Lycée Faidherbe, 2020-2021

Cours d'informatique

MPSI & PCSI

Version du 3 juin 2021

Table des matières

I Présentation **5**
1 Qu'est-ce que l'informatique ? . . . 5
2 Soulevons le capot ! . . . 7
3 Coder l'information . . . 11
4 Les langages . . . 13

II Variables **15**
1 Variables . . . 15
2 Opérations de base . . . 17

III Fonctions **21**
1 Pourquoi des fonctions ? . . . 21
2 Des fonctions déjà écrites . . . 22
3 Définir ses propres fonctions . . . 23
4 Usage des fonctions . . . 26
5 Exercice : le jeu des erreurs . . . 28

IV Boucles simples **29**
1 Pourquoi les boucles ? . . . 29
2 Répétitions . . . 29
3 Boucles inconditionnelles . . . 31
4 Complexité : 1 . . . 33

V Les structures conditionnelles **35**
1 Les structures conditionnelles . . . 35
2 Les expressions booléennes . . . 38

VI Listes : 1 **39**
1 Introduction . . . 39
2 Définitions . . . 39
3 Opérations . . . 41
4 Modification de listes . . . 44
5 Exercice . . . 46
6 Solutions . . . 47

VII Boucles conditionnelles **49**
1 Interrompre une boucle . . . 49
2 Définition . . . 50
3 Usages . . . 51
4 Recherche dans une liste triée . . . 52

VIII Listes : 2 **57**
1 Création de liste par augmentations . . . 57
2 Autres méthodes . . . 59
3 Décomposition en base 2 . . . 59
4 Solutions . . . 60

IX Textes et fichiers **61**
1 Les chaînes de caractères . . . 61
2 Caractères (rappels) . . . 62
3 Fonctions, opérations et méthodes sur les chaînes . . . 62
4 Les fichiers . . . 63
5 Recherche naïve d'un mot dans une chaîne de caractères . . . 65
6 Solutions . . . 66

X Entiers **69**
1 Entiers non signés . . . 69
2 Entiers signés . . . 71

XI Réels **73**
1 Représentations possibles sur 8 bits . . . 73
2 Les nombres flottants dans Python . . . 76

XII Zéros de fonctions **79**
1 Méthode de dichotomie . . . 80
2 Méthode de Newton . . . 83
3 `fsolve` . . . 85

XIII Intégration numérique **87**
1 Subdivision . . . 87
2 Méthode des rectangles . . . 88
3 Méthode des trapèzes . . . 89
4 Compléments hors-programme : méthode de Simpson . . . 90
5 Solutions . . . 91

XIV Équations différentielles **93**
1 Présentation . . . 93
2 Quelques méthodes . . . 95

XV Module Numpy **101**
1 Listes Python . . . 101
2 Le module Numpy . . . 102

XVI Équations différentielles : compléments **105**
1 Équations d'ordre 2 . . . 105
2 Équations couplées . . . 108

XVII Bases de données **111**
1 Présentation . . . 111
2 Relations . . . 114
3 Algèbre relationnelle simple . . . 116
4 SQL . . . 118
5 Fonctions sur les attributs . . . 120

XVIII Bases de données composées **123**
1 Tables multiples . . . 123
2 Utilisation de plusieurs tables . . . 125
3 Exercices . . . 127
4 Compléments . . . 129
5 Solutions . . . 131

Chapitre I

Présentation

1 Qu'est-ce que l'informatique ?

La Société Informatique de France (SIF) propose la définition suivante.

Définition : informatique

L'informatique est la science et la technique de la représentation de l'information d'origine artificielle ou naturelle, ainsi que des processus algorithmiques de collecte, stockage, analyse, transformation et communication de cette information, exprimés dans des langages formels ou des langues naturelles et effectués par des machines ou des êtres humains, seuls ou collectivement.

Cette définition, très structurée, mérite des éclaircissements.

Information L'informatique à pour objet l'information. Cela peut sembler une évidence quand on regarde l'étymologie mais le nom anglais, **computer science**, ne le laissait pas deviner. Lorsque plusieurs processus ou agents interagissent dans la poursuite d'un but commun ils doivent échanger de l'information. L'informatique est ce qui permet ces échanges d'informations.

Machines L'informatique utilise des machines, les ordinateurs.

Mais l'informatique n'est pas **que** la science des ordinateurs : de la même manière que l'astronomie n'est pas que la science des télescopes. Passés les premiers temps héroïques où les prouesses nouvelles des machines nous émerveillaient, l'informatique nous permet de comprendre ce que font les ordinateurs et ce qu'ils ne peuvent pas faire. L'informatique est ce qui rend possible l'existence et l'usage des ordinateurs.

Les ordinateurs sont des machines rapides et fiables : ils permettent d'automatiser des tâches fastidieuses et répétées. En ce sens on peut en voir l'origine dans les machines automatiques : les calculatrices mécaniques, le métier à tisser Jacquard, les orgues de Barbaries, ...Mais une évolution remarquable est que les ordinateurs ne sont pas spécialisés dans un travail unique : ils sont programmables. La première tâche d'un ordinateur est de recevoir les instructions qu'il aura à exécuter.

Langages On a donc besoin d'un langage commun entre les humains et les machines : un langage informatique est le moyen de traduire nos instructions à une machine. Comme celle-ci manque cruellement d'imagination et d'anticipation nous sommes contraints de faire un travail de formalisation qui nous permet de décomposer en tâches simples le travail que nous souhaitons faire faire à l'ordinateur.

Science et technique L'informatique est une science : elle produit des énoncés qui seront évalués selon le critère de vérité. La science informatique est née au début du vingtième siècle par les résultats de mathématiciens (Turing, Gödel, Curry, ...) qui cherchaient à déterminer les calculs mathématiques qui sont **effectivement** réalisables, c'est-à-dire représentables par un algorithme avec le vocabulaire actuel.

La science informatique ne peut être séparée de la question de la réalisation effective : elle cherche à savoir ce qui peut être fait et cherche à faire ce qui peut être fait.

Autres sciences Comme son objet est l'information l'informatique est étroitement liée aux autres sciences.
Elle étudiera souvent les mêmes phénomènes que d'autres sciences mais *différemment*.
Un exemple : les images.

- Un physicien étudie les images sous forme de propagation de rayons lumineux. Cela permet la construction de verres qui corrigent la vue ou de lentilles pour les appareils photo.
- Avec la photo argentique, la chimie permet la reproduction d'images sur une surface par dépôt de pigments colorés qui se fixent sur la surface.
- Les géomètres mathématiciens s'intéressent aux formes qui constituent les images dans le plan ou dans l'espace (et généralisent ensuite dans des espaces plus généraux).
- Les médecins soignent les problèmes de l'œil et des nerfs optiques qui nous permettent de voir les images.
- L'informatique appréhende les images en les *pixélisant* elle en traduit l'information, ce qui permet de la transmettre, la reproduire, la compresser, la transformer, la caractériser ....

Les appareils photo numériques apparus il y a une vingtaine d'années sont le fruit des efforts conjoints de physiciens et d'informaticiens

1.1 Qu'est-ce qu'un ordinateur ?

C'est un truisme de dire aujourd'hui que les ordinateurs sont partout : des machines installées sur un bureau aux serveurs dans des salles climatisées, mais aussi dans les téléphones, les télévisions, les appareils photos, les liseuses, les voitures... et avec la généralisation des «objets connectés», on en trouve jusque dans les ampoules.
Tant et si bien que l'on peut ressentir une certaine confusion en tentant de préciser ce qu'est un ordinateur. On peut commencer par lister les éléments communs suivants :

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

Néanmoins cette première tentative s'avère infructueuse, car il n'est pas difficile de trouver des contre-exemples : un scooter nécessite une source d'énergie, il reçoit des informations de la part de son conducteur, il en émet sous la forme de signaux lumineux indiquant le niveau d'huile, de carburant... mais ce n'est pas un ordinateur.
En fin de compte, ce qui caractérise un ordinateur est plutôt sa capacité à *mener des calculs*. D'ailleurs, le terme *computer* est lié en Anglais au verbe *to compute* : calculer.
Nous allons partir d'un ordinateur tel qu'il se présente aujourd'hui sous sa forme la plus reconnaissable (un «ordinateur personnel») et survoler le rôle de chacun de ses composants, avant de plonger dans les détails et de revenir sur les interrogations ci-dessus.

2 Soulevons le capot !

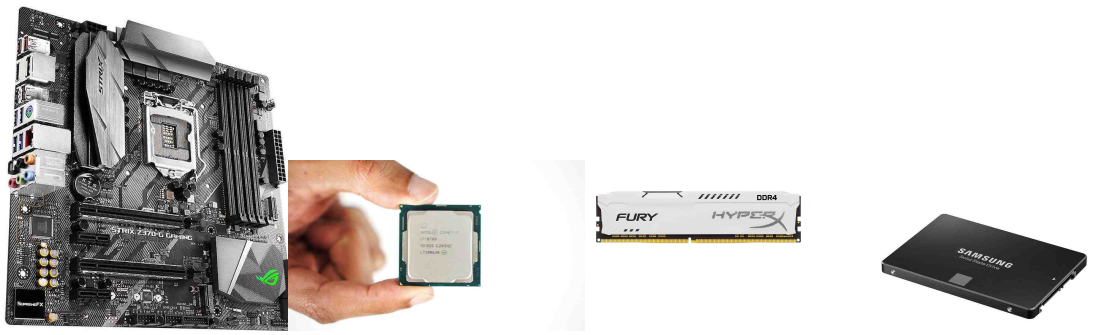
2.1 De multiples périphériques et composants

Clavier, disque dur, clef USB, carte graphique... Tout le monde a entendu ces termes, mais n'est pas forcément au fait de la manière dont ils s'articulent et communiquent pour accomplir leur fonction.
Passons sur les «périphériques externes», que tout le monde connaît au moins de vue (figures I.1a à I.1d). Leur fonction est en général plutôt intuitive (mais leur fonctionnement, lui, beaucoup moins !) Concentrons-nous la partie qui reste en général hors de vue, à l'intérieur du boîtier.

