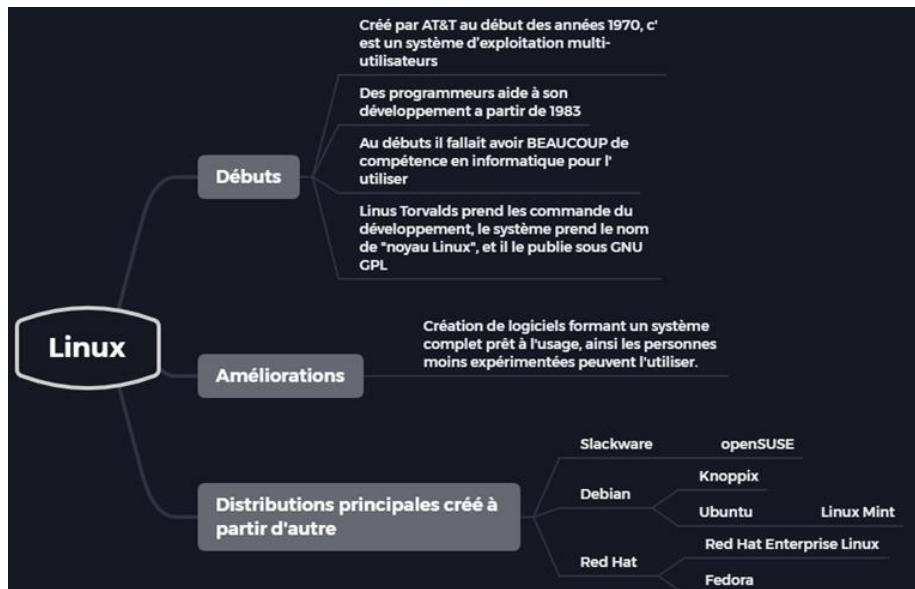
a- Notion de logiciel libre

Le système UNIX est un système dit "**propriétaire**" (certaines personnes disent "**privateur**"), c'est-à-dire un système non libre. Mais plus généralement, qu'est-ce qu'un logiciel libre ?

D'après Wikipédia : "**Un logiciel libre est un logiciel dont l'utilisation, l'étude, la modification et la duplication par autrui en vue de sa diffusion sont permises, techniquement et légalement, ceci afin de garantir certaines libertés induites, dont le contrôle du programme par l'utilisateur et la possibilité de partage entre individus**". Le système UNIX ne respecte pas ces droits (par exemple le code source d'UNIX n'est pas disponible, l'étude d'UNIX est donc impossible), UNIX est donc un système "**propriétaire**" (le contraire de "**libre**"). Attention qui dit logiciel libre ne veut pas forcément dire logiciel gratuit (même si c'est souvent le cas), la confusion entre "**libre**" et "**gratuit**" vient de l'anglais puisque "**free**" veut à la fois dire "**libre**", mais aussi gratuit.

b- Linux

En 1991, un étudiant finlandais, Linus Torvalds, décide de créer un clone libre d'UNIX en ne partant de rien (on dit "**from scratch**" en anglais) puisque le code source d'UNIX n'est pas public. Ce clone d'UNIX va s'appeler Linux (Linus+UNIX). Linux est aujourd'hui le système d'exploitation le plus utilisé au monde si on tient compte des serveurs et des smartphones (Android est dérivé d'un système Linux)



Linux est un **logiciel libre**.

Un logiciel est dit libre si il vérifie les libertés suivantes :

- la liberté de faire fonctionner le logiciel comme désiré pour l'utilisateur,
- la liberté de pouvoir accéder et modifier le code source du logiciel,
- la liberté de pouvoir redistribuer des copies du logiciel,
- la liberté de pouvoir distribuer aux autres des copies de versions modifiées du logiciel.

Un logiciel qui n'est pas libre est dit **propriétaire**.

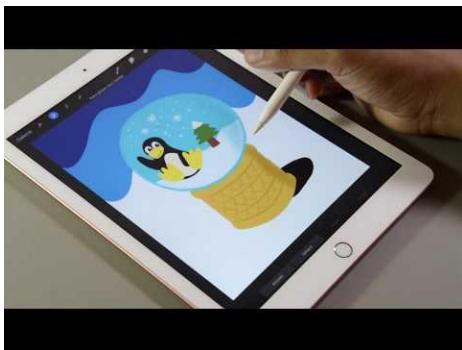


Tux, la mascotte de Linux

Activité 3

Résumez le contenu de la vidéo ci-dessous en rédigeant, à l'aide d'un traitement de texte, un texte de quelques lignes.

https://youtu.be/IquNF_DXcF8



5. Microsoft

Difficile de parler des systèmes d'exploitation sans parler de Microsoft !



Microsoft a été créée par Bill Gates et Paul Allen en 1975. Microsoft est surtout connue pour son système d'exploitation Windows. Windows est un système d'exploitation "propriétaire", la première version de Windows date 1983, mais à cette date Windows n'est qu'un ajout sur un autre système d'exploitation nommé MS-DOS. Aujourd'hui Windows reste le système d'exploitation le plus utilisé au monde sur les ordinateurs grand public (ordinateurs personnels), il faut dire que l'achat de Windows est quasiment imposé lorsque l'on achète un

ordinateur dans le commerce, car oui, quand vous achetez un ordinateur neuf, une partie de la somme que vous versez termine dans les poches de Microsoft. Il est possible de se faire rembourser la licence Windows, mais cette opération est relativement complexe.

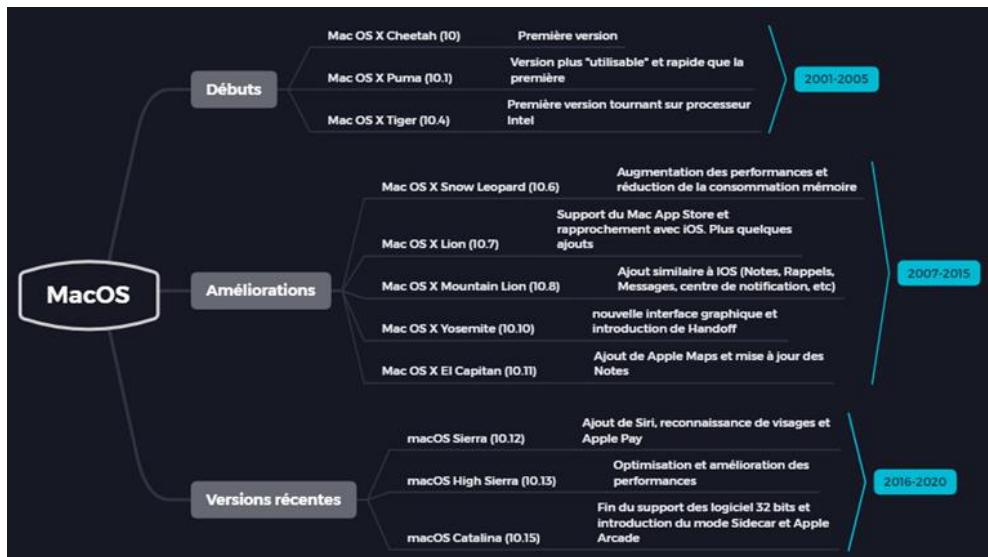


6. Apple

Enfin pour terminer, quelques mots sur le système d'exploitation des ordinateurs de marque Apple : tous les ordinateurs d'Apple sont livrés avec le système d'exploitation macOS. Ce système macOS est un système d'exploitation UNIX, c'est donc un système d'exploitation propriétaire.



Capture d'écran du système macOS



-
- *Windows est principalement utilisé sur les PC de bureau, c'est un système d'exploitation propriétaire tout comme MacOS.*
 - *Linux au contraire est un système d'exploitation libre, il est principalement utilisé dans les serveurs, téléphones portables avec Android ou les supercalculateurs. Son code source est accessible publiquement en ligne.*
-

D. Un langage de programmation c'est quoi ?

1. Qu'est-ce qu'un langage de programmation ?

Le langage machine, c'est-à-dire le code binaire compris par le processeur, est assez peu accessible au commun des mortels. S'il fallait communiquer avec le processeur directement ainsi, seule une élite très restreinte pourrait faire de la programmation, d'autant plus que chaque processeur a son propre langage machine.

Un langage de programmation est un langage qui se place généralement à un niveau plus compréhensible par l'humain lambda, permettant, dans une syntaxe stricte, de décrire un algorithme par la donnée d'une succession d'instructions (des ordres). Cette succession d'instructions sera ensuite traduite en langage machine par un programme spécifique (le compilateur ou l'interpréteur)

Ainsi, un langage de programmation est à voir comme un langage intermédiaire permettant de communiquer indirectement avec le cœur opératoire de l'ordinateur. Nous utiliserons cette année le langage de programmation Python, dans sa troisième version.

2. Niveau de langage

Comme vous le savez, il existe un grand nombre de langages de programmation. Pourquoi une telle variété ? Pourquoi certaines personnes ne jurent-elles que par un langage en particulier ? Y a-t-il réellement des différences fondamentales entre les différents langages ?

La réponse est OUI, de plusieurs points de vue. Nous abordons 3 de ces points de vue dans les 3 paragraphes qui viennent.

Le premier point de vue est celui du niveau (d'abstraction) de langage. Il s'agit essentiellement de savoir à quel point le langage de programmation s'éloigne du langage machine et des contraintes organisationnelles de la

mémoire et des périphériques qui lui sont liées pour s'approcher du langage humain et s'affranchir de la gestion de l'accessoire pour se concentrer sur l'aspect algorithmique :

- **Un langage de bas niveau** est un langage restant proche des contraintes de la machine ; le but est davantage la recherche de l'efficacité par une gestion pensée et raisonnée des ressources (mémoire, périphérique...), au détriment du confort de programmation. Parmi les langages de bas niveau, on peut citer Assembleur ou C.
- **Un langage de haut niveau** est un langage s'approchant davantage du langage humain, et dégageant la programmation de toutes les contraintes matérielles qui sont gérées automatiquement (mémoire, périphériques...). L'intérêt principal est un confort de programmation et la possibilité de se concentrer sur l'aspect algorithmique. En revanche, on y perd nécessairement en efficacité : la gestion automatique des ressources ne permet pas de gérer de façon efficace toutes les situations particulières.

Ainsi, dans la vie pratique, les langages de haut niveau sont largement suffisants en général. C'est le cas en particulier pour la plupart des applications mathématiques, physiques ou techniques, pour lesquels le temps de réponse est suffisamment court, et qui sont amenées à être utilisées à l'unité et non de façon répétée.

Évidemment, dès que l'enjeu de la rapidité intervient (calculs longs, par exemple dans les problèmes de cryptographie, ou encore besoin de réactivité d'un système, donc pour tout ce qui concerne la programmation liée aux systèmes d'exploitations, embarqués ou non), il est préférable d'adopter un langage de plus bas niveau.

Il est fréquent dans des contextes industriels liés à l'efficacité de systèmes embarqués, que le prototypage se fasse sur un langage de haut niveau (travail des algorithmiciens), puis soit traduit en un langage de bas niveau (travail des programmeurs), souvent en C.

Python est un langage de très haut niveau d'abstraction. Sa philosophie est de dégager l'utilisateur de toute contrainte matérielle, et même de se placer à un niveau où la plupart des algorithmes classiques (tri, recherche de motifs, algorithmes de calcul numérique...) sont déjà implémentés. Ainsi, Python propose un certain nombre de modules complémentaires facultatifs, proposant chacun un certain nombre d'outils algorithmiques dans un domaine précis. Python se place donc délibérément à un niveau où une grande partie de la technique est cachée.

Cela fournit un grand confort et une grande facilité de programmation, mais une maîtrise moins parfaite des coûts des algorithmes utilisant des fonctions prédéfinies.

3. Interprétation et compilation

Une autre grande différence entre les langages de programmation est la façon dont ils sont traduits en langage machine. C'est lors de cette traduction qu'est rajoutée toute la gestion de la basse-besogne dont on s'était dispensé pour un programme de haut-niveau. Ainsi, cette traduction est d'autant plus complexe que le langage est de haut-niveau.

Il existe essentiellement deux types de traduction :

- **La compilation.** Elle consiste à traduire entièrement un langage de haut niveau en un langage de bas niveau (souvent en Assembleur), ou directement en langage machine (mais cela crée des problèmes de portabilité d'une machine à une autre). Le programme chargé de faire cette

traduction s'appelle le compilateur, et est fourni avec le langage de programmation. Le compilateur prend en argument le code initial, et retourne en sortie un nouveau fichier (le programme compilé), directement exploitable (ou presque) par l'ordinateur.

- **Avantages** : une fois la compilation effectuée, le traitement par l'ordinateur est plus rapide (il repart de la source compilée). C'est donc intéressant pour un programme finalisé, voué à être utilisé souvent.
 - **Inconvénients** : Lors du développement, la compilation peut être lente, et nécessite que le programme soit syntaxiquement complet : elle nécessite donc de programmer étape entière par étape entière, sans pouvoir faire de vérifications intermédiaires.
- **L'interprétation.** Contrairement à la compilation, il ne s'agit pas de la conversion en un autre programme, mais d'une exécution dynamique. L'exécution est effectuée directement à partir du fichier source : l'interpréteur lit les instructions les unes après les autres, et envoie au fur et à mesure sa traduction au processeur.
- **Avantages** : il est inutile d'avoir un programme complet pour commencer à l'exécuter et voir son comportement. Par ailleurs, les environnements de programmation associés à ces langages permettent souvent la localisation dynamique des erreurs de syntaxes (le script est interprété et validé au moment-même où il est écrit). Enfin, l'exécution dynamique est compatible avec une exécution en console, instruction par instruction. C'est un atout majeur, notamment pour vérifier la syntaxe d'utilisation et le comportement d'une instruction précise.
 - **Inconvénients** : Le programme finalisé est plus lent à l'exécution, puisque la traduction part toujours du code initial, lointain du programme machine.

Python est un langage interprété, semi-compilé. On peut ainsi bénéficier d'une utilisation en console, ainsi que, suivant les environnements, d'une détection dynamique des erreurs de syntaxe. En revanche, l'exécution du programme commence par une compilation partielle, nécessitant une syntaxe complète.

4. Paradigmes de programmation

Un paradigme de programmation est un style de programmation déterminant de quelle manière et sous quelle forme le programmeur manipule des données et donne des instructions. Il s'agit en quelque sorte de la philosophie du langage. Il existe un grand nombre de paradigmes de programmation. Parmi les plus utilisés, citons les suivants :

- **La programmation impérative.** Le programme consiste en une succession d'instructions. L'exécution d'une instruction a pour conséquence une modification de l'état de l'ordinateur. L'état final de l'ordinateur fournit le résultat voulu.
- **La programmation orientée objet.** On agit sur des objets (des structures de données). Les objets réagissent à des messages extérieurs au moyen de méthodes. Ainsi, le gros de la programmation porte sur la définition de méthodes associées à des classes d'objets.

- **La programmation fonctionnelle.** On ne s'autorise pas les changements d'état du système. Ainsi, on ne fait aucune affectation. On n'agit pas sur les variables mais on exprime le programme comme un assemblage de fonctions (au sens mathématique).
- **La programmation logique.** C'est un paradigme basé sur les règles de la logique formelle. Il est notamment utilisé pour les démonstrations automatiques, ou pour l'intelligence artificielle.

Python est un langage hybride, adapté à plusieurs paradigmes. Essentiellement cette année, nous utiliserons Python en tant que langage impératif et orienté objet.