(a) **(b)** **(c)** **(d)**

Figure I.1 – a) Clavier, b) souris, c) micro-casque, d) imprimante et scanner.

Les figures I.2a à I.2g montrent des pièces détachées avec lesquelles on va pouvoir assembler un ordinateur. Il est intéressant de remarquer que, avec un peu de débrouillardise et de connaissance, il



(a) **(b)** **(c)** **(d)**

(e) **(f)** **(g)**

Figure I.2 – a) Carte-mère, b) processeur central, c) barrette de RAM, d) disque dur, e) carte graphique, f) dissipateur thermique surmonté d'un ventilateur (*ventirad*), g) alimentation.

n'est pas difficile de réaliser soi-même l'assemblage d'un ordinateur à partir des ces pièces détachées. Cela ne nécessite en fait pas vraiment de savoir *comment* fonctionne un ordinateur.
Ce fait remarquable résulte de la très grande *normalisation* des ordinateurs : chaque composant réalise des tâches bien déterminées et communique de manière pré-établie, de sorte que l'ordinateur sait précisément quoi lui demander et quoi en attendre, sans que l'utilisateur n'ait à intervenir.

2.2 Et si on les assemble ?

Les pièces ci-dessus, une fois assemblées dans un boîtier adapté, donnent l'ordinateur des figures I.3a et I.3b. Bien sûr, un tel assemblage n'a rien de compact, et un ordinateur portable ou un téléphone

n'est pas fait comme ça. Néanmoins, les éléments constitutifs sont toujours les mêmes. L'exemple traité ici a le mérite de permettre de les observer séparément. On remarque que les divers com-

(a) **(b)**

Figure I.3 – Un PC assemblé en tour : a) intérieur, b) arrière.

posants sont presque tous connectés à un même élément qui occupe toute la surface d'un flanc de la tour. C'est la *carte-mère* (figure I.2a), qui joue le rôle de *matrice de communication* : elle comporte de nombreux connecteurs sur lesquels viennent se brancher tous les autres composants et par lesquels ils pourront communiquer entre eux. On y trouve :

- des connecteurs *externes*, en haut à gauche sur la figure I.3b, que même les utilisateurs non spécialistes connaissent (USB pour brancher clavier et souris, jack pour brancher des enceintes, etc) ;
- des connecteurs *internes*, pour les composants situés à l'intérieur de la tour qui ne sont pas destinés à être branchés et débranchés fréquemment.

Les divers types de connecteurs diffèrent non seulement par leur forme, mais aussi par leurs caractéristiques physiques (électriques, mécaniques ou thermodynamiques). Certains sont optimisés pour laisser passer de gros débits de données, d'autres peuvent transporter des courants électriques intenses pour permettre la recharge d'une batterie, etc.

2.3 Le cheminement des données

Alice [1] vient de saisir un texte et elle tape le raccourci clavier déclenchant sa sauvegarde sur le disque dur de l'ordinateur. Elle s'attend donc à deux choses : d'abord que son document soit enregistré, ensuite qu'il y en ait une confirmation visuelle à l'écran. Quel est le chemin pris par les données au cours de cette opération ?

Un peu de vocabulaire

Quand un composant peut communiquer avec d'autres, il dispose d'un *contrôleur*. C'est un élément électronique qui sert d'interface avec l'extérieur et orchestre les communications.
Comme les flux de données allant d'un composant à l'autre peuvent entrer ou sortir des composants, on parle de flux d'*entrée/sortie* ou I/O (pour Input/Output).

1. Dans certains domaines de l'informatique, on a estimé peu conviviales les appellations «utilisateur A» et «utilisateur B» dans les exemples, et de là est née la tradition de les appeler Alice et Bob.

Bien que ce ne soit pas une règle absolue, on classe en général les I/O en deux catégories, les «lents» et les «rapides», qui ne sont pas gérées par les mêmes contrôleurs.
Le canal de communication entre deux composants s'appelle un *bus*.

Exemple

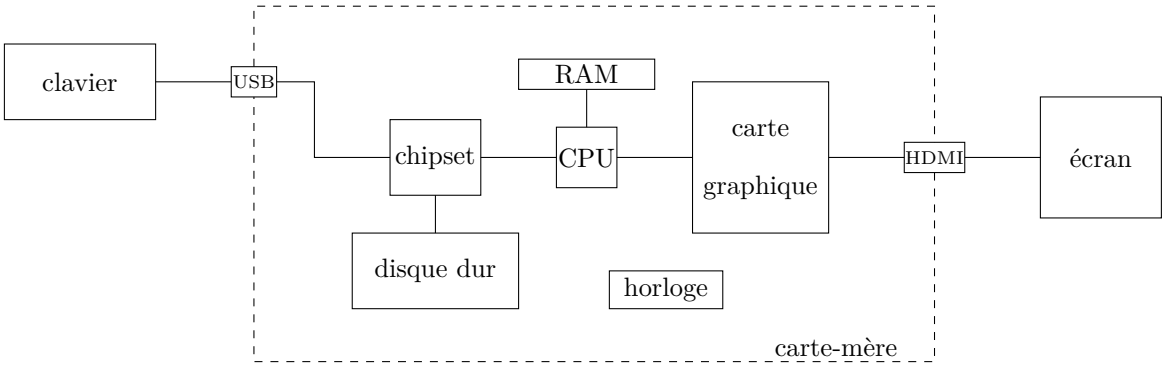


Figure I.4 – Architecture typique d'un ordinateur personnel (bien sûr, il existe des variantes).

Suivons les flux de données sur le schéma de la figure I.4.
Sur la carte-mère, les contrôleurs I/O «lents» sont regroupés dans le chipset,[2] tandis que les contrôleurs I/O «rapides» sont dans le CPU.[3]
Gardez en tête que les diverses étapes ci-dessous ne se font pas toutes seules. Le système d'exploitation intervient fortement pour traiter les données et coordonner leurs échanges.

1. L'information «une touche du clavier a été pressée» arrive donc par le connecteur (typiquement USB) sur lequel le clavier est branché.
   Ce flux est donc reçu par le contrôleur situé dans le chipset.
   Mais l'histoire ne s'arrête pas là : l'écran n'est pas relié au chipset, l'endroit où est mémorisé le document d'Alice non plus, et l'ordinateur ne comprend pas «automatiquement» ce que veulent dire les touches enfoncées par Alice...
2. L'information est envoyée vers le contrôleur du *processeur central* (CPU, représenté seul figure I.2b et caché sous le ventilateur central sur la figure I.3a).
   Une fois qu'il a reçu notre flux de données, le processeur va devoir faire des calculs pour les traiter, afin de comprendre et d'accomplir sa tâche. Mais pour cela, dans quelle partie de l'ordinateur se trouve le document tant qu'il n'a pas été enregistré ?
3. La réponse est, dans la *mémoire vive* (RAM : Random Access Memory). Composant rapide relié au processeur, cet espace de stockage est caractérisé par sa volatilité.[4] La RAM se présente physiquement sous la forme de barrettes, comme représenté figure I.2c et que l'on voit à droite du processeur sur la figure I.3a.
4. Le CPU doit maintenant orchestrer le transfert des données depuis la RAM vers le disque dur[5] (figure I.2d). Ce dernier est bien sûr un dispositif de stockage ; comparé à la RAM, il est notablement plus lent mais les données y sont *persistantes* (elles survivent à une coupure de courant). S'il est d'un modèle courant, le disque dur est connecté au contrôleur du chipset.
   Le document d'Alice est enfin enregistré !
5. Reste à mettre à jour l'affichage pour informer Alice que l'enregistrement a été effectué. Le CPU possède aussi un contrôleur relié à la *carte graphique*, représentée figure I.2e. Composant très complexe capable de mener ses propres calculs, c'est sur elle que l'écran est branché : elle s'occupe donc du traitement et de la mise en forme des données vidéo.[6]

2. Le contrôleur lui-même s'appelle le *southbridge*.
3. Collectivement appelées *northbridge*.
4. Les données n'y sont pas stockées durablement. Une coupure de courant même très brève suffit à perdre tout son contenu, comme beaucoup de gens en ont déjà fait l'amère expérience !
5. En toute rigueur, l'appellation de «disque» n'est plus vraiment appropriée, les nouveaux modèles ne comportant plus de plateau tournant, ni même la moindre pièce mobile. Il vaut mieux parler de lecteur (*drive* en Anglais).
6. Par le passé, le CPU s'en occupait directement, mais c'est une tâche intensive faisant appel à des méthodes

Notons que la plupart de ces opérations sont menées en suivant un rythme précis fourni par un composant de la carte-mère appelé *horloge*.

2.4 L'horloge

Ce composant situé sur la carte-mère (figure I.5) contient un cristal piézoélectrique. Ce matériau possède deux caractéristiques spécifiques :

- C'est un oscillateur mécanique, capable de se contracter et de se dilater à une fréquence qui lui est propre (sa *fréquence de résonance*, déterminée par sa fabrication).
- Quand il est soumis à une tension électrique, il réagit en se mettant à osciller mécaniquement. Réciproquement, quand il oscille mécaniquement, il produit une tension oscillante à ses bornes.

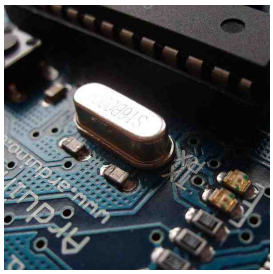


Figure I.5 – Horloge dans la carte-mère.

L'astuce est de prélever la tension qu'il produit par ses oscillations mécaniques naturelles, de l'amplifier et de la réappliquer à ses bornes, ce qui «régénère» les oscillations mécaniques. Si l'amplification est suffisante pour compenser les pertes énergétiques, les oscillations deviennent *entretenues* et le cristal devient la source d'un signal parfaitement périodique sur lequel beaucoup d'opérations se calent.