5. Environnement de développement intégré

Un ordinateur est une machine universelle permettant à tout utilisateur d'écrire et d'exécuter des programmes, dans les limites fixées par le système d'exploitation. Cependant, la création de programmes complexes nécessite des outils adaptés. Cette section présente les environnements de développement intégrés (IDE), conçus pour faciliter la conception de programmes en offrant un cadre unique avec divers outils nécessaires pour les programmeurs.

Un IDE permet de :

- Écrire des programmes dans un éditeur adapté au langage
- Exécuter les programmes écrits
- Déboguer les programmes pour corriger les erreurs
- Consulter éventuellement de la documentation

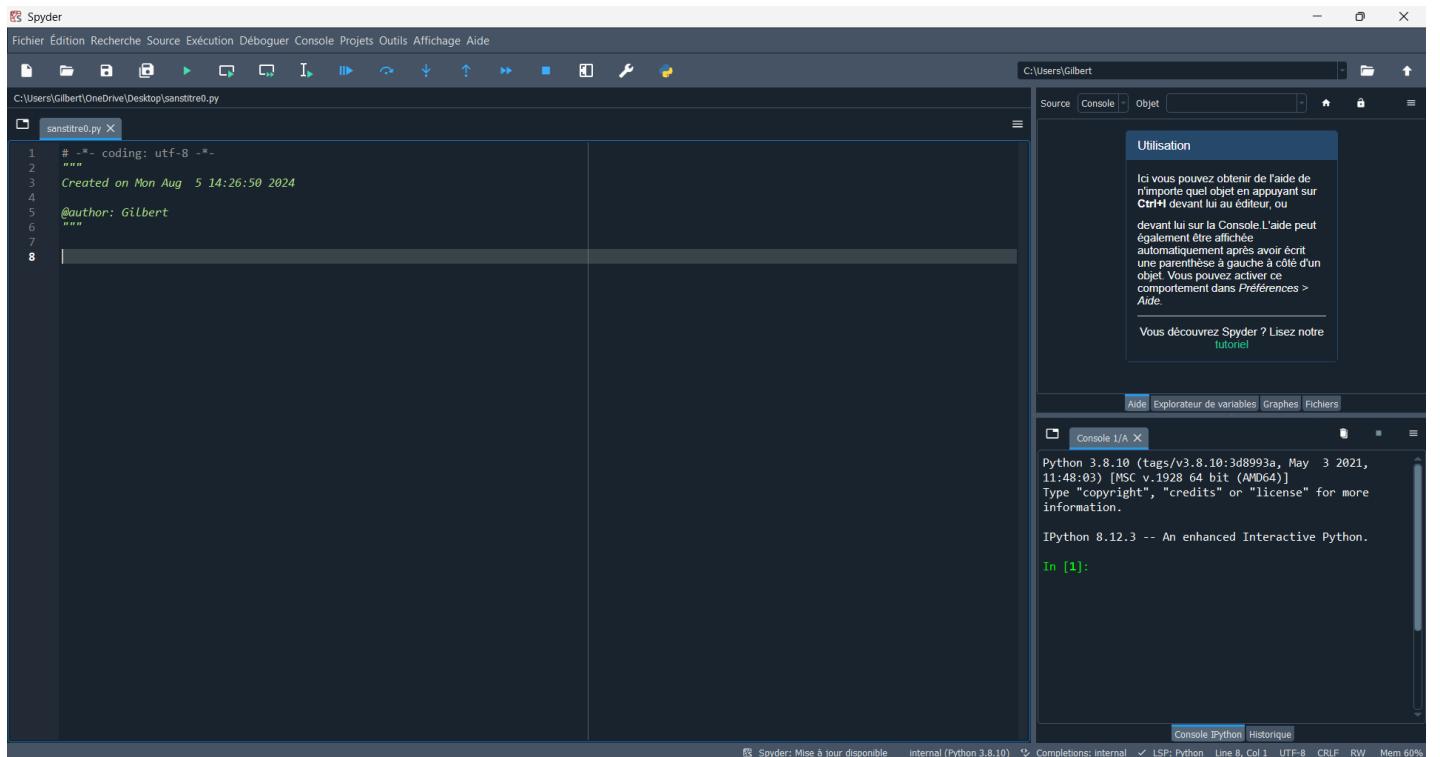
Il existe de nombreux IDE, chacun avec ses spécificités (spécifiques à un langage ou génériques, légers ou complets). Tous fonctionnent de manière similaire, et il est important de comprendre les principes généraux pour choisir un IDE confortable à utiliser. Voici quelques exemples d'IDE gratuits et open source :

- **IDLE** : Fourni avec la distribution standard de Python, il est simple et suffisant pour une utilisation basique de Python.
- **Eclipse** : Plus complexe, il possède des plugins pour presque tous les langages de programmation (dont PyDev pour Python), intéressant pour les étudiants souhaitant utiliser le même IDE dans différents contextes.
- **Emacs ou Vim** : Éditeurs de texte pouvant être étendus pour exécuter des programmes Python ou autres langages. Bien que légers, ils sont dépouillés et peuvent être difficiles à prendre en main.

Nous utiliserons dans ce cours **Spyder**, qui est fourni avec plusieurs distributions de Python : WinPython ou Python(x,y) sous Windows, via le projet MacPorts sous Mac OS, et enfin dans différents paquets pour la plupart des distributions Linux. Nous verrons également d'autres IDE.

Le lecteur n'aura cependant aucun mal à retrouver les mêmes fonctionnalités dans d'autres environnements. L'avantage le plus significatif de Spyder par rapport aux autres distributions est que les bibliothèques utilisées en classes préparatoires sont fournies directement, ce qui en simplifie largement l'installation. La fenêtre de Spyder est divisée en trois parties (figure 1.13) :

- **L'éditeur à gauche**, dans lequel nous écrirons les programmes.
- **L'explorateur en haut à droite**, que nous utiliserons surtout comme débogueur, mais qui peut également servir de documentation.
- **La console interactive en bas à droite**, dans laquelle s'exécuteront les programmes.



6. Consoles interactives

Au démarrage de Spyder, la console interactive affiche des informations sur la version de Python utilisée, ainsi que quelques fonctions d'aide :

```
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.10.2 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.

In [1]:
```

La dernière ligne, qui commence par **In** suivi d'un nombre entre crochets, attend que l'on tape une commande : c'est le mode **interactif** de Python, où chaque ligne tapée est immédiatement exécutée. Ainsi, si l'on tape une expression, sa valeur s'affiche (dans tout cet ouvrage, les entrées tapées par l'utilisateur dans l'interpréteur interactif seront notées en gras) :

```
In [1]: 2+2
Out[1]: 4
```

On appelle **session de travail** une suite d'instructions saisies dans une fenêtre Python interactive avec les réponses correspondantes. Il est possible d'enregistrer le contenu d'une session de travail à l'aide de la commande *Enregistrer l'historique...* accessible par un clic droit dans la console.

Dans ce mode, on peut d'ores et déjà utiliser des variables pour stocker des valeurs. L'affectation s'écrit avec le symbole = et n'affiche aucune valeur ; mais la variable mémorise la valeur qu'on lui a donnée et peut être utilisée dans la suite de la session.

```
In [2]: a = 2
In [3]: a+a
Out[3]: 4
```

Notons que si l'on utilise dans une expression une variable à laquelle on n'a jamais donné de valeur, une erreur se produit.

```
In [4]: b+1
Traceback (most recent call last):

Cell In[4], line 1
  b+1

NameError: name 'b' is not defined
```

La dernière ligne de ce message indique plus précisément d'où vient l'erreur, ici de la variable *b* utilisée à tort. Dans la suite de cet ouvrage, seule cette ligne sera reproduite pour expliquer une erreur.

```
In [5]: b = 1
In [6]: b+1
Out[6]: 2
```

Enfin, il est possible de rappeler une ligne tapée précédemment à l'aide des flèches *haut* et *bas*, et de modifier cette ligne avant de relancer son calcul avec *Entrée*.

```
In [7]: b=b+1
In [8]: b=b+1
In [9]: b=b+1
In [10]: b
Out[10]: 4
```

À chaque nouvelle session, les valeurs des variables sont perdues ; les instructions précédemment saisies peuvent toujours être rappelées, mais cela reste peu pratique et on n'imagine évidemment pas écrire un programme complet de la sorte. Les sessions interactives sont donc à réserver pour tester très rapidement l'évaluation de quelques expressions que l'on ne souhaite pas conserver par la suite.

7. Editeur

Dès que l'on veut écrire un programme, ou même tout simplement une suite d'instructions dont on veut garder une trace, on utilise l'éditeur. Voici un premier programme Python à tester :

```
1 print("Bonjour!")
2
3 x = 42
4
5 print(x)
```

On observe déjà plusieurs différences par rapport au mode interactif :

- Les mots-clés du langage (comme **print**) et les nombres se colorent pour ressortir sur le reste du texte.
- Les chaînes de caractères (entre guillemets) se colorent également.
- Lorsque l'on tape une parenthèse ouvrante, la parenthèse fermante correspondante se crée automatiquement ; et lorsque l'on place le curseur à droite d'une parenthèse, celle-ci se colore en rouge s'il manque la parenthèse correspondante, en vert sinon.

Ainsi, le programme écrit devient plus lisible et on évite de nombreuses fautes de frappe. Cependant, à ce stade, le programme n'est encore qu'un texte, une suite de caractères qui n'a pas de sens pour l'ordinateur. Pour que la machine exécute (on dit aussi *interprète*) les instructions que l'on a tapées, il faut le lui demander par la commande **Exécution** du menu du même nom (raccourci clavier **F5** ou icône triangle vert). Les instructions sont alors lues et exécutées ; le résultat, lorsqu'il y en a un, s'affiche dans l'interpréteur interactif.

On précise qu'il est possible, au moyen de la commande **Configurer...** du menu **Exécution**, de choisir si on utilise une nouvelle fenêtre interactive ou bien celle déjà manipulée. L'intérêt d'une telle option est que, tant qu'une session interactive reste ouverte, elle garde la mémoire des variables auxquelles on a affecté des valeurs. Cela peut influer sur d'autres instructions (interactives ou non) que l'on exécuterait par la suite ; cela permet par ailleurs de consulter la valeur des variables à la fin de l'exécution d'un programme.

SAVOIR-FAIRE Utiliser un environnement de développement intégré

Le minimum pour utiliser un environnement de développement intégré est de savoir :

1 Lancer l'IDE.

2 Ouvrir et enregistrer les programmes que l'on écrit.

3 Exécuter ces programmes.

L'IDE est généralement accessible au moyen d'un raccourci du système d'exploitation, ou en tapant son nom dans un shell texte. La manipulation des programmes se fait par le biais des menus de l'IDE ou par des raccourcis clavier, qui s'avèrent plus efficaces lorsqu'on en a pris l'habitude. Enfin, l'utilisation d'un IDE prend tout son sens lorsque l'on se sert de son débogueur .

8. Débogueur

La fenêtre située en haut à droite de Spyder possède plusieurs onglets :

- L'inspecteur d'objets fournit de l'aide sur un type ou sur une fonction.
- L'explorateur de variables indique la valeur de toutes les variables à tout moment.
- L'explorateur de fichiers parcourt les fichiers Python du système de fichiers.

Celui qui va servir plus particulièrement est l'onglet **Explorateur de variables**. Par exemple, après l'exécution du programme écrit plus haut, on retrouve dans l'explorateur la variable *x* avec une valeur de 42. Si l'interpréteur interactif dans lequel on travaille est le même depuis le début, on y retrouvera également les variables *a* et *b* que l'on a affectées directement dans la console. Enfin, l'explorateur mentionne parfois d'autres valeurs qui ont été initialisées au lancement de l'interpréteur, comme les constantes *e* ou *pi*.

Pour chercher une erreur dans un programme, ou tout simplement pour mieux comprendre son fonctionnement, il peut être utile de l'exécuter pas à pas. Si cela reste faisable de tête pour des programmes simples, il devient vite beaucoup plus commode de confier cette tâche à la machine.

On prendra l'exemple suivant :

```
x = 10  
y = 7  
x = x+y  
y = x  
x = 5 / (x-y)
```

Après avoir recopié ce programme dans l'éditeur, au lieu de l'exécuter normalement, on appelle la commande **Déboguer** du menu **Exécution** (raccourci Ctrl +F5). La première ligne du programme (*x = 10*) est surlignée et recopiée dans l'interpréteur, mais elle n'est pas encore exécutée : on peut le vérifier en constatant que *x* n'a pas la valeur 10 dans l'explorateur de variables. Pour exécuter cette ligne, on peut cliquer sur le bouton **Pas en avant**, ou bien taper *n* (comme *next*) dans l'interpréteur. La ligne est alors exécutée, *x* prend la valeur 10 et c'est la ligne suivante (*y = 7*) qui est surlignée. On peut ensuite continuer à exécuter les lignes les unes après les autres et surveiller l'évolution des différentes variables dans l'explorateur.

Comme il est malcommode de cliquer sur un bouton pour chaque ligne alors que seules certaines posent problème, on peut fixer des **points d'arrêts** dans le programme. Pour cela, on se place sur la ligne à laquelle on désire faire une pause et on choisit Ajouter un **point d'arrêt** dans le menu **Exécution** (raccourci F12). On peut également double-cliquer dans la marge gauche du programme ; le symbole breakpoint.png s'affiche alors en face de cette ligne. Par exemple, dans le programme précédent, il peut être judicieux de placer un point d'arrêt à la dernière ligne, dans laquelle risque de se produire une division par zéro.

Désormais, si au lieu d'effectuer **Pas en avant**, on choisit **Continuer** (c dans l'interpréteur interactif), le programme s'exécute normalement jusqu'au prochain point d'arrêt, puis attend un ordre de l'utilisateur pour continuer à s'exécuter. Dans cet exemple, on voit dans l'explorateur de variables que *x* et *y* contiennent toutes deux la valeur 17 et qu'il y aura donc effectivement une division par zéro.

SAVOIR-FAIRE Utiliser un débogueur Pour corriger une erreur dans un programme :

- on identifie les variables qui ne se comportent pas comme prévu ;
- on localise l'erreur au moyen d'une exécution pas à pas.

Dans le cas d'un programme de taille trop importante pour une exécution pas à pas complète, on veillera à placer des points d'arrêt aux endroits critiques, c'est-à-dire sur les opérations arithmétiques pouvant produire des erreurs, les fins de boucles, certains tests...

Exercice

Cet exercice a pour seul but de s'entraîner à manipuler l'environnement de développement intégré, il n'est pas nécessaire de savoir déjà programmer en Python pour le réaliser.

- 1- Taper le programme suivant dans l'éditeur et l'exécuter. Que se passe-t-il ?

```
i = 10
while i != 0:
    i = 1-i
    print(i)
```

(Il est possible qu'à ce stade il soit nécessaire de fermer complètement l'IDE et de le rouvrir pour continuer...).

- 2- Exécuter ce programme pas à pas et observer les valeurs successives prises par la variable *i*. Expliquer le comportement observé à la première question.
- 3- Placer un point d'arrêt à un endroit approprié du programme pour montrer son comportement sans avoir besoin de détailler des étapes inutiles.

II. Algorithmes et Programmation en Python

II.Un Algorithme

1. Définition

Le mot *Algorithme* vient du nom du mathématicien arabe *Al Khwarizmi*, auteur au IX-e siècle d'un ouvrage faisant la synthèse de la résolution des équations du second degré, suivant le signe des coefficients (afin d'éviter l'usage du signe moins). L'ouvrage en question, proposant des méthodes de résolution par manipulations algébriques (réduction à des formes connues) a donné son nom à *l'algèbre*. Les méthodes exposées peuvent s'apparenter à des algorithmes : on y expose, par disjonction de cas (structure conditionnelle) des façons systématiques de résoudre un certain problème, ne laissant ainsi rien au hasard. Il s'agit bien d'un algorithme de résolution.

Définition (Algorithme)

Un algorithme est une succession d'instructions élémentaires, faciles à faire et non ambiguës, déterminées de façon unique par les données initiales, et fournissant la réponse à un problème posé.

Le développement de l'informatique a marqué l'essor l'algorithme, mais cette discipline n'est pas l'apanage de l'informatique. La notion d'algorithme est liée mathématiquement à la possibilité de résolution systématique d'un problème, donc à la notion de méthode de calcul. On trouve dans le domaine purement mathématique de nombreux algorithmes :

- tous les algorithmes de calcul des opérations élémentaires (addition posée, multiplication posée...)
- l'algorithme de la division euclidienne par différences successives
- l'algorithme d'Euclide du calcul du pgcd
- l'algorithme de résolution des équations de degré 2
- l'algorithme du pivot de Gauss pour résoudre les systèmes d'équations linéaires, et répondre à d'autres questions d'algèbre linéaire.
- l'algorithme de Hörner pour l'évaluation d'un polynôme
- etc.

Les questions qu'on peut se poser sont alors les suivantes :

1. Quelles sont les structures simples élémentaires à partir desquelles sont construits les algorithmes.
2. L'algorithme s'arrête-t-il ? (Problème de la terminaison)
3. L'algorithme renvoie-t-il le résultat attendu ? (Problème de la correction)
4. Combien de temps dure l'exécution de l'algorithme, notamment lorsqu'on le lance sur de grandes données ? (Problème de la complexité).

2. Le Langage

L'étude algorithmique formelle nécessite de se dégager de toute contrainte idiomatique relevant des spécificités de tel ou tel langage. Pour cela, nous utiliserons un pseudo-code, indépendant de toute implémentation. Les traductions ultérieures dans un langage de programmation spécifique se font alors sans difficulté, sachant que certains langages offrent parfois certaines possibilités supplémentaires (qui sont essentiellement du confort, mais n'ajoutent rien à la description formelle des algorithmes, comme par exemple le fait de pouvoir construire une boucle for sur un objet itérable quelconque en Python). Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales (variables d'entrée, préciser leur type)
3. définissant la sortie (variables de résultat, préciser leur type)
4. donnant le bloc d'instructions définissant l'algorithme.

La structure générale est donc la suivante :

Algorithme 1 : Nom ou description de l'algorithme
Entrée : a,b,... : type
Sortie : c,d,... : type
instructions ;
renvoyer (c, d,...)

La dernière ligne, permet de définir le ou les résultats.

3. Les structures élémentaires

Un algorithme repose sur certains types d'instructions élémentaires. Une instruction est une phrase du langage de programmation indiquant à l'ordinateur une ou plusieurs actions à effectuer, induisant un changement d'état de l'ordinateur (c'est-à-dire une modification de la mémoire).

Les différentes structures élémentaires à partir desquelles sont construits les algorithmes en programmation impérative sont, en plus de l'évaluation d'expression, de l'utilisation de fonctions antérieures, et de l'affectation (notée $x \leftarrow$ valeur) :

i. La séquence :

Il s'agit d'un regroupement d'instructions élémentaires, qui seront exécutées successivement. Il s'agit de la notion de *bloc* en informatique, délimité suivant les langages par des balises de début et fin (**begin**, **end**), ou tout simplement par l'indentation, comme en Python.

L'intérêt de la séquence est de pouvoir considérer plusieurs instructions comme une seule, et de pouvoir inclure cette succession dans des structures plus complexes (structures conditionnelles, répétitives...)

Nous écrirons un bloc de la façon suivante :

La plupart du temps, un bloc permettra de délimiter la portée d'une structure composée. Dans ce cas, le mot **bloc** sera remplacé par le mot clé de la structure correspondante, par exemple début **pour** et fin **pour**

Algorithme 2 : Bloc
début bloc
instructions
fin bloc

ii. La structure conditionnelle simple :

C'est la structure permettant d'effectuer des disjonctions de cas. Elle permet de distinguer plusieurs cas si ces différents cas nécessitent des modes opératoires différents. Cela permet également de traiter les cas particuliers.

La structure basique est un branchement à deux issues :

si *condition alors instructions sinon instructions*

La condition peut être n'importe quel booléen, par exemple le résultat d'un test, ou le contenu d'une variable de type booléen. De plus, la clause alternative (**else**) est le plus souvent facultative (algorithmes 3 et 4)

Algorithme 3 : Structure conditionnelle simple sans clause alternative
Entrée : classe : entier
Sortie : Ø
si classe = 4 alors
Afficher('Bestial !')
fin si

Algorithme 4.4 : Structure conditionnelle simple avec clause alternative
Entrée : classe : entier
Sortie : Ø
si classe = 4 alors

```

    | Afficher('Bestial !')
sinon
    | Afficher('Khrass')
fin si
```

Certains langages autorisent le branchement multiple, soit *via* une autre instruction (par exemple case of en Pascal), soit comme surcouche de l'instruction basique. Nous utiliserons cette possibilité en pseudo-code (algorithme 5).

Algorithme 5 : Structure conditionnelle multiple

```

Entrée : classe : entier
Sortie : Ø
si classe = 4 alors
    | Afficher('Bestial !')
sinon si classe = 3 alors
    | Afficher('Skiii !')
sinon
    | Afficher('Khrass')
fin si
```

Évidemment, d'un point de vue de la compléction du langage, cette possibilité n'apporte rien de neuf, puisqu'elle est équivalente à un emboîtement de structures conditionnelles (algorithme 6)

Algorithme 6 : Version équivalente

```

Entrée : classe : entier
Sortie : Ø
si classe = 4 alors
    | Afficher('Bestial !')
sinon
    | si classe = 3 alors
        | Afficher('Skiii !')
    | sinon
        | Afficher('Khrass')
    | fin si
fin si
```

iii. Les boucles

Une boucle est une succession d'instructions, répétée un certain nombre de fois. Le nombre de passages dans la boucle peut être déterminé à l'avance, ou peut dépendre d'une condition vérifiée en cours d'exécution. Cela permet de distinguer plusieurs types de boucles :

01. Boucles conditionnelles, avec condition de continuation.

On passe dans la boucle tant qu'une certaine condition est réalisée. Le test de la condition est réalisé avant le passage dans la boucle. On utilise pour cela une boucle **while** ou **tant que** en français. Voici un exemple :

Algorithme 7 : Que fait cet algorithme ?

```

Entrée : ε (marge d'erreur) : réel
Sortie : Ø
```

```

u ← 1 ;
tant que u > ε faire
|   u ← sin(u)
fin tant que

```

Ici, on calcule les termes d'une suite définie par la récurrence $u_{n+1} = \sin(u_n)$. On peut montrer facilement que cette suite tend vers 0. On répète l'itération de la suite (donc le calcul des termes successifs) tant que les valeurs restent supérieures à ϵ .

Comme dans l'exemple ci-dessus, une boucle **while** s'utilise le plus souvent lorsqu'on ne connaît pas à l'avance le nombre de passages dans la boucle. Souvent d'ailleurs, c'est le nombre de passages dans la boucle qui nous intéresse (afin, dans l'exemple ci-dessus, d'estimer la vitesse de convergence de la suite). Dans ce cas, il faut rajouter un compteur de passages dans la boucle, c'est-à-dire une variable qui s'incrémente à chaque passage dans la boucle : La valeur finale de i nous dit maintenant jusqu'à quel rang de la suite il faut aller pour obtenir la première valeur inférieure à ϵ (et par décroissance, facile à montrer, toutes les suivantes vérifieront la même inégalité)

02. Boucles conditionnelles, avec condition d'arrêt

Il s'agit essentiellement de la même chose, mais exprimé de façon légèrement différente. Ici, on répète la série d'instructions jusqu'à la réalisation d'une certaine condition. Il s'agit de la boucle **repeat... until...**, ou **répéter... jusqu'à ce que...**, en français. L'algorithme précédent peut se réécrire de la façon suivante, de façon quasi-équivalente :

La différence essentielle avec une boucle **while** est que, contrairement à une boucle **while**, on passe nécessairement au moins une fois dans la boucle. À part ce détail, on a équivalence entre

Algorithme 8 : Calcul de i tel que $u_i \leq \epsilon$

```

Entrée : ε (marge d'erreur) : réel
Sortie : i : entier
u ← 1 ;
i ← 0 ;
tant que u > ε faire
|   u ← sin(u) ;
|   i ← i + 1
fin tant que
renvoyer i

```

Algorithme 9 : Vitesse de convergence de u_n

```

Entrée : ε (marge d'erreur) : réel
Sortie : i (rang tel que  $u_i \leq \epsilon$ ) : entier
u ← 0 ;
i ← 0 ;
répéter
|   u ← sin(u) ;
|   i ← i + 1
jusqu'à ce que u ≤ ε ;
renvoyer i ;

```

les deux structures, en remplaçant la condition de continuation par une condition d'arrêt (par négation). Ainsi, une boucle **repeat** instructions **until condition** est équivalente à :

Algorithme 10 : Structure équivalente à repeat... until... : version 1

```
instructions ;
tant que ~ condition faire
|   instructions
fin tant que
```

Remarquez ici le passage forcé une première fois dans la succession d'instructions (bloc isolé avant la structure). On peut éviter la répétition de la succession d'instructions en forçant le premier passage à l'aide d'une variable booléenne :

Algorithme 11 : Structure équivalente à repeat... until... : version 2

```
b ← True ;
tant que (~ condition) ∨ b faire
|   instructions ;
|   b ← False
fin tant que
```

Cela a cependant l'inconvénient d'augmenter le nombre d'affectations. Réciproquement, une boucle **while condition do instructions** est équivalente à la structure de l'algorithme 12.

Algorithme 12 : Structure équivalente à while... do...

```
si condition alors
|   répéter
|   |   instructions
|   jusqu'à ce que ~ condition;
fin si
```

Au vu de ces équivalences, même si en Python, les boucles **repeat** n'existent pas, nous nous autoriseront à décrire les algorithmes en utilisant cette structure, plus naturelle dans certaines situations. Il faut cependant garder dans l'esprit que dans le cadre d'une définition formelle d'un algorithme, cela crée une redondance avec la structure **while**.

03. Boucles inconditionnelles

Il s'agit de boucles dont l'arrêt ne va pas dépendre d'une condition d'arrêt testée à chaque itération. Dans cette structure, on connaît par avance le nombre de passages dans la boucle. Ainsi, on compte le nombre de passages, et on s'arrête au bout du nombre souhaité de passages. Le compteur est donné par une variable incrémentée automatiquement :

Algorithme 13 : Cri de guerre

```

Entrée : Ø
Sortie : cri : chaîne de caractères
cri ← 'besti' ;

pour i ← 1 à 42 faire
|   cri ← cri + 'â'
fin pour
cri ← cri + 'l' ;
renvoyer cri

```

4. Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme. L'intérêt est multiple :

- Éviter d'avoir à écrire plusieurs fois la même séquence d'instructions, si elle est utilisée à différents endroits d'un algorithme. On peut même utiliser une même procédure dans différents algorithmes principaux.
- Une procédure peut dépendre de paramètres. Cela permet d'adapter une séquence donnée à des situations similaires sans être totalement semblables, les différences entre les situations étant traduites par des valeurs différentes de certains paramètres.
- Écrire des procédures permet de sortir la partie purement technique de l'algorithme principal, de sorte à dégager la structure algorithmique de ce dernier de tout encombrement. Cela augmente la lisibilité de l'algorithme.
- Une procédure peut s'appeler elle-même avec des valeurs différentes des paramètres (récursivité). Cela permet de traduire au plus près certaines définitions mathématiques par récurrence. Cela a un côté pratique, mais assez dangereux du point de vue de la complexité si on n'est pas conscient précisément de ce qu'implique ce qu'on écrit.

La procédure ?? est un exemple typique de procédure : il s'agit de l'affichage d'un polynôme entré sous forme d'un tableau. Créer une procédure pour cela permet de pouvoir facilement afficher des polynômes à plusieurs reprises lors d'un algorithme portant sur les polynômes, ceci sans avoir à se préoccuper, ni s'encombrer de la technique se cachant derrière. Dans cette procédure **str** est une fonction convertissant une valeur numérique en chaîne de caractères.

Procédure 14 : affichepolynome(T)

```

Entrée : T : tableau représentant un polynôme
Sortie : Ø
ch ← ” ;
pour i ← Taille (T)-1 descendant à 0 faire
    si T [i] ≠ 0 alors
        si (T [i] > 0) alors
            si ch ≠ ” alors
                | ch ← ch + ‘+’
            fin si
        sinon
            ch ← ch + ‘-’
        fin si
        si (|T [i]| ≠ 1) ∨ (i = 0) alors
            ch ← ch + str(|T [i]|);
            si i ≠ 0 alors
                | ch ← ch + ‘*’
            fin si
        fin si
        si i ≠ 0 alors
            ch ← ch + ‘x’
        fin si
        si i > 1 alors
            ch ← ch + ‘^’ + str(i)
        fin si
    fin si
fin pour
si ch = ” alors
    | ch ← ‘0’
fin si
Afficher (ch)

```

Une fonction est similaire à une procédure, mais renvoie en plus une valeur de sortie. En général, il est conseillé de faire en sorte que la seule action d'une fonction soit ce retour d'une valeur. En particulier, on ne fait aucune interface avec l'utilisateur : pas d'affichage, ni de lecture de valeur (les valeurs à utiliser pour la fonction étant alors passées en paramètres).

Fonction 15 : compteA(ch)

```

Entrée : ch : chaîne de caractères
Sortie : n : entier
n ← 0 ;
pour i ← 0 à Taille(ch) faire
    | si ch[i] = ‘a’ alors
    |     | n ← n + 1
    | fin si
fin pour
renvoyer n

```

Fonction 16 : un(n)

```

Entrée : n : entier positif
Sortie : un(n) : réel
si n = 0 alors
    | renvoyer 1
sinon

```

```

    |   renvoyer sin(un(n - 1))
fin si

```

Fonction 17 : fibo(n)
Entrée : n : entier positif
Sortie : $fibo(n)$: réel
si $n = 0$ alors
renvoyer 0
sinon si $n = 1$ alors
renvoyer 1
sinon
renvoyer $fibo(n - 1) + fibo(n - 2)$
fin si

La fonction 15 est une fonction simple déterminant le nombre de lettres a dans une chaîne de caractères : La fonction 16 est une fonction récursive pour le calcul la suite définie par une récurrence simple, en l'occurrence $u_{n+1} = \sin(u_n)$, initialisée par 1.

La fonction 17 est une très mauvaise façon de calculer le n -ième terme de la suite de **Fibonacci** par récursivité. En pratique, vous calculerez plus vite à la main F100 que l'ordinateur par la fonction ci-dessus. Pourquoi cet algorithme récursif est-il si mauvais ?

L'étude de la récursivité est au programme de Spé (ou de Sup en cours d'option). Nous nous contenterons donc cette année d'une utilisation naïve de la récursivité, tout en étant conscient des dangers d'explosion de complexité que cela peut amener.