2.5 Considérations thermiques

Il est bien connu qu'un composant électronique a tendance à chauffer. Plus il est sollicité, plus il est amené à travailler vite, et les courants électriques circulant dedans produisent de la chaleur par effet Joule. Une température excessive peut perturber les calculs et, dans des cas extrêmes, endommager le composant. [7]

À titre d'exemple, le dégagement de chaleur d'un processeur de moyenne gamme en pleine charge peut approcher 100 W, avec une température atteignant les 80 °. [8] On comprend que, dans les entreprises, les salles informatiques soient thermo-régulées ! [9]

Les figures I.3a et I.3b montrent les points clefs de la stratégie de gestion des flux thermiques :

- Tous les composants dégageant de la chaleur (carte graphique, processeur central, chipset) sont coiffés d'une pièce métallique appelée *dissipateur thermique* (figure I.2f). Très bon conducteur thermique et taillé en multiples ailettes pour maximiser sa surface de contact avec l'air ambiant, il conduit la chaleur vers un ventilateur qui l'évacue vers l'intérieur de l'ordinateur.
- La tour elle-même est munie d'un ou plusieurs ventilateurs qui pompent la chaleur dégagée par les ventilateurs précédents et l'éjectent à l'extérieur du boîtier.
- L'ordinateur est branché sur le secteur et reçoit donc du 220 V alternatif, alors que les composants nécessitent une tension bien plus faible et continue. Il faut donc une *alimentation* (figure I.2g) qui va s'occuper de la conversion. Cela génère de la chaleur, ainsi qu'un rayonnement électromagnétique capable de perturber son environnement dans un rayon de quelques centimètres. L'alimentation est donc en général éloignée du reste de l'ordinateur (on la voit tout en bas de la figure I.3b).

de calcul très spécifiques, de sorte qu'on préfère en général la déporter vers une unité de calcul dédiée, la *carte graphique*.

7. Heureusement, beaucoup de composants sont munis de sondes thermiques pour se désactiver automatiquement en cas de surchauffe.

8. Les processeurs destinés à des machines intégrées, comme des ordinateurs portables ou des smartphones, sont conçus pour dégager beaucoup moins, au prix souvent de performances réduites.

9. Et si l'ordinateur est en plus muni d'une carte graphique puissante, le dégagement thermique peut être doublé, voire triplé.

3 Coder l'information

3.1 Exprimer une quantité d'information

L'unité élémentaire d'information est le bit (binary digit, chiffre binaire). Il n'a donc que deux valeurs possibles, 0 ou 1.
Un groupement de $8 = 2^3$ bits s'appelle un *octet* (*byte* en Anglais), de symbole B. [10]
On peut définir des multiples de l'octets comme pour n'importe quelle unité (par des puissances de 10), mais comme en informatique on travaille toujours en base 2, on peut aussi les définir par des puissances de 2. Remarquant que $2^{10} = 1024$:

Unité (base 2)	Valeur	Unité (base 10)	Valeur
kibi-octet (kiB)	1024 B	kilo-octet (kB)	1000 B
mébi-octet (MiB)	1024 kiB	méga-octet (MB)	1000 kB
gibi-octet (GiB)	1024 MiB	giga-octet (GB)	1000 MB
tébi-octet (TiB)	1024 GiB	téra-octet (TB)	1000 GB

Enfin, attention aux confusions de la langue courante. Par écrit les deux systèmes de notation sont globalement respectés (en tout cas dans le monde professionnel), à l'oral on a tendance à utiliser les appellations de la base 10 même quand on parle de la base 2. Donc prêtez attention au contexte !

3.2 Codage des valeurs

La manière dont est traduite une valeur en bits dépend de la nature de la variable, son **type**.
Le nombre de bits alloués gouverne le nombre maximal de valeurs différentes qui peuvent être utilisées. Les valeurs les plus courantes sont

Espace alloué	Nombre de valeurs différentes
8 bits	$2^8 = 256$
10 bits	$2^{10} = 10\,24$
12 bits	$2^{12} = 40\,96$
16 bits	$2^{16} = 65\,536$
24 bits	$2^{24} = 16\,777\,216$
32 bits	$2^{32} = 4\,294\,967\,296$
64 bits	$2^{64} = 18\,446\,744\,073\,709\,551\,616$

Voici quelques utilisations typiques :

- 8 bits : codage d'une couleur dans les formats d'images et de vidéos (JPEG, Blu-ray ...)
- 10 bits : code d'une couleur dans certains formats d'images professionnels.
- 16 bits : codage des sons dans les formats audio grand public (CD, MP3 ...)
- 24 bits : codage des sons dans les formats audio professionnels (mastering).
- 32 bits : codage des entiers dans la plupart des langages de programmation.
- 64 bits : codage des flottants dans la plupart des langages de programmation.

Les types simples les plus utilisés sont les entiers, les flottants et les caractères.

Entiers

En première approche, on peut dire que les entiers sont stockés sous forme binaire dans l'espace mémoire alloué : l'entier n est représenté sur p bits par la suite $(a_{p-1}, a_{p-2}, \ldots, a_1, a_0)$ de valeurs 0 ou 1 telle que $n = \sum_{k=0}^{p-1} a_k 2^k$ [11].
Il reste le problème de la représentation des entiers négatifs, nous y reviendront dans un chapitre ultérieur.
Cela limite les entiers à un intervalle de taille 2^p. Cependant, en théorie, Python est capable d'utiliser les entiers sans limite de valeur.

10. En France, on trouve souvent la notation o à la place de B, mais elle n'est pas reconnue internationalement.
11. C'est la représentation **big endian** ou **petitboutienne**, d'autres ordres des indices existent.

Flottants

La notion mathématique de réel n'a pas d'équivalent en informatique. En effet un nombre réel nécessite souvent un processus infini pour être défini (suite de ses décimales, suite qui tend vers le réel, etc) qu'on ne peut pas contenir dans une mémoire finie. On devra se contenter d'approximations.
Les valeurs approchant les nombres réels sont appelés *nombres flottants* pour rappeler que leur virgule «flotte», leur forme est semblable à la notation scientifique utilisée dans les sciences, par exemple $6,0210^23$. Si on suppose que l'on dispose de 8 chiffres significatifs le nombre précédent s'écrit $\frac{60200000}{10^7}10^{30}$, il peut être représenté par les entiers 60200000 et 30.
En Python, ces valeurs ont le type `float`. On les écrit avec un point à la place de notre virgule ; on peut aussi utiliser un exposant `e` qui signifie puissance 10 ; `898547e-5` signifie `8.98547`
Leur codage en mémoire utilise deux entiers, comme l'exemple ci-dessus que nous étudierons dans un chapitre ultérieur.

Caractères

De même chaque caractère est associé à un nombre entier que l'on peut coder dans la mémoire.
Il faut donc une table de correspondance entre les caractères et leurs représentations entières. Cela s'appelle un *encodage*.
L'encodage ASCII Le plus ancien encodage encore utilisé, l'ASCII 7 bits, ne contient que les caractères utilisés en Anglais. ASCII veut dire American Standard Code for Information Interchange : Code américain normalisé pour l'échange d'information.
Il contient $128 = 2^7$ caractères, donc chaque caractère peut être représenté par un entier sur 7 bits.
En pratique, l'unité élémentaire d'information en mémoire étant l'octet, un caractère se voit donc plutôt allouer 8 bits, le 8e bit étant alors laissé à 0.
On y trouve les 26 lettres de l'alphabet latin en majuscules et minuscules, les principaux symboles de ponctuation et les chiffres arabes. Quelques exemples :

Caractère	Code	Caractère	Code
A	65	0	48
B	66	1	49
a	97	(	40

Il y a aussi des «caractères spéciaux» qui n'ont pas d'apparence mais sont utiles par exemple pour structurer une chaîne de caractères :

Caractère	Code	Représentation en Python
Nouvelle ligne	10	`\n`
Tabulation	9	`\t`

Par contre, l'ASCII 7 bits ne contient pas de caractères accentués, de lettres grecques, etc. [12] Du coup, dans la deuxième moitié du 20e siècle, chaque pays a utilisé le 8e bit pour étendre l'ASCII et y ajouter les caractères nécessaires pour sa ou ses langues.
Mais ces efforts n'ont pas été normalisés, conduisant à de multiples encodages variant selon les pays et les types d'ordinateurs, générant un casse-tête d'interopérabilité.
L'Unicode Il a fallu attendre les années 2000 pour voir se concrétiser un standard international recouvrant la grande majorité des besoins de toutes les langues : l'Unicode. De nos jours, l'Unicode est l'encodage par défaut de la majorité des ordinateurs usuels.
Tous les langages de programmation modernes savent manipuler des caractères en Unicode. [13]
Dans la variante UTF-8 de l'Unicode (la plus répandue), l'espace mémoire alloué à un caractère est variable, allant de 1 à 4 octets.
La distinction se fait sur les premiers bits :

- Si le premier bit est 0, alors le caractère occupe un octet. Il reste 7 bits pour le caractère lui-même, qui est alors encodé suivant la table ASCII 7 bits.

12. Ces limitations se retrouvent encore aujourd'hui, par exemple, dans certains communications par internet : il est bien connu qu'utiliser des lettres accentuées dans des mails ou des noms de fichier quand on communique avec des gens d'autres pays peut parfois créer des problèmes.
13. Python, depuis sa version 3, travaille nativement en Unicode.

- Si les trois premiers bits sont 110, alors il occupe 2 octets. L'octet suivant doit alors commencer par 10 (bits de continuité), ce qui laisse un total de 11 bits pour le caractère. Cette plage contient les caractères latins absents de l'Anglais, mais aussi d'autres alphabets (grec, hébreu, cyrillique, arabe...) Exemples :

Caractère	Code
¿	11000010 10111111
É	11000011 10001001
ψ	11001111 10001000

- Si les quatre premiers bits sont 1110, alors il occupe 3 octets. Retirant les bits de continuité, cela laisse 16 bits pour le caractère. On y trouve encore d'autres alphabets, le Braille, des écritures idéographiques comme le Chinois ou le Japonais, mais aussi les célèbres caractères Dingbats, des symboles de ponctuation et des symboles mathématiques. Exemples :

Caractère	Code
$\oint$	11100010 10001000 10101110

- Si les cinq premiers bits sont 11110, alors il occupe 4 octets, ce qui laisse 21 bits pour le caractère. On y trouve des langues anciennes, les notations musicales, et encore plus de symboles, en particulier les célèbres emoji. [14]

En fin de compte, la représentation Unicode permet d'inclure des *millions* de caractères différents. La majorité des valeurs binaires correspondantes ne sont d'ailleurs pas encore utilisées à ce jour.
Cette variabilité de l'espace mémoire alloué peut compliquer les traitements informatiques, mais elle permet de garder la compatibilité avec l'ASCII tout en optimisant l'espace mémoire occupé par une chaîne de caractères (inutile de prendre 4 octets par caractère si on n'écrit qu'en Français !)

4 Les langages

Double-cliquer sur une icône et déclencher ainsi le visionnage d'un film (par exemple) est une forme de programmation de très haut niveau. Le langage utilisé, à base d'icônes et de menus est l'aboutissement d'une évolution rapide de ces dernières décennies.
Si l'on veut faire exécuter à un ordinateur des instructions plus spécialisées on va utiliser un langage de programmation ; on écrit un texte qui va être traduit à l'ordinateur.

4.1 Programmer un calcul

Quand on programme un calcul à l'aide d'un langage de programmation «haut niveau» (par exemple Python), il n'est pas nécessaire de savoir quelles opérations sont câblées dans le processeur. En contrepartie, à un moment ou à un autre, [15] il est nécessaire de convertir le programme écrit par l'humain en un programme adapté aux caractéristiques de l'ordinateur.
Au bout du compte, un tel programme est nécessairement écrit en binaire. Mais il existe un langage «intermédiaire», qui nécessite de connaître les caractéristiques de l'ordinateur tout en restant (relativement) lisible par l'humain. C'est l'*assembleur*. [16]
Ci-dessous se trouve un court exemple d'addition programmée en assembleur x86_64, qui est l'assembleur des processeurs 64 bits placés dans la grande majorité des ordinateurs domestiques actuels. Un tel processeur possède des registres d'une taille de 64 bits dont le nom commence toujours par un `r`.
On peut donc avoir un contrôle très fin sur ce qui se passe : en spécifiant des registres, on s'assure que la mémoire la plus rapide possible est utilisée, et que tout le calcul est mené sur le même cœur. Mais ce code paraît bien abscons pour faire juste $1 + 3$! Rappelons que, dans l'immense majorité des cas, l'assembleur n'est qu'un intermédiaire dans la conversion d'un programme vers une forme

14. On passe ici sous silence le mécanisme de *combinaison* de caractères en Unicode, nécessaire pour les emoji.
15. Selon les langages, cela se passe typiquement juste après l'écriture du programme (compilation) ou au moment de son exécution (interprétation).
16. Comme l'assembleur est directement lié aux caractéristiques du processeur, il s'agit plutôt d'une famille de langages.

compréhensible par l'ordinateur, et qu'il n'est alors pas destiné à être lu (et encore moins écrit) par un humain.
Notez aussi que, contrairement aux usages de la programmation habituelle, ce petit programme n'utilise aucune variable. À la place, il fait référence aux noms des zones mémoire (ici les registres) où se trouvent les valeurs à manipuler.

```
mov %rax, 3      ; place la valeur 3 (en base 10)
                 ; dans le registre rax
mov %rbx, 1      ; place la valeur 1 (en base 10)
                 ; dans le registre rbx
add %rax, %rbx   ; additionne les valeurs stockées
                 ; dans rax et rbx de sorte que
                 : le résultat soit dans rax
```

4.2 Évolution des langages

Les langages ajoutent une nouvelle interface entre les idées du programmeur et leur exécution par l'ordinateur.
Voici quelques étapes de l'apparition des nouveaux langages.

- Années 50 : langages spécialisés FORTRAN, LISP, COBOL
- 1958 : définition de ALGOL, qui donnera l'impulsion des langages universels
- 1967 : simula-67 premier langage orienté objet
- 1972 : C, langage défini en parallèle avec le système UNIX
- 1972 : PROLOG, programmation logique
- 1973 : ML, programmation fonctionnelle (évolution de lisp)

C'est le langage C qui a le plus inspiré les langages utilisés ensuite : c'est un langage impératif de bas niveau qui se traduit très facilement en langage machine tout en offrant un système de types. Les langages successifs améliorent en terme de puissance d'expression et de facilité d'écriture les langages antérieurs.
Les avantages des langages modernes sont

- la lisibilité du code
- sa concision
- son indépendance par rapport au processeur.

4.3 Propriétés de Python

Nous allons apprendre les concepts importants de la programmation avec le langage Python.

1. C'est un langage Open Source, disponible gratuitement sur la grande majorité des architectures,
2. qui s'appuie sur une importante communauté de développeurs.
3. Il contient de nombreux outils sous forme de modules qui sont intégrés (Batteries Included).
4. Plusieurs milliers de packages supplémentaires sont disponibles dans tous les domaines.
5. C'est un langage moderne, rapide dans le cycle écriture-test (interprété) et simple (indentation significative).
6. L'interpréteur Python est écrit en C, de même que certaines bibliothèques (d'autres sont écrites en Fortran).
7. On peut lui reprocher parfois une lenteur lors de l'exécution de programmes importants, ce n'est pas un langage compilé. Lors du développement de projets industriels on peut être amené à ré-écrire le code (ou une partie du code) dans un autre langage.
8. Cependant c'est un langage bien adapté à l'écriture de projets, il est devenu le standard dans les laboratoires pour écrire simplement des programmes personnalisés.

Chapitre II

Variables

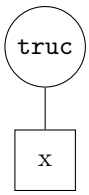
Résumé

Dans ce chapitres nous allons introduire l'affectation. Cette instruction est très importante car elle permet de dépasser la simple machine à calculer : un ordinateur calcule, certes, mais surtout garde en mémoire le résultat calculé. L'affectation établit la connexion entre les **expressions**, *c'est-à-dire les calculs et les* **variables**. *Ces deux notions sont interconnectées :*

- *les variables mémorisent le résultat d'une expression*
- *les expressions utilisent les valeurs des variables.*

1 Variables

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses, mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de bits (symboles que l'on représente par 0 ou 1) qui est stockée dans la mémoire. Pour pouvoir accéder aux données, il suffit de se souvenir à quelle adresse est stockée la valeur de la donnée. Cependant il est plus simple d'associer un nom à une adresse ce qui revient, pour l'utilisateur, à associer une valeur à un nom.
L'association d'une valeur à un nom se fait par l'affectation.
Ceci se note, en python, `nom_de_variable = x`.
On sera vigilant sur le fait que le `=` de Python n'est pas l'égalité classique ;
`x = 4` devrait se lire *la variable* `x` *prend la valeur 4* et non pas *x égale 4*.
D'autres langages on une écriture plus explicite :
`x := 4` ou `x <- 4`
On remarque de plus que ce signe = **n'est pas symétrique** :
`x = 2` a un sens mais `2 = x` n'en a pas,
`a = b` donne à la variable a la valeur de b, tandis que `b = a` modifie la valeur de b sans changer celle de a.



1.1 Noms de variables

Pour l'utilisateur une variable est d'abord un nom : les noms de variables sont des noms que vous choisissez vous-même assez librement.
La majorité des langages ont les mêmes règles pour la création des noms de variables.

- Un nom de variable est une séquence de lettres (a . . .z , A . . .Z), de chiffres (0 . . .9) et du caractère _ (underscore). Elle doit toujours commencer par une lettre.
- La casse est significative (les caractères majuscules et minuscules sont distingués).
- Il existe une liste de mots clés réservés qui ne peuvent pas être employés comme noms de variables. Dans le cas de Python il s'agit de

and	as	assert	break	class	continue	def
del	elif	else	except	False	finally	for
from	global	if	import	in	is	lambda
None	nonlocal	not	or	pass	raise	return
True	try	while	with	yield		

Il y a, de plus, quelques usages qu'il est recommandé de respecter.

- Choisissez de préférence des noms aussi explicites que possible, de manière à exprimer clairement ce que la variable est sensée contenir.
  Par exemple si on a besoin d'un majorant, d'un minorant et d'une moyenne il vaut mieux éviter des les appeler `m1`, `m2` et `m3`, on ne saura pas vraiment à quoi ils correspondent. Il vaudrait mieux utiliser simplement `majorant`, `minorant` et `moyenne`.
  Par contre `majorant_des_termes_de_la_suite` est sans doute un peu excessif.
- Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre).
- Les majuscules ou le symbole _ permettent de découper le nom :
  `premierTerme` ou `premier_terme`.

1.2 Expressions

La valeur que prend une variable peut être donnée explicitement : `g = 9.81`
Le plus souvent ce sera le résultat d'un calcul, c'est la valeur d'une expression : `somme = 2 + 3`.

Définition :
Une expression est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur.

L'affectation se fait en 3 étapes successives.

- L'expression est évaluée, le résultat est stocké à une adresse `ad`.
- Un nouveau nom de variable est défini (même s'il existait déjà avant)
- `ad` est liée au nom de la variable.

Une expression peut utiliser des variables déjà définies : lors de l'affectation la valeur **à l'instant du calcul** de la variable est utilisée, le résultat ne sera pas mis à jour si la variable est associée plus tard à une autre valeur. Par exemple l'instruction `a = 2 + b` :

1. cherche la valeur de `b`, par exemple 3,
2. calcule 2 + b, ce qui donne l'entier 5,
3. réserve de la place en mémoire pour un entier,
4. y stocke l'entier 5,
5. associe le nom de variable `a` à cette adresse.