II. Deux Programmation en Python

Python est le langage de programmation au programme des CPGE scientifiques. Ce langage a été développé par Guido Von Russom à la fin des années 80 et au début des années 90. Celui-ci a nommé le langage en référence à la troupe d'humoristes britanniques des Monthy Python.

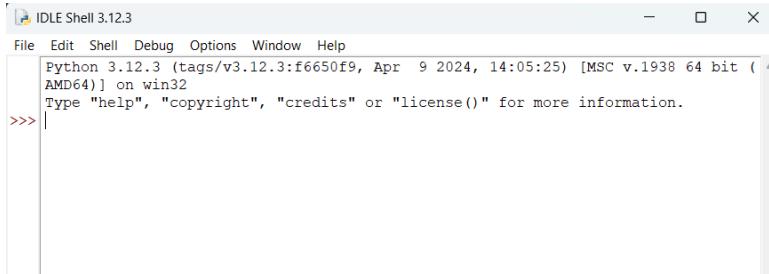
Python est un langage hybride adapté à la programmation impérative et objet, permettant de créer des classes et des méthodes. C'est un langage de haut niveau qui gère le matériel, laissant au programmeur le soin des aspects algorithmiques, avec des fonctions et modules spécialisés pour des tâches complexes.

Python est semi-compilé, offrant flexibilité et souplesse via une compilation à l'exécution, mais au prix d'une exécution plus lente. Il est souvent utilisé pour le prototypage avant une traduction en C pour plus d'efficacité. La syntaxe de Python repose sur l'indentation, garantissant une clarté du code, et dispose d'une fonction d'aide intégrée pour explorer les objets, modules et fonctions.

En revenant à la page principale du site sus-cité <https://www.python.org/>, vous trouverez également différentes distributions de Python à télécharger et installer sur vos ordinateurs personnels (plus que conseillé !), notamment pour Windows ou MacOS. Pour une distibution Linux, renseignez-vous sur les forums ; il y a moyen de l'installer avec les moteurs d'installation (yum install pour Fedora par exemple). Sur certains

systèmes, Python est installé par défaut (vérifier dans ce cas la version). Il existe des différences notables entre les versions 2 et 3, les rendant incompatibles. Nous utiliserons la version 3 en TP. Choisissez donc la version 3 la plus récente.

Il peut également être utilisé de télécharger un environnement de programmation (essentiellement un programme incluant dans un même environnement graphique un éditeur de code et un interpréteur permettant de taper des instructions en ligne, et de visualiser les résultats de vos programmes). Certains environnements sont généralistes (prévus pour la programmation dans plusieurs langages différents), comme Eclipse ; d'autres sont plus spécifiques à Python, comme **IDLE**, conçu par le concepteur de Python, ou Pyzo, que nous utiliserons en TP. Ce dernier est donc à privilégier sur vos ordinateurs personnels, afin de vous habituer. IDLE est plus minimaliste, pour ceux qui préfère la sobriété. Certains environnements viennent avec une distribution précise de Python (qu'il n'est dans ce cas pas nécessaire d'installer avant).

A screenshot of the IDLE Shell 3.12.3 window. The title bar says "IDLE Shell 3.12.3". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the Python interpreter prompt ">>> |". Above the prompt, there is a status message: "Python 3.12.3 (tags/v3.12.3:f6650f9, Apr 9 2024, 14:05:25) [MSC v.1938 64 bit (AMD64)] on win32 Type "help", "copyright", "credits" or "license()" for more information.".

On peut aussi utiliser un éditeur de texte standard, comme **emacs**, puis lancer Python sur le fichier en ligne de commande. La plupart des éditeurs reconnaissent le langage Python et proposent une coloration syntaxique, mais ils ne disposent pas en revanche des aides syntaxiques lorsqu'on tape le nom d'une fonction.

Une fonction essentielle, et inclassable, est la fonction d'aide, que vous pouvez appliquer à la plupart des noms d'objets, de modules, de fonctions :

```
help() # ouverture de la page d'aide de l'objet spécifié
```

Par exemple **help(print)** renvoie la page d'aide sur la fonction **print()** réalisant l'affichage d'une chaîne de caractères.

Pour plus d'information <https://docs.python.org/3/tutorial/index.html>

A. Types simples et expressions

1. Expressions

Une expression est une suite de caractères définissant une valeur. Pour connaître cette valeur, la machine doit évaluer l'expression. Voici quelques exemples numériques :

```
>>> 1+4
5
>>> 2.1+7
9.1
>>> 5/2
2.5
>>> 5//2*4.5
9.0
```

Les valeurs possèdent ce qu'on appelle un *type* : par exemple *entier*, *flottant*, *booléen*, *chaîne de caractères*, *liste*, *fonction*... Le type détermine les propriétés formelles de la valeur (par exemple, les opérations qu'elle peut subir) et matérielles (par exemple, la façon dont elle est représentée en mémoire et la place qu'elle occupe).

Pour connaître le type d'une expression après évaluation, il suffit de le demander à Python à l'aide de **type** :

```
>>> type(1+4)
<class 'int'>
>>> type(2.1+7)
<class 'float'>
>>> type(5/2)
<class 'float'>
>>> type(4<7)
<class 'bool'>
>>> type("blabla")
<class 'str'>
>>> type([0,1,2])
<class 'list'>
>>> type(lambda x:x+1)
<class 'function'>
```

On retrouve ici des types simples : entier (**int**), flottant (**float**) et booléen (**bool**). Chacun de ces types va faire l'objet d'un traitement particulier dans ce qui suit, de même que les types plus compliqués (chaînes de caractères, listes, fonctions...) un peu plus tard. On ne se préoccupera pas du mot clef `class` qui fait référence au caractère **orienté objet** du langage Python.

Comme dans la plupart des langages de programmation, une expression en Python est soit :

- une constante comme 2 ou 3.5 ;
- un nom de variable comme **x**, **i**, ou **compteur** ;
- le résultat d'une fonction appliquée à une ou plusieurs expressions, comme PGCD(5,9) ;
- la composée de plusieurs expressions réunies à l'aide d'opérateurs, comme **not a**, **3**6**, **(6+7)*8**. Les parenthèses servent comme en mathématiques à préciser quels opérateurs doivent être évalués en premier.

Les principaux types (hors modules complémentaires) :

```
int      # Entiers, de longueur non bornée
float    # Flottants (réels, en norme IEEE 754 sur 64 bits, voir chapitre 1)
complex  # Nombres complexes
bool     # Booléens (True / False)
list     # Listes
set      # Ensembles
tuple    # $n$-uplets
str      # Chaînes de caractères (string)
function # Fonctions
```

Voyons maintenant les types simples en détail.

2. Entiers

Constantes. Il n'y a pas grand-chose à dire sur les entiers. On soulignera simplement qu'en Python, les entiers sont non bornés et permettent donc de faire des calculs exacts, avec des entiers gigantesques.

```
>>> 5**76 # ** est l'exponentiation.  
132348898008484427979425390731194056570529937744140625
```

Opérateurs. Les opérateurs sur les entiers sont précisés dans la liste ci-dessous :

opérateur	+	-	*	//	%	**
signification	Addition	soustraction	multiplication	division entière	modulo	exponentiation

On notera bien que // produit une division entière (quotient dans la division euclidienne). On ne peut pas évaluer $a//b$ si b est nul, et on fera attention si b est négatif ; en effet le comportement est un peu différent de la définition vue en cours de mathématiques.

Règles de priorités. Certains opérateurs sont évalués avant les autres, dans l'ordre de priorité suivant :

1. Exponentiation.
2. Modulo.
3. Multiplication et division entières.
4. Addition et soustraction.

Sur les opérateurs de même priorité, c'est celui qui est le plus à gauche qui est évalué en premier. Les parenthèses permettent de changer ces priorités.

```
>>> 2+25%3*2**4  
18  
>>> (2+25%(3*2))**4  
81
```

3. Flottants

Constantes. Les flottants sont représentés en mémoire sur 32 ou 64 bits suivant le système (plutôt 64 de nos jours). Sur 64 bits, on a 1 bit de signe, 11 bits d'exposant et 52 bits de mantisse (voir le cours sur la représentation des nombres). On tiendra compte du fait que seul un nombre fini de réels sont représentables en mémoire, ce qui ne permet pas de faire des calculs exacts. En particulier, le plus petit nombre strictement positif représentable exactement en flottant sur 64 bits est 2^{-1074} et le plus grand est légèrement inférieur à 2^{1024} .

Opérateurs. Les opérateurs sur les flottants sont précisés dans la liste ci-dessous (on s'en servira rarement, mais on peut également utiliser le modulo...) :

opérateur	+	-	*	/	**
signification	Addition	soustraction	multiplication	Division	Exponentiation

Règles de priorités. De même que sur les entiers, certains opérateurs sont évalués avant les autres, dans l'ordre de priorité suivant :

1. Exponentiation.
2. Multiplication et division.
3. Addition et soustraction.

Comme pour les entiers, les opérateurs de même priorité sont évalués de gauche à droite, et les parenthèses permettent de changer ces priorités.

Conversion automatique. On remarque que la plupart des opérateurs sur les entiers et flottants sont les mêmes.

Lorsque l'on utilise l'un de ces opérateurs avec des entiers et des flottants, les entiers sont automatiquement convertis en flottants (on pourrait forcer la conversion de l'entier n en flottant avec **float(n)**). C'est le cas également pour la division flottante utilisée avec des entiers.

```
>>> 4*3.1
12.4
>>> 3/4
0.75
```

4. Booléens

Constantes. Les booléens sont essentiels en informatique. Ce type comprend uniquement deux constantes : **True** et **False** (Vrai et Faux). Ils sont principalement utilisés dans les structures de contrôle.

Opérateurs. Les opérateurs sur les booléens sont au nombre de trois. L'un (**not**) est un opérateur unaire (ne prenant qu'un opérande), les deux autres (**and** et **or**) sont des opérateurs binaires (nécessitant deux opérandes). La liste suivante présente les différents opérateurs booléens et leurs *tables de vérité*.

a	b	not a	a or b	a and b
False	False	True	False	False
False	True	True	True	False
True	False	False	True	False
True	True	False	True	True

Ce tableau est intuitif : `not` correspond à la négation. Pour que `a and b` soit vrai, il faut que `a` et `b` le soient tous les deux. Pour que `a or b` soit vrai, il suffit que l'un des deux le soit. Attention : le « ou » français peut parfois avoir le sens d'un ou exclusif, comme dans « fromage ou dessert ». Le `or` en informatique est toujours inclusif (si `a` et `b` sont vrais, alors `a or b` aussi).

Règles de priorité. L'ordre de priorité d'évaluation pour les opérations booléennes est le suivant :

1. `not`.
2. `and`.
3. `or`.

De même que pour les entiers et les flottants, on évalue ensuite de gauche à droite les opérateurs de même priorité, et on peut user de parenthèses.

```
>>> False and False or True
True
>>> False and (False or True)
False
```

Opérateurs de comparaisons et booléens. On utilise rarement des booléens tels quels. Leur intérêt réside dans les structures de contrôle conditionnelles que l'on verra en section 1.3. Ces structures font beaucoup usage de l'évaluation d'expressions produisant des booléens, parmi lesquelles on trouve les opérations de comparaisons sur les entiers/flottants :

Opérateur	<code>==</code>	<code>!=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
signification	Egal	Non égal	Inférieur	Inférieur ou égal	Supérieur	Supérieur ou égal

```
>>> 4>=3 or 5==0
True
```

Pour les évaluations des opérateurs binaires, les deux opérandes des opérateurs de comparaisons sont amenés à un type commun avant l'évaluation de la comparaison (flottant dans le cas d'entier et flottant).

La priorité des opérateurs de comparaison est inférieure à celle des opérateurs arithmétiques : ainsi, l'expression `a==b+c` signifie `a==(b+c)`, ce qui est assez logique.

On peut également, comme sur beaucoup d'objets Python, utiliser les opérateurs `==` et `!=` pour l'égalité et la différence de booléens. Notons que si `x` est un booléen, il est parfaitement inutile d'écrire quelque chose comme `x==True` : ce booléen est égal à `x`. De même, on préférera écrire `not x` que `x==False`.

Caractères paresseux des opérateurs `and` et `or`. On notera que dans une expression de la forme `a and b`, où `a` et `b` sont des expressions, si `a` s'évalue en `False`, on n'a pas besoin d'évaluer `b` pour s'apercevoir que `a and b` s'évalue en `False`. De même avec `a or b` si `a` s'évalue en `True`. Python respecte cette logique : si la partie

gauche suffit à déterminer si l'expression s'évalue en **True** ou **False**, il n'évalue pas la partie droite (on parle du caractère paresseux des opérateurs).

C'est particulièrement utile lors de l'évaluation d'une expression dont la seconde partie pourrait produire une erreur, mais dont la première sert de garde-fou : **x>0 and log(x)>2** ne produit pas d'erreur, même si **x** est un nombre négatif. En effet, dans ce cas **x>0** s'évalue en False et on n'a pas besoin d'évaluer **log(x)>2** qui produirait une erreur, le **log** n'étant pas défini sur les nombres négatifs.

Raccourcis. Plutôt que d'écrire **a<=b and b<=c**, Python comprend très bien **a<=b<=c**. On n'abusera cependant pas de ces raccourcis, pour écrire des choses illisibles comme **a=c**.

B. Variables

1. Identificateurs

Un identificateur est une suite de lettres et chiffres, qui commence par une lettre, et qui n'est pas un mot réservé du langage. Les mots réservés du langage Python sont par exemple **if**, **else**, **def**, **return**, **True**... Le caractère _ (underscore, ou « tiret du 8 ») est considéré comme une lettre. Ainsi, **i**, **j**, **x**, **x2**, compteur et **taille_de_la_liste** sont des identificateurs corrects, contrairement à **4a**, **x{}**, if ou encore taille de la liste. Les majuscules et minuscules ne sont pas équivalents : **x** et **X** sont des identificateurs distincts. Même si les accents sont autorisés, on veillera à ne pas en mettre dans les identificateurs pour ne pas faire dépendre la bonne exécution d'un programme de l'encodage des caractères.

2. Variables

Une variable est constituée de l'association d'un identificateur à une valeur. Cette association est créée lors de l'affectation, qui s'écrit sous la forme **variable = expression**. Attention : il ne faut pas confondre **=** (affectation) avec **==** (test d'égalité). Le mécanisme de l'affectation est le suivant : l'expression à droite du signe égal est évaluée, puis le résultat de l'évaluation est affecté à la variable. Cela n'a donc rien à voir avec le signe **=** des mathématiques.

À la suite d'une telle affectation, chaque apparition de la variable ailleurs que dans la partie gauche d'une autre affectation représente la valeur en question. Cette association entre la variable et la valeur est valable tant qu'il n'y a pas de nouvelle affectation avec cette même variable.

```
>>> x=2+2          #on évalue 2+2, on obtient 4, qu'on affecte à x.  
>>> y=x**x+1      #ici, x représente la valeur 4. 4**4+1 vaut 257, qu'on affecte à y.  
>>> print(y-1)    #print est une fonction d'affichage. y-1 s'évalue en 256, qu'on affiche.  
256  
>>> x=y/2          #nouvelle affectation de x.  
>>> print(x)  
128.5
```

Comme on le voit sur ces exemples, en Python :

- contrairement à plusieurs autres langages de programmation, les variables n'ont pas besoin d'être déclarées (c'est-à-dire préalablement annoncées), la première affectation leur tient lieu de déclaration ;
- les variables ne sont pas liées à un type (mais les valeurs auxquelles elles sont associées le sont forcément) : la même variable x a été associée à des valeurs de types différents (un **int**, puis un **float**).

Dans la suite, on confondra allègrement l'identificateur, la variable (l'association de l'identificateur à une valeur) et la valeur elle-même. Si un identificateur n'a pas été affecté (en toute rigueur il n'est donc pas un nom de variable) son emploi ailleurs que dans le membre gauche d'une affectation est illégale et provoque une erreur. Par exemple :

```
>>> print(variable_inconnue)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'variable_inconnue' is not defined
```

Seul un identificateur correct peut figurer dans le membre gauche d'une affectation. Une syntaxe de la forme **x+1=y** n'a aucun sens :

```
>>> y=5
>>> x+1=y
File "<stdin>", line 1
SyntaxError: can't assign to operator
```

L'erreur est explicite : on ne peut pas affecter à l'opérateur +. Un mécanisme qui sert souvent en informatique est **l'incrémantion** d'une variable : on lui rajoute un certain nombre, souvent 1.

```
>>> x=4
>>> x=x+1                                # le signe = n'a rien à voir avec celui des mathématiques !
>>> print(x)
5
```

Il y a un raccourci en Python pour cette opération : **x+=1**. On peut de même écrire **x-=2** ou encore **x/=3**.

Cette Raccourci est appelé **syntaxe Pythonesque**

Enfin, lors de l'affectation **x=y**, la variable x prend la valeur de celle de y (l'évaluation de l'expression y donne simplement la valeur stockée dans la variable y), mais x et y ne sont pas « liées » pour autant :

```
>>> y=2
>>> x=y
>>> y=5
>>> print(x)
2
```

Revenons un peu sur les calculs effectuer avec les Flottants

```
divmod(x)          # renvoie le couple (x // y, x % y)
                  # permet de faire les deux calculs en utilisant une seule
                  # fois l'algorithme de la division euclidienne.
x ** y           # x puissance y
abs(x)           # valeur absolue
int(x)           # partie entière de l'écriture décimale (ne correspond pas
```

```

# à la partie entière mathématique si x < 0:
# par exemple int(-2.2)=-2.
x.conjugate()      # retourne le conjugué du nombre complexe x
x.real              # partie réelle
x.imag              # partie imaginaire

```

Sur les Booléens

```

Opérations sur les booléens:
x.__and__(y)    ou x & y    ou x and y          # et
x.__or__(y)     ou x | y    ou x or y           # ou
x.__xor__(y)   ou x ^ y    ou x ^ y             # ou exclusif
                                         not x        # négation de x

Tests:
x == y           # égalité (la double égalité permet de distinguer
                  # syntaxiquement de l'affectation)
x < y           # infériorité stricte
x > y           # supériorité stricte
x <= y          # infériorité large
x >= y          # supériorité large
x != y           # différent (non égalité)
x in y           # appartenance (pour les listes, ensembles, chaînes de caractères)
x is y           # identité (comparaison des identifiants de x et y)

```

3. Input

A part ‘**print**’ et ‘**type**’ qui sont des **Fonctions Intégrées en Python** nous avons également ‘**input**’

La fonction **input()** permet de lire des données entrées par l'utilisateur. Elle renvoie toujours une chaîne de caractères.

```
input('texte')
```

Exemple

```

# Lire le nom de l'utilisateur

nom = input("Quel est votre nom ? ")
print("Bonjour, " + nom + "!")

```

input permet de récupérer quelque chose depuis l'entrée standard, qui est par défaut, le clavier :

```

>>> help(input)
Help on built-in function input in module builtins:

input(...)
    input([prompt]) -> string
    Read a string from standard input. The trailing newline is stripped.
    If the user hits EOF (Unix: Ctl-D, Windows: Ctl-Z+Return), raise EOFError.
    On Unix, GNU readline is used if enabled. The prompt string, if given,
    is printed without a trailing newline before reading.

```

Cette fonction prend en paramètre optionnel une chaîne de caractères (prompt string dans la documentation), l'affiche avant de lire une chaîne de caractères depuis l'entrée standard (standard input, par défaut le clavier). En général, on affecte cette chaîne lue à une variable. Dans l'exemple ci-dessous, **une chaîne** a été entrée avec mes petites mains au clavier. La saisie s'arrête lorsqu'on appuie sur la touche Entrée.

```
>>> s=input("Entrez-moi quelque chose : ")
Entrez-moi quelque chose : une chaîne
>>> print(s)
une chaîne
```

Les fonctions de conversions de type permettent de convertir une chaîne de caractères en le type qui nous intéresse. Une conversion de type se fait sous la forme **t(x)** où t est le type voulu et x l'objet à convertir. Lorsqu'on utilise **input** ou lorsqu'on lit des informations dans un fichier, on convertit souvent des chaînes de caractères en d'autres types.

```
>>> n=int(input("Entrez-moi un entier : "))
Entrez-moi un entier : 42
>>> (n+12)//5
10
```

c. Écrire des Programmes Python dans des Fichiers : Utilisation de IDLE

IDLE est un environnement de développement intégré pour Python. Il permet de créer, exécuter et déboguer des programmes Python facilement. Voici comment utiliser IDLE pour travailler avec des fichiers Python.

(1) Création d'un Fichier Python

- i. **Ouvrir IDLE** : Lancez IDLE depuis votre menu de démarrage ou depuis un raccourci.
- ii. **Créer un Nouveau Fichier** : Cliquez sur **File** dans le menu, puis sélectionnez **New File**. Une nouvelle fenêtre s'ouvrira pour écrire votre code.
- iii. **Écrire du Code** : Tapez votre code Python dans cette nouvelle fenêtre. Par exemple, un programme simple utilisant **input**, **print**, et **type** pourrait ressembler à ceci :

```
# Exemple de programme Python
nom = input("Quel est votre nom ? ")
age = int(input("Quel est votre âge ? "))

print("Bonjour, " + nom + "!")
print("Vous avez", age, "ans.")
print("Le type de la variable 'age' est", type(age))
```

- iv. **Sauvegarder le Fichier** :

Cliquez sur **File** puis sur **Save As....**. Donnez un nom à votre fichier avec l'extension **.py** (par exemple, **mon_programme.py**), puis cliquez sur **Save**.

(2) Exécution du Programme

Exécuter le Code :

Cliquez sur **Run** dans le menu, puis sélectionnez **Run Module** ou appuyez sur **F5**. Votre programme s'exécutera et vous pourrez voir les résultats dans la fenêtre de console d'IDLE.

D. Débogage du Code avec IDLE

Déboguer un programme consiste à identifier et corriger les erreurs ou les comportements inattendus dans le code. Voici comment vous pouvez utiliser IDLE pour déboguer votre code Python :

i. Vérifier les Messages d'Erreur

Lorsque vous exécutez un programme dans IDLE, la fenêtre de console affiche les messages d'erreur si le code rencontre un problème. Ces messages sont cruciaux pour localiser et comprendre les erreurs. Voici comment les utiliser efficacement :

- **Messages d'Erreur :**

- Lorsqu'une erreur se produit, IDLE affiche un message d'erreur dans la console. Ce message inclut souvent une description de l'erreur, ainsi qu'un numéro de ligne où l'erreur a été détectée.
- Exemple d'erreur

```
Traceback (most recent call last):
  File "mon_programme.py", line 3, in <module>
    print("Bonjour, " + nom)
TypeError: can only concatenate str (not "int") to str
```

Dans cet exemple, l'erreur indique que vous essayez de concaténer une chaîne de caractères avec un entier, ce qui n'est pas permis en Python.

- **Localiser le Problème :** Le message d'erreur indique le fichier et la ligne où l'erreur s'est produite. Utilisez ces informations pour naviguer rapidement à l'endroit précis du problème dans votre code.

ii. Utiliser les Instructions `print()` pour Déboguer

Les instructions `print()` sont un outil simple mais efficace pour déboguer votre code. Elles vous permettent de visualiser les valeurs des variables et l'état du programme à différents moments de l'exécution.

Afficher des Valeurs Intermédiaires :

Insérez des `print()` à des endroits stratégiques pour afficher les valeurs des variables et suivre l'exécution du programme.

Exemple

```

x = 10
y = 20
print("Avant l'ajout : x =", x, "y =", y)
z = x + y
print("Après l'ajout : z =", z)

```

iii. Trouver des Erreurs Logiques

Utilisez **print()** pour vérifier si les variables contiennent les valeurs attendues et pour comprendre la logique de votre programme.

Exemple

```

def somme(a, b):
    print("Valeurs de a et b :", a, b)
    return a + b

result = somme(5, 3)
print("Résultat de la somme :", result)

```

iv. Corriger les Erreurs

Une fois que vous avez identifié les erreurs à l'aide des messages d'erreur et des instructions **print()**, il est temps de les corriger :

- Retourner à la Fenêtre d'Édition :

- Dans IDLE, passez à la fenêtre d'édition où vous avez écrit votre code. Les erreurs identifiées dans la console doivent être corrigées dans cette fenêtre.

- Apporter les Corrections Nécessaires :

- Modifiez le code en fonction des informations obtenues. Par exemple, si vous avez une erreur de type, assurez-vous que les types de données sont correctement gérés.
- Exemple de Correction

```

nom = input("Quel est votre nom ? ")
# Assurez-vous que toutes les variables sont du même type avant de les concaténer
print("Bonjour, " + nom + "!")

```

v. Sauvegarder et Réexécuter le Programme

- Sauvegardez les modifications apportées à votre fichier Python (en cliquant sur **File** puis **Save** ou en utilisant le raccourci **Ctrl+S**).
- Exécutez à nouveau le programme en cliquant sur **Run** puis **Run Module** ou en appuyant sur **F5** pour vérifier si les erreurs ont été corrigées

E. Les structures conditionnelles

La structure générale d'une instruction conditionnelle est la suivante (rappelez-vous, en Python, l'indentation est primordiale).

```
if expression:  
    [instructions effectuées si expression s'évalue en True]  
elif autre_expression:  
    [instructions effectuées si expression s'évalue en False et autre_expression en True]  
else:  
    [instructions effectuées si expression et autre_expression s'évaluent en False]
```

Les expressions utilisées sont des expressions booléennes : leur évaluation doit produire un booléen : **True** ou **False**.

De telles expressions sont par exemple **a>=4**, ou bien **not a==0 and b>=2** (**a** est non nul et **b** est supérieur ou égal à 2).

Dans une telle structure conditionnelle, les expressions booléennes sont évaluées les unes après les autres, de haut en bas, jusqu'à ce que l'une d'entre elles s'évalue en **True**. Le bloc d'instructions correspondant (et seulement celui-ci) est alors exécuté, puis on sort de la structure conditionnelle. Le bloc correspondant au **else** est exécuté seulement si toutes les expressions conditionnelles situées au-dessus se sont évaluées en **False**. Il est possible de mettre plusieurs **elif** :

Voici un exemple, avec note une variable supposée contenir un flottant entre 0 et 20.

```
if note>=16:  
    print("Mention Très bien")  
elif note>=14:  
    print("Mention Bien")  
elif note>=12:  
    print("Mention Assez bien")  
elif note>=10:  
    print("Mention Passable")  
else:  
    print("Raté !")
```

Même si note contient un flottant supérieur à 16, on n'affichera à l'écran (effet de la fonction **print**) qu'une seule ligne : la première telle que la condition **note>=...** soit réalisée, ou « Raté ! » si note est strictement inférieure à 10.

elif et **else** peuvent tous deux être omis. Dans une telle suite d'instructions au plus une (et exactement une si **else** est présent) est exécutée : la première telle que l'expression booléenne associée s'évalue en **True**. Par exemple, dans la séquence suivante, **x** est incrémenté de 1 s'il est supérieur ou égal à 2, et divisé par 2 s'il est strictement inférieur à 0.

```
if x<0:  
    x=x/2  
elif x>=2:  
    x+=1
```

Si **x** appartient à l'intervalle $[0, 2[$, il est inchangé. Il est parfaitement inutile d'écrire quelque chose comme **x=x** dans un bloc **else**.

Prenons un exemple complet un peu plus complexe, la résolution d'une équation polynomiale de degré 2 sur les réels, en supposant que les variables **a**, **b** et **c** contiennent des flottants, avec **a** non nul.

```
Delta=b**2-4*a*c
if Delta<0:
    print("Pas de racines !")
elif Delta>0:
    r=sqrt(Delta)
    x1=(-b-r)/(2*a)
    x2=(-b+r)/(2*a)
    print("Il y a deux racines distinctes, qui sont: ",x1,"et",x2)
else:
    print("Il y a une racine double, qui est: ",-b/(2*a))
```

Écriture en ligne. Il n'est en fait pas obligatoire de faire un saut de ligne après un **if**, **elif** ou **else**, en particulier s'il n'y a qu'une instruction à écrire. Cela est valable également pour les boucles et les fonctions (voir la suite). Par exemple le code suivant est équivalent à celui vu plus haut :

```
if note>=16: print("Mention Très bien")
elif note>=14: print("Mention Bien")
elif note>=12: print("Mention Assez bien")
elif note>=10: print("Mention Passable")
else: print("Raté !")
```

Exercice A

1. Écrire un programme qui demande à l'utilisateur de saisir un nombre entier et affiche si ce nombre est pair ou impair.
2. Écrire un programme qui demande à l'utilisateur de saisir un nombre et affiche si ce nombre est positif, négatif ou zéro.
3. Écrire un programme qui demande à l'utilisateur de saisir deux nombres et une opération (addition, soustraction, multiplication, division). Le programme doit ensuite effectuer l'opération et afficher le résultat.

4. Comparaison de Trois Nombres

Écrire un programme qui demande à l'utilisateur de saisir trois nombres et affiche le plus grand des trois.

5. Année Bissextile

Écrire un programme qui demande à l'utilisateur de saisir une année et vérifie si cette année est bissextile.

6. Notes et Mention

Écrire un programme qui demande à l'utilisateur de saisir une note (sur 20) et affiche la mention correspondante :

- "Très bien" si la note est supérieure ou égale à 16.
 - "Bien" si la note est comprise entre 14 et 15.9.
 - "Assez bien" si la note est comprise entre 12 et 13.9.
 - "Passable" si la note est comprise entre 10 et 11.9.
 - "Insuffisant" si la note est inférieure à 10.
 - Calcul du Prix avec Réduction
7. Écrire un programme qui demande à l'utilisateur de saisir le prix d'un article et le type de réduction (10%, 20%, ou 30%). Le programme doit ensuite afficher le prix final après application de la réduction.

8. Calcul de l'IMC

Écrire un programme qui demande à l'utilisateur de saisir son poids (en kg) et sa taille (en mètres), puis calcule et affiche son Indice de Masse Corporelle (IMC) ainsi que la catégorie correspondante :

- "Maigreur" si l'IMC est inférieur à 18.5.
- "Normal" si l'IMC est compris entre 18.5 et 24.9.
- "Surpoids" si l'IMC est compris entre 25 et 29.9.
- "Obésité" si l'IMC est supérieur ou égal à 30.

9. Calcul des Impôts

Écrire un programme qui demande à l'utilisateur de saisir son revenu annuel et calcule l'impôt à payer en fonction des tranches suivantes :

- 0% pour les revenus jusqu'à 10,000 euros.
- 10% pour les revenus entre 10,001 et 25,000 euros.
- 20% pour les revenus entre 25,001 et 50,000 euros.
- 30% pour les revenus au-delà de 50,000 euros.