La temporalité permet d'écrire des instructions de la forme `i = i + 1` : l'ancienne valeur de `i` est lue, on lui ajoute 1 et on affecte cette nouvelle valeur à la variable de nom `i`. L'ancienne valeur est perdue.

1.3 Une bonne pratique

On peut s'étonner du premier exemple donné ci-dessus : `g = 9.81`. Pourquoi créer une variable pour une valeur fixée ? En fait il est recommandé de systématiquement affecter à une variable les valeurs numériques que l'on emploie. La avantages sont multiples.

- **La lisibilité** : dans les expression, les valeurs numériques ne donnent pas le sens de ce qu'elles représentent, une variable au nom bien choisi aide à la compréhension.
- **La cohérence** : quand une même valeur est utilisée plusieurs fois, il est important que la valeur écrite soit la même à chaque occurrence, nous sommes capables de nous tromper.
- **La variabilité** : si on veut modifier la valeur utilisée, modifier la variable associée est beaucoup plus simple que de repérer toutes ses apparitions et de les changer.

2 Opérations de base

2.1 Entiers

Les opérations sur les entier sont, en Python,

- +, -, $*$ l'addition, la soustraction et la multiplication
- $**$ l'exponentiation `n**p` donne n^p, elle peut aussi s'écrire `pow(n, p)`,
- // et % donnent respectivement le quotient et le reste de la division euclidienne, on les obtient aussi par la fonction `divmod`,
- `abs` renvoie la valeur absolue.

Voir quelques exemples.

```
>>> 2+3
5
>>> 3*14
42
>>> 3**5
243
>>> 18//7
2
>>> 18%4
2
>>> divmod(7,4)
(1, 3)
```

2.2 Flottants

Les flottants supportent les mêmes opérations que les entiers avec la division en plus. Si on mélange entiers et flottants le résultat sera un flottant, lors d'une division de deux entiers le résultat est un flottant.
Les résultat de `4/2` et de `4//2` peuvent, dans certains contextes, ne pas être interchangeables !
L'import du module **`math`** ajoute un grand nombre d'opérations mathématiques usuelles.

```
>>> import math
>>> 2 + 3.5
5.5
>>> 7/3
2.333333333
>>> int(4.3)
4
>>> int(-3.2)
-3
>>> float(5)
>>> 2.3**1.4
3.2093639532679714
>>> math.sin(math.pi/4)
0.7071067811865475
>>> math.log(2.7)
0.9932517730102834
```

Il est possible de convertir une variable flottante en un entier avec la fonction `int` qui retire la partie fractionnaire ou la fonction `round` qui calcule l'entier le plus proche.
La fonction `float` convertit un entier en un flottant (en conservant la valeur).
La fonction partie entière (`floor`) est définie dans le module `math` ainsi que la partie entière supérieure (`ceil`). Il y a donc 4 manières différentes de convertir un flottant en entier.

Table II.1 – Conversions entières

x	int(x)	round(x)	floor(x)	ceil(x)
4.0	4	4	4	4
4.26	4	4	4	5
4.83	4	5	4	5
3.5	3	3	3	4
-2.35	-2	-2	-3	-2
-7.97	-7	-8	-8	-7
-2.5	-2	-2	-3	-2

2.3 Booléens

Le résultat d'une comparaison est un `Booléen` : `False` ou `True`, c'est-à-dire vrai ou faux, une erreur classique est d'oublier la majuscule initiale.

- Les tests d'égalité, noté `==`, ou d'inégalité, noté `!=`, donnent un résultat booléen.
- Les comparaisons d'entiers ou de flottants ont pour résultat des valeurs booléennes. Les opérateurs sont notés `<, >, <=, >=`
- On dispose aussi des opérateurs logiques usuels : `and, or, not` qui permettent de combiner les résultats de plusieurs opérations de comparaison.

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

```
>>> (5+7) == 12
True
>>> (5+7) != 13
True
>>> 6 > 8
False
>>> 6 <= 8 and 5 < 3
False
```

N.B. Les flottants ont une précision limitée. Une conséquence est qu'un test de nullité d'un flottant ne donne que rarement le résultat `True` même si, mathématiquement, la variable devrait avoir la valeur nulle.
Nous utiliserons souvent des variables booléennes, par exemple pour suivre la valeur de vérité d'une propriété qui doit être vérifiée pour un grand nombre d'éléments.

2.4 Chaînes de caractères

Dans Python, les caractères usuels sont associées à un entier entre 0 et 255 selon un encodage ASCII sur 8 bits avec les fonctions

- `ord`, qui donne le code d'un caractère,
- `chr`, qui donne le caractère associé à un entier.

Les caractères seront utilisés le plus souvent sous la forme d'une assemblage de caractères, la **chaîne de caractères** de type `str`.

On définit une chaîne en en écrivant les caractères entourés d'apostrophes simples, `nom = 'Jean Moulin'`, ou doubles, `nom = "Raymond Aubrac"`.
Python permet de convertir les nombres en chaînes de caractères à l'aide de la fonction `str`.

```
>>> a = 1/33
>>> b = 257
>>> str(a)
'0.3333333333333333'
>>>str(b)
'257'
```

Si une chaîne représente un réel (ou un entier) directement (sans opération) on peut la convertir avec la fonction nommée selon le type.

```
>>> float("3.14159")
3.14159
>>> int("254")
254
>>> float("254")
254.0
```

2.5 Affichage

`print` permet d'afficher des chaînes de caractères ou les valeurs des expressions à l'écran
Ce n'est pas une copie directe du contenu d'une variable.

```
>>> print("Tom Morel")
Tom Morel
>>> print(2+3)
5
```

On peut utiliser des caractères spéciaux dans une chaînes de caractères, ils seront interprétés par la fonctions `print`.

- `\'` est remplacé par une apostrophe,
- `\"` est remplacé par une apostrophe double,
- `\n` est remplacé par un retour à la ligne,
- `\t` est remplacé par une tabulation ...

Ces caractères spéciaux sont signifiés à l'aide de deux signes mais sont considérés chacun comme un seul caractère.

```
>>> a = 'Sans liberté de blâmer\nIl n\'est d\'éloge flatteur'
>>> print(a)
```

```
Sans liberté de blâmerIl n'est d'éloge flatteur
```

On pouvait ici éviter les `\'` en écrivant
`"Sans liberté de blâmer\n Il n'est d'éloge flatteur"`
Si on envoie plusieurs paramètres à imprimer, séparés par une espace.

```
>>> x = 3
>>> y = 'fois'
```

```
>>> z = 5
>>> print(x, y, z)
3 fois 5
```

2.6 Lecture

Un programme Python peut lire une variable.
L'instruction `arg = input(ch)`

- affiche la chaîne de caractères `ch` ; il est utile quel la chaîne indique que le programme attend une entrée au clavier,
- attend que l'on saisisse une chaîne de caractère au clavier suivie de l'appui de la touche **return**,
- puis affecte cette chaîne à la variable `arg`.

Le résultat est une chaîne de caractères ; on peut le vérifier avec la fonction `type` qui renvoie le type de données correspondant à une variable. Pour convertir le résultat reçu en nombre il faut convertir la chaîne à l'aide des fonctions `int` ou `float`.

Programme II.1 – Calcul du prix TTC

```
#Programme de calcul du prix TTC
prix_ht = input ("Quel est le prix hors taxes ? \n")
print(type(prix_ht))
prix_ht=float(prix_ht)
print(type(prix_ht))
prix_ttc=prix_ht+prix_ht*20/100
print("Le prix TTC est {:.2f} euros".format(prix_ttc)
-----------------------------------
Quel est le prix hors taxes ?
145.5
<class 'str'>
<class 'float'>
Le prix TTC est 174.60 euros
```

Nous n'utiliserons que rarement la fonction `input` : en effet le but sera de construire des programmes qui passeront directement les paramètres entre les différentes parties.

Chapitre III

Fonctions

Résumé *Dans ce chapitre nous allons apprendre l'écriture des fonctions et leur usage. L'écriture de fonctions sera l'activité principale pendant l'enseignement de l'informatique.*

1 Pourquoi des fonctions ?

Readability counts

Écrire un code nécessite deux impératifs quant à sa lecture :

- il doit être lisible par l'ordinateur, sa syntaxe doit être correcte,
- il doit être lisible par un programmeur afin de pouvoir le corriger, l'améliorer ...

Un des meilleurs moyens pour rendre le code humainement lisible est d'utiliser des fonctions.

Modularité

On sépare le programme à écrire en plusieurs parties sous forme de fonctions, on écrit ainsi des parties indépendantes que l'on va pouvoir assembler. On peut répéter ce procédé, une fonction peut elle-même se décomposer en plusieurs parties, qui seront écrites sous forme de fonctions qui elles aussi peuvent se décomposer ...

Mise en commun

Il arrivera souvent que l'on répète le même calcul à plusieurs endroits. L'usage des fonctions permet alors de n'écrire qu'une fois le code et l'utiliser à chaque fois que nécessaire. On applique ici le principe DRY `Don't Repeat Yourself` : ne pas se répéter.

Bien entendu l'écriture et l'usage des fonctions doivent se conformer à des règles strictes : une fonction, lorsqu'elle est écrite, correspond à une "boîte noire" qui reçoit des paramètres (les variables de la fonction) bien déterminés, dans un ordre fixé et qui renvoie le résultat.
Par exemple la fonction **sinus** reçoit un angle α dont on doit fixer l'unité, radian ou degré, et renvoie $\sin(\alpha)$. Cette fonction fait des calculs mais nous ne savons pas lesquels lors de son utilisation.

2 Des fonctions déjà écrites

2.1 Fonctions primitives

Python, comme tout langage, contient des fonctions de base, en voici la liste. On y reconnaît

Table III.1 – Fonctions de base de Python

abs	all	any	ascii	bin	bool	breakpoint
bytearray	bytes	callable	chr	@classmethod	compile	complex
delattr	dict	dir	divmod	enumerate	eval	exec
filter	float	format	frozenset	getattr	globals	hasattr
hash	help	hex	id	input	int	isinstance
issubclass	iter	len	list	locals	map	max
memoryview	min	next	object	oct	open	ord
pow	print	property	range	repr	reversed	round
set	setattr	slice	sorted	@staticmethod	str	sum
super	tuple	type	vars	zip	___import___	

- des fonctions mathématiques : `abs, divmod, max, min, pow`,
- les nombreuses fonctions de conversion : `bin, bool, chr, complex, float, hex, int, list, oct, ord, round, set, str, tuple`,
- `bin, hex, oct` donnent les écritures des entiers en base 2, 16 et 8 sous forme de chaînes de caractères précédées respectivement de `'0b` `'0x` et `'0o`,
- des fonctions d'entrée-sortie [1] : `input, open, print`.

Nous utiliserons lors de l'étude des listes les fonctions `len` et `range`.
Les autres fonctions ne seront pas utilisées.

2.2 Fonctions de bibliothèques : Batteries Included Philosophy

On voit qu'il y a peu de fonctions. Cependant python est fourni avec un grand nombre de fonctions regroupées par thèmes dans des **modules**.
Ces modules ne sont pas inclus au démarrage mais on peut charger ceux dont on a besoin très simplement.
On a déjà utilisé le module typemath qui contient les fonctions mathématiques et des constantes.
Si on veut utiliser les fonctions d'un module, il y a plusieurs méthodes que nous allons exposer pour le module `math`.

- `import math`. C'est la méthode de base, on accède aux fonctions en préfixant par le nom du module, `math.sin(math.pi/6)`
- `import math as m`. C'est la méthode employée pour écrire un alias plus court (ici `m`) à la place du nom complet du module : on accède aux fonctions du module en préfixant par cet aliax, `m.cos(0.5)`
- `from math import pi, tan`. On peut n'importer que quelques fonctions : dans ce cas les noms sont accessibles directement, `tan(pi/4`.
- `from math import *`. On importe de nouveaux toutes les fonctions du module mais ici les fonctions sont nommées directement : `exp(1.5) - e`. Bien que séduisante cette méthode ne sera pas toujours recommandée, en effet elle masque l'origine des fonction et peut induire des confusions quand un même nom de fonction est utilisé dans deux modules.

Bien que très complète, la collection fournie par python n'offre pas les fonctions très spécialisées. Nous emploierons dans l'année 3 bibliothèques scientifiques : `numpy`, `matplotlib` et `scipy`. Il faudra les ajouter (c'est déjà fait dans la distribution Anaconda).

1. La fonction `open` sera étudiée plus tard.

3 Définir ses propres fonctions

3.1 Un exemple

On considère une fonction mathématique simple, $f : x \mapsto \dfrac{x^2+3x+2}{1+x^2}$.
En Python on va retrouver les mêmes éléments.

1. La déclaration du nom de la fonction, ici f
2. le nom choisi pour la variable (ou les noms), ici x
3. les calculs qui permettent de trouver le résultat souhaité.

```
def f(x):
    """Entrée : une variable flottante
       Sortie : la valeur de la fonction en ce point"""
    numerateur = x**2 + 3*x + 2
    denominateur = 1 + x**2
    y = numerateur/denominateur
    return y
```

on peut alors utiliser la fonction directement

```
>>> f(2)
8.0
>>> f(5)
5.5
>>> a = 2
>>> b = f(a)
>>> b
8.0
```

ou par des appels dans l'éditeur, il conviendra d'utiliser la fonction `print` afin d'afficher le résultat.

```
print(f(5))
a = 2
b = f(a)
print(b)
-----
5.5
8
```

3.2 Structure

Une fonction est définie par :

1. une première ligne de déclaration composée
   (a) du mot-clé **def**,
   (b) du nom choisi pour la fonction
   (c) de la liste des variables utilisées, entre parenthèses,
   les parenthèses doivent être écrites, même s'il n'y a pas de variable,
   (d) du symbole **:**
2. des instructions permettant le calcul attendu, dans un bloc indenté (de 4 espaces)
3. d'une ligne facultative, indentée elle aussi, commençant par **return** pour donner la valeur (ou les valeurs) que la fonction doit renvoyer.

Afin de savoir quel est le rôle d'une fonction, dans quel ordre il faut rentrer les paramètres ou quelles sont les valeurs renvoyées, il est important de documenter sa fonction. Pour cela on peut écrire une documentation de la fonction : le **docstring**. Voici un autre exemple.

```
def hypotenuse(a, b):
    """Entrees : 2 nombres, les cotes d'un triangle rectangle
        Sortie : l'hypotenuse du triangle """
    c = (a**2 + b**2)**0.5
    return c
```

Le docstring s'écrit dans le corps de la fonction, juste après la première ligne de déclaration. La documentation commence par `"""` et se termine par `"""`. Cette documentation doit donc indiquer :

- la liste des paramètres attendus dans l'ordre,
- la ou les valeurs renvoyées s'il y en a.

Il est possible de faire appel à cette documentation dans une console en tapant `help(nom_fonction)`.

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

def hypotenuse(a, b):
    Entrees : 2 nombres, les cotes d'un triangle rectangle
    Sortie : l'hypotenuse du triangle
```

Cette documentation n'est pas obligatoire, Python saura utiliser la fonction. Mais elle a pour rôle de permettre à un utilisateur de mettre en œuvre une fonction dont il ne connaît pas la structure interne, à ce titre elle est indispensable.

3.3 Documentation d'une fonction pour les programmeurs

Lorsque l'on écrit une fonction il faut prévoir qu'elle sera relue, modifiée, corrigée.
À ce moment le lecteur (ce peut être l'auteur original) devra comprendre les idées de l'auteur, c'est souvent très difficile.
Il est donc recommandé de "commenter" la fonction en indiquant les étapes intermédiaires, les significations des variables, les astuces utilisées.
Pour cela on peut écrire des phrases humainement lisibles après le caractère `#`, tout ce qui suit sera ignoré par python mais sera lisible par celui ou celle qui lira le code.
Voici, par exemple, un code produit dans un TIPE, il n'y a malheureusement pas de docstring. Les éléments utilisés seront expliqués dans les chapitres suivants.

```
def premiersZeros(nb,n):
    h = 0.1           # Pas pour la recherche
    epsilon = 1e-10   # Précision pour les zéros,
                      # on peut la diminuer
    zeros = []        # Liste pour recevoir les zéros
    a = h             # On commence à h, pas en 0,
                      # car jn(n,0) = 0
    def f(x):           # On définit la fonction que l'on
        return jn(n,x) # utilise, ici Bessel à l'ordre n
    for i in range(nb):  # On veut nb zéros
        while f(a)*f(a+h) > 0:      # On balaye à la recherche
            a  = a + h              # d'un changement de signe
        z = dicho(f,a,a+h,epsilon)  # On cherche la racine
        zeros.append(z)             # On l'ajoute à la liste
        a = a + h                   # On part un cran plus
           loin
    return zeros
```

Plus la fonction est longue plus il sera nécessaire de la commenter, il arrivera fréquemment que les commentaires prennent plus de place que le code.

4 Usage des fonctions

La traduction d'un programme s'effectuant ligne par ligne, la définition d'une fonction doit s'effectuer en amont de son usage. Ainsi pour notre fonction hypoténuse, son utilisation peut s'écrire dans l'éditeur mais seulement après l'écriture de la fonction.

```
def hypotenuse(a, b):
    """Entrees : 2 nombres, les cotes d un triangle rectangle
      Sortie : l hypotenuse du triangle """
    c = (a**2 + b**2)**0.5
    return c

valeur_hyp=hypothenuse(3 ,5)
print(valeur_hyp)
```

On obtient `5.830951894845301`.

4.1 Paramètres

Les paramètres d'une fonction doivent être rentrés dans l'ordre

```
def devoirs(lycee, annee, jour) :
    print("Au lycee", lycee, ", les devoirs sur table en",
       annee, "ont lieu le", jour)

devoirs("Faidherbe","1ere annee","samedi")
devoirs("samedi","1ere annee","Faidherbe")
```

donnera

```
Au lycee Faidherbe, les devoirs sur table en 1ere annee ont
   lieu le samedi
Au lycee samedi, les devoirs sur table en 1ere annee ont lieu
   le Faidherbe
```

Il est cependant possible d'utiliser le nom des paramètres et ainsi de s'affranchir de l'ordre de la définition.

```
devoirs(jour="samedi", annee="1ere annee", lycee="Faidherbe")
   :
```

Lorsqu'une fonction ne demande pas de paramètre son appel doit comporter les parenthèses.

```
def bonjour():
    print("Bienvenue a Faidherbe")

>>> bonjour()
Bienvenue a Faidherbe
>>> bonjour
<function bonjour at 0x7f4ae14e06a8>
```

4.2 Valeurs renvoyées

Une fonction fait des calculs et on veut pouvoir les utiliser.

- On ne pourra pas invoquer les variables définies dans la fonction car celles-ci sont effacées de la mémoire à la fin du calcul de la fonction, elles ne peuvent donc pas servir à transmettre des résultats.
- Si on imprime une valeur, elle sera consultable par un humain qui lirait l'écran mais elle est inutilisable par le programme, il ne sait pas lire l'écran.

Pour recevoir les valeurs utiles calculées par une fonction, on doit utiliser la commande **`return`** : les valeurs ainsi retournées remplaceront, lors de l'usage de la fonction, l'appel de la fonction.
Par exemple `hypothenuse(3.0, 4.0)` sera remplacé par `5.0`.
Il faudra alors affecter une variable avec ce résultat ou l'utiliser dans une expression pour pouvoir l'utiliser ensuite.
Quand une fonction renvoie plusieurs valeurs, on recevra les différents résultats dans plusieurs variables (en nombre correspondant) séparées par des virgules.