10. Tri de Trois Nombres

Écrire un programme qui demande à l'utilisateur de saisir trois nombres et les affiche en ordre croissant.

Exercice B

« Premiers pas en Python »

1. Affectez les variables temps et distance par les valeurs 6.892 et 19.7. Calculez et affichez la valeur de la vitesse. Améliorez l'affichage en imposant un chiffre après le point décimal.

2. Saisir un nom et un âge en utilisant l'instruction `input()`. Les afficher.
Refaire la saisie du nom, mais avec l'instruction `raw_input()`. L'afficher.
Enfin, utilisez la « bonne pratique » : recommencez l'exercice en *translating* les saisies effectuées avec l'instruction `raw_input()`

« Structure conditionnelle »

1. Saisissez un flottant. S'il est positif ou nul, affichez sa racine, sinon affichez un message d'erreur.
2. L'ordre lexicographique est celui du dictionnaire. Saisir deux mots, comparez-les pour trouver le « plus petit » et affichez le résultat.
Refaire l'exercice en utilisant l'instruction ternaire : `<res> = <a> if <condition> else `

3. On désire sécuriser une enceinte pressurisée.

On se fixe une pression seuil et un volume seuil : `pSeuil = 2.3, vSeuil = 7.41`.

On demande de saisir la pression et le volume courant de l'enceinte et d'écrire un script qui simule le comportement suivant :

- si le volume et la pression sont supérieurs aux seuils : arrêt immédiat ;
- si seule la pression est supérieure à la pression seuil : demander d'augmenter le volume de l'enceinte ;
- si seul le volume est supérieur au volume seuil : demander de diminuer le volume de l'enceinte ;
- sinon déclarer que « tout va bien ».

Ce comportement sera implémenté par une alternative multiple.

4. Initialisez deux entiers : `a = 0` et `b = 10`.

Écrire une boucle affichant et incrémentant la valeur de `a` tant qu'elle reste inférieure à celle de `b`.

Écrire une autre boucle décrémentant la valeur de `b` et affichant sa valeur si elle est impaire. Boucler tant que `b` n'est pas nul.

F. Boucle

1. Boucle conditionnelle while (tant que)

La boucle while permet de réaliser une suite d'instructions tant qu'une certaine condition est vraie. La structure est la suivante.

```
while expression:  
    [instructions]
```

Le mécanisme est le suivant : on évalue **expression**. Si le résultat est **True**, on effectue toutes les instructions du bloc indenté, puis on recommence l'évaluation de **expression**. Sinon, on passe aux instructions situées après la boucle.

Par exemple, la séquence :

```
i=0
while i<10:
    print(i)
    i+=1          #un raccourci pour i=i+1
print("fini !")
```

affiche à l'écran tous les nombres entre 0 et 9, puis « fini ! ». En effet, lorsque *i* atteint 9, on l'affiche à l'écran, puis on incrémente *i* (qui vaut 10 en bas de la boucle). On réévalue ensuite la condition, mais **10<10** s'évalue en **False**, donc on sort de la boucle, et on affiche « fini ! » qui est une expression en dehors du corps de la boucle.

Notez bien que l'expression est évaluée uniquement en haut de la boucle : si elle s'évalue en **True**, on effectue toutes les instructions du corps de boucle, et on réitère l'évaluation. La boucle suivante affiche les entiers de 1 à 10, puis « fini ! ».

```
i=0
while i<10:
    i+=1
    print(i)
print("fini !")
```

Il se peut très bien qu'à la première évaluation de l'expression, celle-ci soit **False** : dans ce cas on n'effectue jamais le corps de boucle :

```
i=-1
while i>=0:
    print("on n'affichera jamais ça.")
```

Enfin, on fera attention avec les boucles **while**, si on s'y prend mal, on crée un morceau de code qui boucle sans fin :

```
while True:           #l'expression s'évalue en True !
    print("ce texte sera affiché, encore et encore !")
```

L'obtention d'une « boucle infinie » est parfois plus subtile :

```
x=0.1
while x!=1:
    x=x+0.1
```

On verra dans le chapitre sur la représentation des nombres qu'en arithmétique flottante, additionner 10 fois 0.1 ne fait pas tout à fait 1 (le résultat est évidemment très proche, mais ne vaut pas exactement 1).

2. Boucle inconditionnelle for... (pour...)

En informatique, on a très souvent besoin qu'une variable prenne successivement comme valeur tous les entiers entre deux bornes, par exemple de 0 à 100. Évidemment, on peut réaliser ça comme dans la section précédente avec une boucle **while** :

```
i=0
while i<=100:
    [instructions]
    i=i+1
```

Il est intéressant de raccourcir cette écriture, pour ne pas avoir à initialiser manuellement **i** ou écrire l'incrémentation **i+=1**, limiter les erreurs possibles et rendre le code plus lisible. On utilise alors une boucle *inconditionnelle* (**i** prend toutes les valeurs entières de 0 à 100 sans condition) : la boucle **for**.

Dans certains langages de programmation, celle-ci ne diffère conceptuellement pas d'une boucle **while** et est traduite ainsi au moment de la compilation du programme. Par exemple en C, qui est un langage très populaire :

Une boucle for, en langage C

```
for (i=0; i<=100; i++) {
    [instructions]
}
```

Traduction à la compilation

```
i=0 ;
while (i<=100) {
    [instructions]
    i++ ;
}
```

En Python, la structure de la boucle **for** est légèrement différente, c'est celle-ci :

```
for element in iterable:
    [instructions]
```

L'itérable est quelque chose que l'on peut itérer : en gros, c'est quelque chose qui fournit une séquence de valeurs. La syntaxe **for element in iterable** signifie que la variable **element** doit prendre successivement toutes les valeurs que fournit l'itérable. Pour chacune de ces valeurs, on exécute les instructions du corps de boucle. Bien souvent, on utilisera le constructeur **range** qui fournit des suites (finies) d'entiers. La syntaxe est la suivante, tous les paramètres *m*, *n* et *p* intervenant sont des entiers :

- pour $n \geq 0$, **range(n)** fournit tous les entiers de 0 inclus à *n* exclus (attention, on s'arrête donc à $n - 1$!)
- on peut décider de commencer à un autre entier que 0 en précisant un autre paramètre : pour $m \leq n$, **range(m,n)** fournit tous les entiers de *m* inclus à *n* exclus.

En précisant un troisième paramètre, on peut faire varier le pas :

- si $p > 0$, **range(m,n,p)** fournit successivement les entiers $m, m + p, m + 2p, \dots$ strictement inférieurs à n ;
- Si $p < 0$, **range(m,n,p)** fournit successivement les entiers $m, m + p, m + 2p, \dots$ strictement supérieurs à n .

Par exemple la boucle :

```
for i in range(0,m,2):
    print(i)
```

affiche à l'écran successivement tous les entiers pairs entre 0 et $m - 1$ (la borne m est exclue). La boucle suivante affiche tous les entiers de $n - 1$ à 0 , (dans l'ordre décroissant) :

```
for i in range(n-1,-1,-1):
    print(i)
```

Petit conseil : apprendre par cœur la syntaxe **range(n-1,-1,-1)**, elle sert souvent. Donnons un exemple un peu plus complet : le calcul de **10!**

```
x=1
n=10
for i in range(1,n+1):
    x*=i           #un raccourci pour x=x*i
```

On peut vérifier que la variable **x** contient bien $10! = 3628800$ à l'issue de cette boucle.

L'itérable peut également être une liste (dans ce cas **element** prend successivement toutes les valeurs de la liste), ou une chaîne de caractères (dans ce cas, **element** prend successivement comme valeurs tous les caractères de la chaîne), ou encore un tuple (*n-uplet*)... Ces types seront examinés un peu plus bas.

Donnons un petit exemple, montrant que Python peut faire beaucoup de choses (l'exemple n'est pas à retenir). Le module **itertools** permet de faire de la combinatoire. Considérons le triplet **(1, 4, 7)**. On peut facilement produire toutes les permutations possibles du triplet avec la fonction **permutations** du module **itertools** :

```
import itertools          # pour pouvoir utiliser le module
iterable=itertools.permutations((1,4,7))
for x in iterable:
    print(x)
```

Le code précédent affiche à l'écran :

```
(1, 4, 7)
(1, 7, 4)
(4, 1, 7)
(4, 7, 1)
(7, 1, 4)
(7, 4, 1)
```

3. Break et Continue

Ces instructions ne sont pas exigibles, mais sont parfois très pratiques (surtout **break**). Dans une boucle (**while** ou **for**), on peut utiliser les commandes **break** et **continue** : **break** sort de la boucle, et **continue** poursuit l'itération en revenant « tout en haut », sans se préoccuper de ce qu'il y a derrière. Voici un exemple un peu stupide qui mélange les deux :

```
u=0
while u<100:
    print(u)
    u+=1 #un raccourci pour u=u+1
    if u==5:
        break
    continue
    print(1/0)
```

Ce code ne produit pas d'erreur et n'affiche que 6 entiers : 0, 1, 2, 3, 4 et 5. Dans le corps de boucle, la partie **print(1/0)** n'est jamais exécutée (ce qui produirait une erreur) puisqu'elle se trouve après le **continue**.

Lorsque **u** atteint 5, on rentre dans le **if** et on sort de la boucle avec **break**.

Exercice

1. Écrire une *saisie filtrée* d'un entier dans l'intervalle 1 à 10, bornes comprises. Affichez la saisie.
2. Affichez chaque caractère d'une chaîne en utilisant une boucle **for**.
Affichez chaque élément d'une liste en utilisant une boucle **for**.
3. Affichez les entiers de 0 à 15 non compris, de trois en trois, en utilisant une boucle **for** et l'instruction **range()**.
4. Utilisez l'instruction **break** pour interrompre une boucle **for** d'affichage des entiers de 1 à 10 compris, lorsque la variable de boucle vaut 5.
5. Utilisez l'instruction **continue** pour modifier une boucle **for** d'affichage de tous entiers de 1 à 10 compris, sauf lorsque la variable de boucle vaut 5.
6. Compter les Voyelles dans une Chaîne
Écrire un programme qui demande à l'utilisateur de saisir une chaîne de caractères et utilise une boucle **for** pour compter et afficher le nombre de voyelles dans cette chaîne.
7. Multiples de 3 et 5
Écrire un programme qui utilise une boucle **for** pour afficher tous les nombres entre 1 et 50 qui sont multiples de 3 ou de 5.
8. Somme des Nombres Pairs

Écrire un programme qui utilise une boucle **for** pour calculer et afficher la somme des nombres pairs entre 1 et 100. Factorielle d'un Nombre

9. Écrire un programme qui demande à l'utilisateur de saisir un nombre entier positif et utilise une boucle **while** pour calculer et afficher sa factorielle.
10. Table de Multiplication
Écrire un programme qui demande à l'utilisateur de saisir un nombre entier et utilise une boucle **for** pour afficher sa table de multiplication jusqu'à 10.
11. Utilisez une **exception** pour calculer, dans une boucle évoluant de -3 à 3 compris, la valeur de $\sin(x)/x$.
12. La clause **else** des boucles. Dans cet exercice, effectuez les saisies avec des **integerbox** et les affichages avec des **msgbox**, tous deux appartenant au module **easygui**. Initialisez une liste avec 5 entiers de votre choix puis saisissez un entier. Dans une boucle for, parcourez la liste. Si l'entier saisi appartient à la liste, sauvez-le et interrompez la boucle (puisque vous l'avez trouvé). Si la boucle s'est bien terminée, utilisez une clause **else** pour afficher un message l'annonçant. Entrez maintenant un autre entier, cette fois-ci positif.

Écrivez une boucle **while** pour déterminer si cet entier est premier. S'il ne l'est pas, la boucle devra afficher le premier diviseur trouvé et s'interrompre. S'il est premier, l'afficher dans une clause **else**.

G. Fonctions

Les fonctions sont d'une importance capitale en informatique, et plus prosaïquement quasiment toutes les questions des sujets de concours demandent d'écrire ou d'examiner des fonctions.



1. Notions et syntaxe de base

Une fonction en informatique est une séquence d'instructions, dépendant de paramètres d'entrée (appelés *arguments*), et retournant un résultat.

Deux points de vue, souvent complémentaires, permettent de préciser ce qu'est une fonction :

- c'est une séquence d'instructions qui permet de réaliser un calcul précis, que l'on peut utiliser plusieurs fois.
- c'est une brique de base d'un problème plus complexe.

La structure générale d'une déclaration de fonction en Python se fait avec le mot-clef **def** de la façon suivante :

Déclaration d'une fonction

```
def nom_fonction(a_1,a_2,...,a_k): # nom_fonction: nom de la fonction, a_1,...,a_k : arguments
    """ Description de l'action de la fonction """
    instruction 1
    instruction 2
    ...
    instruction p
# ici, on est hors de la définition de la fonction.
```

- La première ligne **def nom_fonction(a₁,...,a_k)** est l'en-tête de la fonction. Les éléments **a₁,...,a_k** sont des identificateurs appelés arguments formels de la fonction et **nom_fonction** est le nom de la fonction. Pour une fonction ne prenant pas d'arguments, on écrit simplement **def fonction():**, les parenthèses étant indispensables. Le nom de la fonction est un identificateur qui suit les mêmes règles que les identificateurs de variables.
- La seconde (qui est facultative et peut être sur plusieurs lignes) est une chaîne de caractères appelée chaîne de documentation décrivant la fonction : ce que doivent respecter les paramètres passés en entrée, l'action effectuée et la nature du résultat retourné.
- La suite d'instructions est le corps de la fonction.
- Le retour à une indentation au même niveau que **def** marque la fin de la fonction, tout ce qui est à ce niveau ne fait plus partie de la fonction.

Attention, le rôle d'une définition de fonction n'est pas d'exécuter les instructions qui en composent le corps, mais uniquement de mémoriser ces instructions en vue d'une exécution ultérieure (facultative !), provoquée par une expression faisant appel à la fonction.

Par exemple, définir la fonction qui suit ne provoque pas d'erreur.

```
def fonction_erreur():
    print(1/0)
```

Évidemment, l'appeler en provoque une !

Appel d'une fonction. L'appel de la fonction `nom_fonction` présentée ci-dessus se fait par :

```
nom_fonction(e_1,e_2,...,e_k),
```

où `e_1,...,e_k` sont des *expressions*. Elles forment les *arguments effectifs* de l'appel à la fonction : lors de l'appel `e_1` (respectivement `e_2,...,e_k`) est évaluée, puis le résultat est affecté à `a_1` (respectivement `a_2,...,a_k`) juste avant l'exécution du corps de la fonction. Tout se passe comme si l'exécution de la fonction commençait par la suite d'instructions d'affectation

```
a_1=e_1  
a_2=e_2  
...  
a_k=e_k
```

et se poursuivait avec

```
instruction 1  
instruction 2  
...  
instruction p
```

Exemple

```
def nom_de_la_fonction(parametre1, parametre2):  
    resultat = parametre1 + parametre2  
    return resultat
```

Pour utiliser une fonction, il suffit de l'appeler en utilisant son nom suivi de parenthèses. Si la fonction prend des paramètres, ils doivent être fournis dans ces parenthèses.

```
somme = nom_de_la_fonction(3, 5)  
print(somme)                                # Affiche 8
```

Paramètres et Arguments

Paramètres : Ce sont les variables dans la définition de la fonction.