```
def sphere(rayon):
    """ Entree : un nombre positif, le rayon de la sphere
        Sortie : la surface et le volume de la sphere """
    surf=4*math.pi*rayon**2
    vol=(4/3)*math.pi*rayon**3
    return surf, vol


>>> surface, volume = sphere(2)
>>> surface
50.26548245743669
>>> volume
33.510321638291124
>>>
```

Une fonction qui ne possède pas d'instruction `return` ne renvoie pas rien ; par défaut la valeur renvoyée est `None`, c'est-à-dire "rien".

4.3 Paramètres optionnels

Lorsque l'on définit une fonction on peut utiliser des paramètres qui ont une valeur par défaut mais que l'on voudrait bien pouvoir faire varier si besoin.
Python permet cette expressivité.

- Lors de l'écriture de la fonction, on écrit les paramètres optionnels **après** ceux qui ne le sont pas dans la liste des paramètres de la fonction. À ce moment, on leur affecte une valeur par défaut.
- Lors de l'appel de la fonction, on peut ignorer ces paramètres optionnels, on n'introduit que les paramètres "normaux".
- On peut aussi modifier la valeur donnée par défaut : il faut alors donner une nouvelle affectation dans l'appel de la fonction. On peut aussi indiquer simplement la valeur du paramètre (sans le nom) à condition de respecter l'ordre des paramètres.

```
import math

def logarithme(x, base = math.e):
    """Entrees : un nombre positif, x
                 un nombre positif optionnel, base
                 qui vaut e par défaut
      Sortie : le logarithme en base "base" de x"""
      y = math.log(x)/math.log(base)
      return y
```

```
>>> logarithme(10) # Sans paramètre optionnel
                   # on a le logarithme népérien
2.302585092994046
>>> logarithme(10, base = 2)
3.3219280948873626
>>> logarithme(9, 3)
2.0
```

5 Exercice : le jeu des erreurs

On veut calculer la hauteur parcourue par un objet en chute libre.
Chacun des programmes suivants est réalisé dans l'éditeur puis compilé complètement ; il engendre une erreur ou des résultats surprenants. Prévoir les problèmes et les corriger.

```
def chuteLibre1(t):
    z = 0.5*g*t**2
    return z

print(chuteLibre(2))
```

```
def chuteLibre2(t):
    g = 10
    z = 0.5*g*t**2
    print(z)

resultat = chuteLibre(2)
print(resultat)
```

```
print(chuteLibre(2))

def chuteLibre3(t):
    g = 10
    z = 0.5*g*t**2
    return z
```

```
def chuteLibre4(t):
    g = 10
    z = 0.5*g*t**2
    return z

chuteLibre(2)
print(z)
```

```
def chuteLibre5(t):
    g = 10
    t = 1
    z = 0.5*g*t**2
    return z

print(chuteLibre(2))
```

```
def chuteLibre6(t):
    g = 10
    z = 0.5*g*t**2
    return z

 print(chuteLibre(2))
```

Pour ne pas envahir les textes le docstring n'est pas écrit. Il devrait être

```
    """ Entrée : un réel t
        Sortie : la distance parcourue pour une durée t
                 pour un objet en chute libre
                 de position et vitesse initiales nulles"""
```

Indications
Les messages d'erreur peuvent être instructifs.

1. `NameError: global name 'g' is not defined`
2. `20.0` La distance calculée est affichée mais le résultat est `None`, on ne peut rien en faire.
3. `NameError: name 'chuteLibre' is not defined`
   On utilise la fonction avant sa définition.
4. `20.0`, `NameError: name 'z' is not defined`
   `z` n'existe plus en dehors de la fonction.
5. `5.0` : le temps est modifié dans la fonction, on a calculé `chuteLibre(1)`.
6. `IndentationError: unindent does not match any outer indentation level`.
   Une erreur parfois difficile à détecter

Chapitre IV

Boucles simples

Résumé *Dans ce chapitre nous allons voir comment faire beaucoup travailler un ordinateur sans devoir écrire beaucoup de code.*

1 Pourquoi les boucles ?

Un ordinateur peut exécuter des tâches à notre place mais si il faut écrire une instruction pour le moindre calcul, le temps d'écriture peut devenir rédhibitoire.
Nous allons ici introduire les instructions qui permettent de faire répéter un grand nombre de fois des opérations similaires avec peu de moyens.
Nous utilisons très souvent de tels raccourcis.

1. Faire 10 rangées de point droit (tricot).
2. Copier 100 fois "Je dois apporter mon cahier en cours".
3. Faire 10 longueurs de bassin à la brasse.
4. Écrire la table de multiplication par 7.
5. Ranger les habits dans l'armoire.

Toutes ces instructions contiennent des gestes élémentaires à répéter : soit en répétant les mêmes choses (1, 2, 3), soit en faisant une action pour chaque élément d'un ensemble, les entiers de 1 à 10 dans le cas 4, les différents habits pour le cas 5.

2 Répétitions

Nous commençons par un cas particulier de répétition inconditionnelle.

2.1 Un exemple

La méthode de Héron permet de calculer une valeur approchée de la racine carrée d'un réel positif a : on part d'une valeur pas trop éloignée, x, et on calcule $x' = \frac{1}{2}\left(x + \frac{a}{x}\right)$. C'est en fait un cas particulier de la méthode de Newton que l'on étudiera plus tard. Une propriété remarquable est que x' est plus proche de $\sqrt{a}$ que x en général. Si on répète l'opération en choisissant x' à la place de x on obtient un meilleure approximation et on peut alors recommencer un certain nombre de fois.

Voici un programme qui itère 3 fois le procédé à partir de a.

```
def racine3(a):
    """Entrée : un réel positif a
       Sortie : une valeur approchée de la racine de a"""
    x = a
    x = (x + a/x)/2
    x = (x + a/x)/2
    x = (x + a/x)/2
    return x
```

On notera qu'on n'a pas utilisé 3 variables, on recycle `x`.

On peut alors tester la fonction

```
>>> racine3(2)
1.4142156862745097

>>> racine3(9)
3.023529411764706
```

Le résultat est satisfaisant pour des réels entre 1 et 10. Comme la qualité du résultat dépend de la valeur initiale le résultat est moins spectaculaire pour d'autres valeurs.

```
>>> racine3(100)
15.025530119986813
```

Cependant on a ici une instruction que l'on répète 3 fois et il suffit de la répéter encore pour obtenir une meilleure précision. On aimerait éviter d'avoir à effectuer cette réécriture et pouvoir demander à la machine, de façon concise, de répéter un ensemble d'instructions.

```
def racine(a, n):
    """Entrée : un réel positif a et un entier n
       Sortie : une valeur approchée de la racine de a
                obtenue en itérant n fois
                la méthode de Heron """
    x = a
    # repéter n fois
        x = (x + a/x)/2
    return x
```

2.2 Traduction python

Les instructions Python pour répéter n fois un ensemble d'instructions sont

```
    for i in range(n):
        instruction 1 à répéter
        instruction 2 à répéter
        ...
        instruction p à répéter
    suite des instructions après répétition
```

Les instructions à répéter forment un bloc qui est repéré par l'indentation supplémentaire.
Notons que ce bloc est précédé du symbole de ponctuation : comme lors de la définition d'une fonction. Dans la syntaxe python toute instruction qui nécessite un bloc est suivie du deux-points.

Le programme ci-dessus devient donc

```
def racine(a, n):
    """Entrée : un réel positif a et un entier n
       Sortie : une valeur approchée de la racine de a
                obtenue en itérant n fois
                la méthode de Heron """
    x = a
    for i in range(n):
        x = (x + a/x)/2
    return x
```

3 Boucles inconditionnelles

3.1 Définition

Définition : Boucle inconditionnelle
Les instructions

```
    for i in range(n):
        instruction 1 à répéter
        instruction 2 à répéter
        ...
        instruction p à répéter
    suite des instructions après répétition
```

1. *créent une suite d'entiers 0, 1, ..., $n-1$,*
2. *pour chacun de ces entiers,*
   la variable (i ici, mais tout nom de variable est possible) prend cette valeur
   et les instructions du bloc sont exécutées.

La variable i prend bien n valeurs ; cependant ces n valeurs ne sont pas les entiers de 1 à n mais ceux de 0 à $n-1$. Ils correspondent aux indices des suites de n éléments que l'on étudiera dans un prochain chapitre, les **listes**.
Ce décalage d'indice par rapport aux habitudes mathématiques demande un apprentissage mais la cohérence avec les autres notations python le rend vite naturel.

À chaque passage dans la boucle la variable i prend une valeur différente : on peut donc l'utiliser pour faire des calculs.
Par exemple $n!$ est le produit des entiers de 1 à n, on peut donc calculer la factorielle en multipliant par $i+1$ pour chaque valeur de i entre 0 et $n-1$. On a besoin d'un support pour contenir les produits, on va donc initialiser une variable, la valeur de départ doit être neutre pour le produit, c'est 1.

```
def factorielle(n):
    """Entrée : un entier positif n
       Sortie : n! """
    produit = 1
    for i in range(n):
        k = i + 1
        produit  = produit*k
    return produit
```

On peut remarquer que, lors du premier passage ($i=0$), l'instruction `produit = produit*k` aurait donné une erreur si on n'avait pas initialisé `produit` ; dans la partie droite la variable `produit` n'existe pas encore.

Dans la fonction `racine` ci-dessus, l'instruction `for i in range(n):` introduit une variable, `i`, dont on ne s'est pas servi. On pouvait l'éviter avec le caractère tiret bas qui permet le calcul de la répétition sans utiliser de variable.