Arguments : Ce sont les valeurs réelles passées à la fonction.

L'instruction return. Le corps de la fonction comprend bien souvent une ou plusieurs instructions de la forme `return resultat`, où `resultat` est une expression. Lors du déroulement du corps de la fonction, si une telle instruction est rencontrée, alors l'expression `resultat` est évaluée, l'exécution de la fonction est interrompue et la valeur de `resultat` prend la place de `nom_fonction(e_1,e_2,...,e_k)` là où la fonction a été appelée. Prenons un exemple simple :

```
def incremente(x):  
    return x+1
```

Si on exécute `a=incremente(6-2)+9`, alors :

- `6-2` s'évalue en **4**.
- `x` prend la valeur 4 dans la fonction.
- `x+1` s'évalue en 5, qui est retourné par la fonction.
- `incremente(6-2)` est donc remplacée par 5.
- `incremente(6-2)+9` s'évalue donc en 14, qui est ensuite affecté à la variable `a`.

Retourner une Valeur

Une fonction peut retourner une valeur à l'aide du mot-clé `return`. Ce mot-clé met également fin à l'exécution de la fonction.

```
def multiplier(a, b):  
    return a * b  
  
resultat = multiplier(4, 7)  
print(resultat)      # Affiche 28
```

Le type « rien ». Si la fonction ne comprend pas de `return` ou qu'aucun `return` n'est rencontré lors de l'exécution, la fonction ne renvoie **rien**. On dit parfois qu'elle fonctionne uniquement par **effets de bord**, et c'est là une différence fondamentale avec les fonctions en mathématiques. Il y a un type pour ça : `NoneType`, qui comporte une unique valeur : `None`. La fonction suivante ne prend aucun paramètre en entrée, se contente d'afficher 4 à l'écran, et ne renvoie rien : elle agit par effets de bord.

```
def affiche4():  
    print(4)
```

Lors de l'évaluation de `a=affiche4()`, 4 est affiché à l'écran, on sort de la fonction (car on est arrivé en bas !) et a prend la valeur `None`. Notez que `return` seul (sans rien derrière) est souvent fort utile pour interrompre une fonction. La fonction en question renvoie alors `None`.

Différence entre `print` et `return`. Une erreur classique est de confondre `print` et `return`. `return` est une instruction de sortie de fonction, `print` est une fonction Python, qui affiche l'argument passé en entrée à l'écran et qui ne renvoie rien. Lorsqu'on teste une fonction dans la console, on ne voit pas vraiment la différence mais elle est pourtant significative : une fonction sans `return` ne renvoie rien !

Exemple

Une fonction n'a pas nécessairement besoin de retourner une valeur. Si aucune valeur n'est retournée, la fonction renvoie implicitement `None`.

```
def afficher_message():  
  
    print("Ceci est un message.")
```

```
afficher_message() # Affiche "Ceci est un message."
```

Prenons l'exemple de la fonction suivante, qui calcule un PGCD

```
def PGCD(a,b):
    """Avec a et b>0, renvoie le PGCD de a et b."""
    while b!=0:
        a,b=b,a%b
    return a
```

Exécuté dans la console, on ne verrait pas de grande différence entre cette fonction et la même avec **print** à la place de **return**. Si maintenant, on veut utiliser cette fonction pour calculer un PPCM, on définit alors la fonction suivante :

```
def PPCM(a,b):
    """Avec b>0, renvoie le PPCM de a et b."""
    return a*b//PGCD(a,b)
```

L'appel **PPCM(6,4)** produit bien 12. Avec **print** a au lieu de **return** a dans la fonction **PGCD**, l'expression **PGCD(6,4)** est remplacée par **None**, et l'évaluation **a*b//PGCD(a,b)** produit une erreur :

```
>>> PPCM(6,4)
2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in PPCM
TypeError: unsupported operand type(s) for //: 'int' and 'NoneType'
```

Remarquez que l'erreur est très explicite : l'opérateur **//** ne peut faire d'opération entre un entier (de type **int**, obtenu ici par l'évaluation de **6*4**) et un objet de type **NoneType**, c'est-à-dire **None**. Vous vous posez peut-être la question : mais qu'est-ce que le 2 qui traîne ? Il provient de notre fonction **PGCD**, qui a été appelée par **PPCM**, s'est déroulée sans accroc, et a bravement affiché à l'écran le PGCD de *a* et *b*, comme demandé puisqu'on a utilisé **print**.



Chaîne de documentation. Il est important de préciser ce que fait une fonction lorsqu'on l'écrit. La chaîne de documentation ainsi que l'en-tête, sont accessibles lorsqu'on tape **help(nom_fonction)** :

```
>>> help(PPCM)
Help on function PPCM in module __main__:

PPCM(a, b)
    Avec b>0, renvoie le PPCM de a et b.
```

Et cela marche aussi avec les fonctions Python.

```
>>> from math import *
>>> help(log2)
Help on built-in function log2 in module math:

log2(...)
    log2(x)

    Return the base 2 logarithm of x.
```

Il existe certaines règles qui régissent la rédaction des chaînes de documentation, mais il est inutile de s'embêter avec ça. Mettez une chaîne de caractères après l'en tête qui explique un peu ce que fait votre fonction et ce sera déjà très bien. Cette chaîne est bien sûr facultative.

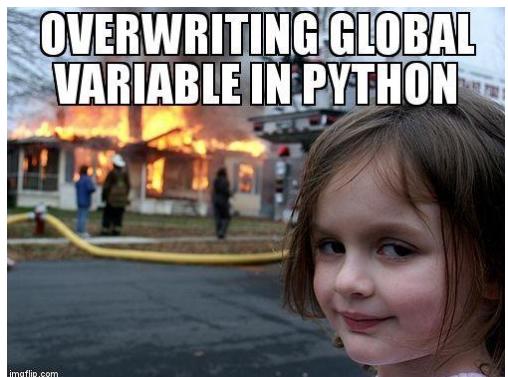
Définition d'une fonction avec l'opérateur lambda On peut avoir besoin d'une fonction simple (par exemple pour la transmettre en argument d'une autre) sans avoir envie de lui donner un nom (et la définir avec def). Pour cela, on dispose de l'opérateur lambda

```
>>> f = lambda x : x * x
>>> f (2)
4
>>> ( lambda x , y : x + y ) (3 ,4)
7
```

2. Variables locales et globales

Dans les fonctions PGCD et PPCM de la sous-section précédente, les variables *a* et *b* ont été utilisées à peu près partout : comme paramètres d'appel des deux fonctions mais également comme variables pour calculer le PGCD puisque par exemple la valeur de retour de PGCD est *a*. Pourtant, Python ne se mélange pas les pinceaux et produit le résultat auquel on s'attend, parce que *a* et *b* dans les fonctions sont des variables *locales*.

Si elle n'est pas explicitement déclarée **globale**, toute variable apparaissant dans une fonction comme membre gauche d'une affectation est **locale** à cette fonction. Cela signifie que sa *portée* est réduite à la fonction, qu'elle est créée à chaque fois que la fonction est appelée et « détruite » à la fin de chaque exécution. Par exemple, supposons que la variable *a* n'ait pas été affectée mais la fonction PGCD précédente déclarée :



```
>>> PGCD(8,3);
1
```

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

On voit bien que *a* est inconnu en dehors de la fonction PGCD. Les paramètres des fonctions se comportent comme des variables locales : on peut les modifier mais cette modification est interne à la fonction :

```
>>> a=5 ; b=7
>>> p=PGCD(a,b)
>>> print(a)
5
```

On parle de passage par valeurs : lors de l'appel à PGCD, les valeurs de *a* et *b* sont recopiées et les variables *a* et *b* de la fonction ne sont pas les mêmes que les variables *a* et *b* qu'on a définies en dehors de la fonction.

A l'opposé de cela, les **variables globales** sont des variables créées à l'extérieur de toute fonction. Elles existent depuis le moment de leur création, jusqu'à la fin de l'exécution du programme. Une variable globale peut être utilisée à l'intérieur d'une fonction si elle n'est pas le membre gauche d'une affectation ou le nom d'un paramètre. Par exemple, la fonction suivante ajoute le contenu de la variable (nécessairement globale !) *n* au paramètre *x* et renvoie le résultat de l'addition.

```
def ajoute_n(x):
    return x+n
```

Si *n* n'est pas défini au moment de l'appel à **ajoute_n**, on obtient bien sûr une erreur. En général, on réserve l'usage des variables globales aux constantes d'un problème. Si on veut réaliser une simulation en cinétique des gaz, on pourra commencer le script de simulation par **R=8,3144621** (*la constante universelle des gaz parfaits*), et on pourra librement utiliser **R** dans n'importe quelle fonction.

Pour pouvoir affecter à une **variable globale** dans une fonction, cette variable doit faire l'objet d'une déclaration explicite comme variable globale de la forme **global variable_globale**. Par exemple, imaginons que l'on veuille maintenir un compteur, qui contient le nombre de fois où une certaine fonction a été appelée. On peut donc utiliser une variable globale **nombre_appels**, qu'on incrémentera de 1 dans l'appel de la fonction. La fonction en question commencerait donc par :

```
global nombre_appels
nombre_appels+=1
```

Ainsi, une variable est locale sauf si :

- elle est explicitement déclarée globale ;
- ou bien elle est utilisée sans être affectée.

En général, on n'utilisera pas de variables globales, sauf si la situation le justifie. À l'opposé des fonctions qui fonctionnent par effets de bord, il y a les fonctions pures : elles n'agissent pas sur l'environnement (même par

affichage !) et n'en dépendent pas (elles n'ont pas recours à des variables globales). Par exemple, les fonctions PGCD et PPCM définies plus haut sont pures.

3. Typage des arguments, du retour - Signature d'une fonction

On peut faire apparaître un « typage » des fonctions (c'est notamment le cas dans les sujets de Centrale). Par exemple

```
def multiplie ( chaine : str , nombre : int ) -> str :
    return nombre * chaine
```

Dans notre utilisation, ce typage n'est qu'une indication (Python ne va pas contrôler le type des arguments - ça peut être utilisé par d'autres programmes d'analyse ou de vérification de code).

Les types usuelles : **int**, **float**, **str**, **list** (et **list[int]** pour des listes d'entiers), **tuple**, **dict**, **Callable** (pour les fonctions)

Exercice 3.1 (renverser une chaîne)

Écrire une fonction qui prend une chaîne en argument et qui renvoie la chaîne renversée. Par exemple inverse('bonjour') renvoie 'ruojnob'.

Exercice 3.2 (Palindrome)

Écrire deux fonctions qui testent si une chaîne est un palindrome (elle se lit dans les deux sens) - l'une pourra utiliser la fonction de l'exercice précédent, l'autre non.

Exercice

Exercice 1 :

Écrire une procédure **table** avec quatre paramètres : **base**, **début**, **fin** et **inc**. Cette procédure doit afficher la table des **base**, de **début** à **fin**, de **inc** en **inc**.

Tester la procédure par un appel dans le programme principal.

Exercice 2

Écrire une fonction **cube** qui retourne le cube de son argument.

Écrire une fonction **volume_Sphere** qui calcule le volume d'une sphère de rayon r fourni en argument et qui utilise la fonction **cube**.

Tester la fonction **volume_Sphere** par un appel dans le programme principal

Exercice 3 : Calcul de la Factorielle

Écrire une fonction **factorielle(n)** qui prend en paramètre un entier n et retourne sa factorielle.

Exercice 4 : Calcul de la Puissance

Écrire une fonction **puissance(base, exposant)** qui prend deux paramètres, base et exposant, et retourne base élevé à la puissance exposant.

Exercice 5 : Maximum de Deux Nombres

Écrire une fonction **max_deux_nombres(a, b)** qui retourne le plus grand des deux nombres a et b.

Exercice 6 : Calcul de Moyenne

Écrire une fonction **moyenne(liste_nombres)** qui prend en paramètre une liste de nombres et retourne la moyenne de ces nombres.

Exercice 7 : Conversion Celsius en Fahrenheit

Écrire une fonction **celsius_to_fahrenheit(celsius)** qui convertit une température en Celsius en Fahrenheit.

Exercice 8

Écrire une fonction **maFonction** qui retourne $f(x) = 2x^3 + x - 5$.

Écrire une procédure tabuler avec quatre paramètres : **fonction**, **borneInf**, **borneSup** et **nbPas**. Cette procédure affiche les valeurs de **fonction**, de **borneInf** à **borneSup**, tous les **nbPas**. Elle doit respecter **borneIn f < borneSup**.

Tester cette procédure par un appel dans le programme principal après avoir saisi les deux bornes dans une **floatbox** et le nombre de pas dans une **integerbox** (utilisez le module **easyguiB**).

Exercice 9

Écrire une fonction **volMasseEllipsoide** qui retourne le volume et la masse d'un ellipsoïde grâce à un tuple. Les paramètres sont les trois demi-axes et la masse volumique. On donnera à ces quatre paramètres des valeurs par défaut.

On donne : $v = \frac{4}{3}\pi abc$

Tester cette fonction par des appels avec différents nombres d'arguments.

H. Listes

On étudie ici le type **list** en Python, qui est essentiel et nous servira souvent. L'appellation **list** de Python est un peu malheureuse et la traduction en « liste » maladroite : en toute rigueur, il faudrait parler de « tableau redimensionnable inhomogène ». Mais les sujets de concours parlent de listes, donc nous aussi. Voici un exemple de liste : `[True, 4, 5, 3.0]`.

Comme l'exemple le montre, les listes sont des séquences finies d'éléments, possiblement de types différents. La syntaxe consiste à les mettre entre crochets, séparés par des virgules.

1. Construction de listes

On peut construire une liste de plusieurs manières :

- par la donnée explicite des éléments, entre crochets, séparés par des virgules, comme ci-dessus.
- par concaténation de listes (à l'aide de +) : `[1, 2, 3]+[4, 5, 6]` s'évalue en `[1, 2, 3, 4, 5, 6]`.
- `list(iterable)` permet de fabriquer une liste à partir d'un **itérable**. Par exemple, `list(range(4))` s'évalue en `[0, 1, 2, 3]` et `list("truc")` en `['t', 'r', 'u', 'c']`.
- par **compréhension**, très pratique avec la structure suivante :
`L=[f(x) for x in iterable if P(x)]`, où `f(x)` est une expression dépendant (ou non) de `x`, et `P(x)` est une expression booléenne (facultative).

Activité 1

Quel est le contenu du tableau associé à `mon_tab` après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```
mon_tab = [p for p in range(0, 5)]
```

Nous avons une boucle for entre crochets. `p` va successivement prendre les valeurs 0, 1, 2, 3, 4. Ces différentes valeurs de `p` vont permettre de remplir le tableau `mon_tab`.

Activité 2

Quel est le contenu du tableau associé à `mon_tab` après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```
l = [1, 7, 9, 15, 5, 20, 10, 8]
mon_tab = [p for p in l if p > 10]
```

Ci-dessus nous utilisons le tableau `l` pour créer le tableau `mon_tab` : on parcourt le tableau `l` grâce à la boucle `for p in l` mais on "garde" uniquement les valeurs supérieures à 10 (grâce au `if p > 10`).

Après l'exécution du programme ci-dessus, le tableau `mon_tab` est constitué des éléments suivants : `[15, 20]`



Autre possibilité, utiliser des composants "arithmétiques" :

Activité 3

Quel est le contenu du tableau associé à *mon_tab* après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```

l = [1, 7, 9, 15, 5, 20, 10, 8]
mon_tab = [p**2 for p in l if p < 10]
    
```

Rappel : p^{**2} permet d'obtenir la valeur de p élevée au carré

Comme vous pouvez le remarquer, nous obtenons un tableau (*mon_tab*) qui contient tous les éléments du tableau *l* élevés au carré à condition que ces éléments de *l* soient inférieurs à 10. Comme vous pouvez le constater, la compréhension de tableau permet d'obtenir des combinaisons relativement complexes.

Activité 4

Quel est le contenu du tableau associé à *mon_tab* après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```

mon_tab = [x*x for x in range(5) if x%2==0]
    
```

- par *slicing* (tranchage), qu'on va voir bientôt.

2. Accès aux éléments

Pour *L* une liste, sa *longueur* (nombre d'éléments, *length* en anglais) est accessible avec `len(L)`. En notant *n* cette longueur, les éléments sont indexés par les entiers de *0* à *n - 1*.

Activité 5

Quelle est la valeur associée au nom *nb_ele* après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```
L = [5, 8, 6, 9]
nb_ele = len(L)
```

Exemples :

```
>>> L=list(range(1,6)) #range(1,6) fournit les entiers de 1 à 5.
>>> L[2]
3
>>> L[len(L)-1]
5
```

Si on demande l'accès à un caractère d'indice négatif i compris entre -1 et $-n$, où n est la longueur de la liste, celui-ci est considéré comme étant $n + i$:

```
>>> L[-1] # très pratique pour accéder au dernier élément !
5
>>> L[-5]
1
```

L'accès à tout autre indice ie un indice qui dépasse la **valeur_longueur_de_la_liste -1** provoquera produit une erreur :

```
>>> L[len(L)]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Retenez bien cette erreur, vous l'aurez souvent !

Me: submits my code for the
100th time praying it will work My code:
IndexError: list index out of range



3. Modification d'un élément

Étant donnée une liste L , on peut modifier l'élément d'indice i de L en le remplaçant par l'élément de son choix. La syntaxe est la même que pour une affectation

```
>>> L=list(range(1,6))
>>> L
[1, 2, 3, 4, 5]
>>> L[2]=10
>>> L
[1, 2, 10, 4, 5]
```

Les règles régissant l'indice i sont les mêmes que précédemment.

Suppression d'un élément d'une liste par l'instruction del

L'instruction `del` (qui n'est pas une fonction) permet de supprimer un élément en donnant son indice.

```
>>> L= [8, 4, 2, 5, 7]
>>> del L[3]
>>> L
[8, 4, 2, 7]
```

4. Slicing (tranchage)

On peut créer une nouvelle liste en extrayant certains éléments d'une liste. Pour extraire les éléments d'indice entre d inclus et $f \geq d$ exclu, on utilise `L[d:f]` : la liste obtenue est donc composée des éléments `L[d], L[d+1], ..., L[f-1]`. L'un ou l'autre de ces deux indices peut être omis, et même les deux (dans ce cas, d vaut 0 et f vaut la longueur de la liste). Ce mécanisme est tolérant envers les indices trop grands ou trop petits (attention, les indices négatifs entre -1 et $-n$ sont interprétés comme précédemment), et si $d \geq f$, on obtient la liste vide.

```
>>> L
[1, 2, 10, 4, 5]
>>> L[3:4]
[4]
>>> L[4:3]
[]
>>> L[:4]
[1, 2, 10, 4]
>>> L[:8]
[1, 2, 10, 4, 5]
```

On peut également spécifier un pas, positif ou négatif. L'interprétation est la même que pour les listes et l'itérateur `range`, on ne précisera donc pas ici.

```
>>> L[::-2]
[1, 10, 5]
```

5. Méthodes sur les listes

Python est un langage *orienté objet*. À chaque classe d'objets (comme les listes) peuvent s'appliquer plusieurs *méthodes*, qui modifient l'objet ou renvoient certaines de ses caractéristiques. Au programme en classes préparatoires, on trouve seulement `append` et `pop`, qui permettent d'ajouter ou d'enlever un élément en fin de liste. La syntaxe générale de l'utilisation d'une méthode sur un objet est la suivante : `objet.methode(paramètres)`.

Le tableau suivant récapitule les principales méthodes sur les listes. Le but est de présenter ce qu'on peut faire en Python, et de voir que ces opérations ne sont pas toutes triviales pour le processeur. La colonne complexité ne peut être comprise qu'après le chapitre dédié. On note n la longueur de la liste L .

méthode	description	complexité
<code>L.append(x)</code>	Ajoute x à la fin de L .	$O(1)$ (<i>amorti</i>)
<code>L.extend(T)</code>	Ajoute les éléments de T à la fin de L (équivalent à $L+=T$)	$O(\text{len}(T))$ (<i>amorti</i>)
<code>L.insert(i, x)</code>	Ajoute l'élément x en position i de L , en décalant les suivants vers la droite.	$O(n - i)$ (<i>amorti</i>)
<code>L.remove(x)</code>	Supprime de la liste la première occurrence de x si x est présent, sinon produit une erreur	$O(n)$.
<code>L.pop()</code>	Supprime le dernier élément de L , et le renvoie.	$O(1)$
<code>L.pop(i)</code>	Supprime l'élément d'indice i de L , en décalant les suivants vers la gauche. Cette méthode renvoie l'élément supprimé.	$O(n - i)$
<code>L.index(x)</code>	Retourne l'indice de la première occurrence de x dans L si x est présent, produit une erreur sinon.	$O(n)$
<code>L.count(x)</code>	Retourne le nombre d'occurrences de x dans L .	$O(n)$
<code>L.sort()</code>	Trie la liste L dans l'ordre croissant (en place).	$O(n \ln(n))$
<code>L.reverse()</code>	Renverse la liste (en place)	$O(n)$

Attention, ces méthodes ne renvoient en général rien : elles modifient l'objet. C'est le cas pour `append`, qui permet d'ajouter en fin de liste un nouvel élément. Pour ajouter x à la fin de L , on écrit simplement `L.append(x)`, et non pas `L=L.append(x)`. En effet, `L.append(x)` est une expression dont l'évaluation produit `None`, c'est-à-dire rien. Écrire `L=L.append(x)` reviendrait à affecter `None` à la variable `L`, ce qui n'est pas a priori ce qu'on veut faire ! Voir la section 1.5 pour des précisions sur `None`.

6. Listes et références

Ce point est important. Vous ferez l'erreur un jour, mais vous vous douterez du problème si vous avez bien compris ce paragraphe. Prenons tout de suite un exemple :

```
>>> T=[0,2,3]
>>> U=T
>>> T[0]=1
>>> T.append(4)
>>> print(U)
[1, 2, 3, 4]
```

Si on modifie T , on modifie U . Le comportement semble bien différent des variables ! Essayons autre chose :

```
>>> T=[0,2,3]
>>> U=T[:]
>>> T[0]=1
>>> T.append(4)
>>> print(U)
[0, 2, 3]
```

Le comportement est plus sympathique. Il faut savoir que lorsqu'on crée une liste, la variable utilisée est ce qu'on appelle une référence (ou un pointeur) vers l'emplacement mémoire où est stockée la liste. L'instruction `U=T` du premier exemple stocke dans la variable U la référence stockée dans T. Autrement dit, l'emplacement mémoire désigné par T et U est le même ! Lorsqu'on effectue l'instruction `T[0]=1`, on va modifier directement la mémoire (de même avec `append`), et il est logique que ce changement soit visible lorsqu'on tape `print(U)`, puisque cette action va chercher en mémoire ce qu'indique U.

Dans le deuxième exemple, l'instruction `U=T[:]` est différente : on crée une liste dont les éléments sont les mêmes que ceux de T, mais ailleurs en mémoire. Autrement dit, les références T et U pointent vers des endroits différents en mémoire, et donc si on modifie l'une, l'autre n'est pas modifié.

7. Listes de listes

On utilisera couramment des listes qui contiennent des listes, en particulier pour représenter des matrices. L'accès aux éléments se fait de manière similaire :

```
>>> L=[[0,1,2,3],[4,5]] # une liste de deux listes
>>> L[0] # premier élément de L
[0, 1, 2, 3]
>>> L[1][0] # premier élément du deuxième élément de L
4
>>> L[0][2]=6
>>> print(L)
[[0, 1, 6, 3], [4, 5]]
```

Prenons par exemple

```
L = [[1, 3, 4], [5 ,6 ,8], [2, 1, 3], [7, 8, 15]]
```

Le premier élément du tableau ci-dessus est bien un tableau ([1, 3, 4]),

Le deuxième élément est aussi un tableau ([5, 6, 8])...

Il est souvent plus pratique de présenter ces "Liste de listes" comme suit :

```
L = [[1, 3, 4],
      [5, 6, 8],
      [2, 1, 3],
      [7, 8, 15]]
```

Nous obtenons ainsi quelque chose qui ressemble beaucoup à un "**objet mathématique**" très utilisé : **une matrice**

Activité 1

Quelle est la valeur associée au nom a après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```
L = [[1, 3, 4],
      [5, 6, 8],
```

```
[2, 1, 3],  
[7, 8, 15]]  
a = L[1][2]
```

Comme vous pouvez le constater, le nom `a` est bien associé à l'entier situé à la 2e ligne (indice 1) et à la 3e colonne (indice 2), c'est-à-dire 8.

Activité 2

Quel est le contenu du tableau associé au nom `mm` après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```
L = [1, 2, 3]  
L1 = [L, L, L]  
L[0] = 100
```

Pour terminer, faisons une mise en garde supplémentaire, concernant encore les références :

```
>>> T=[[0,2],[3,4]]  
>>> U=T[:]  
>>> U[0][0]=1  
>>> print(T)  
[[1, 2], [3, 4]]
```

Ici, on a pris soin de recopier les éléments de `T`. Mais comme ces éléments sont des références vers `[0,2]` et `[3,4]`, le problème reste le même que précédemment puisque ces listes-là n'ont pas été recopiés. Il aurait fallu écrire :

```
U=[A[:] for A in T]
```

Mais si `T` avait été une liste de listes de listes, le problème se serait encore posé. Faisons deux remarques :

- premièrement, on manipulera rarement des listes de listes de listes ;
- deuxièmement, il existe un module `copy` dont la fonction `deepcopy` permet de copier « en profondeur » un objet.

8. La boucle ‘for’ : parcourir les élément d'un tableau

La boucle `for... in` permet de parcourir chacun des éléments d'une séquence:

Activité 1

Analysez puis testez le code suivant :

```
mon_tab = [5, 8, 6, 9]
for element in mon_tab:
    print(element)
```

Dans la boucle *for... in* il est possible d'utiliser la fonction prédefinie *range* à la place d'un tableau d'entiers:

Activité 2

Analysez puis testez le code suivant :

```
for element in range(0, 5):
    print (element)
```

Comme vous pouvez le constater, "**range(0,5)**" est, au niveau de la boucle "**for..in**", équivalent au tableau [0,1,2,3,4],

Le code ci-dessous donnerait le même résultat que le programme vu dans le " **Activité 2**" :

```
mon_tab = [0, 1, 2, 3, 4]
for element in mon_tab:
    print (element)
```

ATTENTION : si vous avez dans un programme "*range(a,b)*", *a* est la borne inférieure et *b* a borne supérieure. Vous ne devez surtout pas perdre de vu que la borne inférieure est incluse, mais que la borne supérieure est exclue.

Il est possible d'utiliser la méthode "**range**" pour "remplir" un tableau :

Activité 3

Quel est le contenu du tableau associé à *mon_tab* après l'exécution du programme ci-dessous ? (utilisez la console pour vérifier votre réponse)

```
mon_tab = []
for element in range(0, 5):
    mon_tab.append(element)
```

Exercice

Exercice 1

Définir la liste : *liste* =[17, 38, 10, 25, 72], puis effectuez les actions suivantes :

- triez et affichez la liste ;
- ajoutez l'élément 12 à la liste et affichez la liste ;
- renversez et affichez la liste ;
- affichez l'indice de l'élément 17 ;
- enlevez l'élément 38 et affichez la liste ;
- affichez la sous-liste du 2^eau 3^eélément ;
- affichez la sous-liste du début au 2^eélément ;
- affichez la sous-liste du 3^eélément à la fin de la liste ;
- affichez la sous-liste complète de la liste ;
- affichez le dernier élément en utilisant un indice négatif.

Bien remarquer que certaines méthodes de liste ne retournent rien.

Exercice 2

Initialisez `truc` comme une liste vide, et `machin` comme une liste de cinq flottants nuls.

Affichez ces listes.

Utilisez la fonction `range()` pour afficher :

- les entiers de 0 à 3 ;
- les entiers de 4 à 7 ;
- les entiers de 2 à 8 par pas de 2.

Définir `chose` comme une liste des entiers de 0 à 5 et testez l'appartenance des éléments 3 et 6 à `chose`.

Exercice 2

Utilisez une liste en compréhension pour ajouter 3 à chaque élément d'une liste d'entiers de 0 à 5.

Exercice 3

Ecrire un programme Python sous forme de fonction Python qui prend en paramètres deux listes et renvoie True si les deux listes ont au moins un élément commun et False si non.

Exercice 4: Accès et Boucle sur une Liste

1. Créez une liste avec les éléments suivants : "pomme", "banane", "orange", "cerise".
2. Affichez le premier et le dernier élément de la liste.
3. Parcourez la liste et affichez chaque fruit avec sa position dans la liste.

Exercice 5

Utilisez une liste en compréhension pour ajouter 3 à chaque élément d'une liste d'entiers de 0 à 5, mais seulement si l'élément est supérieur ou égal à 2.

Exercice 6

Utilisez une liste en compréhension pour calculer la somme d'une liste d'entiers de 0 à 9.

Exercice 7 : Fonction et Recherche dans une Liste

Écrivez une fonction **trouve_maximum(liste)** qui retourne le maximum d'une liste.

Testez votre fonction avec une liste de 10 éléments aléatoires.

Exercice 8

Ecrire un programme en Python permettant de réaliser la différence symétrique de deux listes L1 et L2, c.a.d la liste formée des éléments de L1 qui ne sont pas dans L2 et les éléments de L2 qui ne sont pas dans L1

Exemple si :

```
1 L1 = [11 , 3 , 22 , 7 , 13 , 23 , 9]
2 L2 = [5 , 9 , 19 , 23 , 22 , 23 , 13]
```

Le programme renvoie la liste

```
1 [11, 3, 7, 5, 19]
```

Exercice 9

Ecrire un algorithme en python qui renvoie le nombre d'occurrences d'un élément a dans une liste L donnée sans utiliser aucune fonction prédéfinie en Python.

Exemple si

```
1 L = [7 , 23 , 5 , 23 , 7 , 19 , 23 , 12 , 29]
```

et **a = 23**, l'algorithme renvoie 3.

Exercice 10

Ecrire un algorithme en python sous forme de fonction qui prends en paramètre une liste de nombres entiers L et qui renvoie la liste obtenue à partir de L en supprimant tous les nombres négatifs.

Exemple si

```
1 L = [7 , -2 , 11 , -25 , 16 , -3 , 14]
```

, l'algorithme renvoie la liste:

```
1 [7 , 11 , 16 , 14]
```

Exercice 10 : Fonction de Filtrage

Écrivez une fonction **filtrer_paires(liste)** qui retourne une nouvelle liste contenant uniquement les nombres pairs de la liste d'entrée.

Testez cette fonction sur une liste de nombres de 1 à 20.