```
def racine(a, n):
    x = a
    for _ in range(n):
        x = (x + a/x)/2
    return x
```

3.2 Généralisation

La création des indices par `range` crée un compteur simple.
Quand on a besoin d'une suite d'entiers non consécutifs on peut les calculer à partir du compteur. On l'a fait dans la factorielle, voici un autre exemple.

```
def sommeImpairs(n):
    """Entrée : un entier positif n
       Sortie : la somme des n premiers impairs """
    somme = 0
    for i in range(n):
        k = 2*i + 1
        somme  = somme + k
    return somme
```

La fonction `range` permet de créer une suite finie d'entiers plus générale si on lui donne 3 paramètres : `range(debut,fin,pas)` crée une suite finie d'entiers

- qui commence par `debut`
- qui progresse de `pas` à chaque étape
- qui s'arrête dès qu'elle dépasse (ou atteint) `fin`

On peut omettre le pas : il prendra alors la valeur 1.
Si le pas est positif (resp. négatif) et si on a `debut >= fin` (resp. `debut <= fin`), la boucle n'est pas parcourue : en particulier c'est le cas d'une boucle de la forme `for i in range(0)` ou `for i in range(a, a)`.
Quelques exemples

- `range(5,17,3)` produit $(5, 8, 11, 14)$ (la borne supérieure est exclue)
- `range(5,9)` est équivalent à `range(5,9,1)` donc produit $(5, 6, 7, 8)$
- `range(5)` est équivalent à `range(0,5)` donc à `range(0,5,1)` et produit $(0, 1, 2, 3, 4)$.
- `range(6,0,-1)` produit $(6, 5, 4, 3, 2, 1)$,
- `range(4,-1,-1)` produit $(4, 3, 2, 1, 0)$.

On peut écrire alors la sommes des impairs

```
def sommeImpairs(n):
    """Entrée : un entier positif n
       Sortie : la somme des n premiers impairs """
    somme = 0
    for k in range(1, 2n, 2):
        somme = somme + k
    return somme
```

4 Complexité : 1

4.1 Introduction

Le but de la programmation sera souvent de répondre à un problème. Nous verrons qu'il existe dans de nombreux cas plusieurs algorithmes (que l'on traduit en programmes) qui permettent de résoudre un même problème.
Dans ce cas il sera utile d'essayer de comparer ces différents algorithmes en nous posant la question de l'efficacité.
Cette efficacité doit être celle de l'algorithme utilisé.

1. Ce n'est donc pas la durée de réflexion nécessaire à l'achèvement de l'algorithme.
2. Ce n'est pas non plus la facilité d'écriture du programme dans un environnement donné : *"Il a fallu 250 heures pour écrire le programme"*. On mesurerait en fait l'habileté des programmeurs, la richesse des bibliothèques ...

Nous choisirons souvent de nous intéresser au temps de calcul nécessaire [1].
On peut penser à mesurer le temps effectivement pris par le programme pour exécuter la tâche. Si on veut comparer deux algorithmes il faudra le faire sur les mêmes machines, avec le même langage, le mêmes conditions etc
Cependant cette mesure est à la fois trop précise, on n'a pas réellement besoin de la durée **exacte**, et trop imprécise car on veut savoir ce qui se passe pour des données d'entrée de plus en plus grandes (et sans perdre notre temps à faire tourner réellement le programme). On veut pouvoir prévoir l'ordre de grandeur du temps.
Pour cela on va compter les instructions "élémentaires" qu'effectue le programme.
La notion d'instruction élémentaire est difficile à définir, nous allons dans un premier temps compter les affectations `variable = calcul`. Lorsque le calcul fait appel lui-même à des fonctions il faudra compter les affectations qu'il engendre.

Dans la suite de l'année on pourra choisir de compter d'autres instructions élémentaires :

- les opérations arithmétiques,
- les comparaisons,
- les lectures et écritures dans un fichiers, ...

Dans le cas de boucles les lignes de programme de la boucle sont effectuées plusieurs fois, il faudra compter ce nombre de passages.
L'intérêt du calcul de complexité est d'anticiper le temps de calcul pour des données d'entrées de plus en plus grandes. Pour cela on fixe un mesure de la taille de cette entrée et on évalue la complexité de l'algorithme en fonction de cette taille. On obtiendra une fonction $C(n)$ où n représente la taille de l'entrée.

Définition : Complexités
La complexité est

1. **linéaire** *si elle est de la forme* $C(n) = an + b$ *ou si elle est majorée par une telle fonction.*
2. **quadratique** *si elle est de la forme* $C(n) = an^2 + bn + c$.
3. **polynomiale** *si elle est majorée par un polynôme en la taille de l'entrée*

Un complexité linéaire donnera des programmes dont le temps de calcul est approximativement proportionnel à la taille de l'entrée.
Par contre si on double la taille d'entrée d'un programme de complexité quadratique le temps de calcul va être multiplié par 4 : le temps de calcul va devenir un problème plus rapidement que dans le cas d'un programme de complexité linéaire.
De manière générale on préférera des algorithme de complexité polynomiale de degré le plus petit possible.

Exemple Dans le calcul de la factorielle, on commence par une affectation, la boucle comporte 2 affectations et elle est parcourue n fois. La complexité est dont $C(n) = 2n + 1$, elle est linéaire.

1. Nous verrons dans un prochain chapitre qu'on peut s'intéresser aussi à la quantité de mémoire utilisée à l'exécution du programme

4.2 Une étude de cas

On s'intéresse à la suite définie par $\begin{cases} u_0 = 2 \\ u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right) \end{cases}$

```
def u(n):
    """Entrée : un entier n
       Sortie : le n-ième terme de la suite u"""
    x = 2
    for i in range(n):
        x = (x + 2/x)/2
    return x
```

La complexité de cette fonction, en nombre d'affectations, est $C_u(n) = 1 + n$.

On veut ensuite calculer $S_n = \sum_{k=0}^{n} u_k$.

L'écriture naturelle d'une fonction python est

```
def S(n):
    """Entrée : un entier n
       Sortie : le n-ième terme de la suite S"""
    somme = 0
    for k in range(n+1):
        somme = somme + u(k)
    return somme
```

On notera que la borne du `range` est $n+1$ pour calculer les u_k jusqu'à u_n.
À chaque passage dans la boucle on effectue une instruction et l'appel de la fonction `u(k)` demande k instructions. La complexité est donc $C_S(n) = \sum_{k=0}^{n}(1+k) = \sum_{p=1}^{n+1} = \frac{(n+1)(n+2)}{2}$.
La complexité est quadratique.
Le calcul de `S(1000)` demande 0,06 secondes, celui de `S(10000)` demande 6 secondes.

On peut faire plus rapide en remarquant que les appels successifs à la fonction `u` calculent à chaque fois les mêmes termes avant de calculer le dernier. On va calculer les termes successifs dans le corps de la fonction principale.

```
def S_mieux(n):
    """Entrée : un entier n
       Sortie : le n-ième terme de la suite S"""
    u = 2
    somme = 0
    for k in range(n+1):
        somme = somme + u
        u = (u + 2/u)/2
    return somme
```

La complexité cette fonction, en nombre d'affectations, est $C(n) = 2 + 2(n+1)$. C'est maintenant une complexité linéaire. Le calcul de `S_mieux(10000)` demande 1 ms.

Chapitre V

Les structures conditionnelles

Résumé

Pour l'instant nous n'avons vu les programmes que sous la forme de suites d'instructions exécutées l'une après l'autre, en répétant plusieurs fois une partie d'entre elles.
On est parfois amené à n'appliquer une partie de programme que si une condition est vérifiée ou à exécuter des traitements différents selon l'appartenance de valeurs à certains ensembles.

1 Les structures conditionnelles

1.1 Instruction if

L'instruction `if` permet de vérifier des conditions avant d'exécuter un bloc d'instructions. C'est la forme la plus simple des instructions conditionnelles.

Définition : condition simple
La structure conditionnelle est composée par
- *l'instruction* `if` *suivi d'une expression booléenne puis de* `:`,
- *un bloc d'instructions indenté, ces instructions ne seront exécutées que si l'expression booléenne est évaluée avec la valeur* `True` *(vrai).*

Nous préciserons dans la suite la notion d'expression booléenne.
L'indentation s'ajoute à l'indentation déjà existante au moment de l'écriture de l'instruction `if`.
Exemple : valeur absolue (elle existe déjà sous le nom `abs`).

```
def valeur_absolue(x):
    if x < 0:
        x = -x
    return x
```

1.2 Instruction else

Parfois le test (expression booléenne) sert à discriminer : on effectue des instructions si la condition est vérifiée et d'autres instructions si elle ne l'est pas.

Définition : condition complète
La structure conditionnelle complète est composée par

- *l'instruction* `if` *suivi d'une expression booléenne puis de* `:`,
- *un bloc d'instructions indenté qui sera effectué si la l'expression booléenne vaut* `True`,
- `else:` *au même niveau d'indentation que* `if`,
- *un bloc d'instructions indenté qui sera effectué si la l'expression booléenne vaut* `False`.

Un des deux blocs (et un seul) sera toujours exécuté. On pourra donc définir une nouvelle variable en donnant sa valeur selon les cas.

```
def valeur_absolue(x):
    if x < 0:
        val_abs = -x
    else:
        val_abs = x
    return val_abs
```

Les deux branches de l'alternative peuvent contenir une instruction `return`.

```
def valeur_absolue(x):
    if x < 0:
        return -x
    else:
        return x
```

En fait, comme une instruction `return` interrompt la fonction on peut écrire le programme précédent sous la forme suivante, sans doute moins lisible que les autres.

```
def valeur_absolue(x):
    if x < 0:
        return -x
    return x
```

1.3 Instruction elif

Il arrivera régulièrement que l'on ait à distinguer plusieurs cas : il faut alors imbriquer plusieurs instructions conditionnelles

```
def signe(x):
    """Entrée : un nombre x
       Sortie : 1, 0 ou -1 selon que x est
                positif, nul ou négatif"""
    if x == 0:
        return 0
    else:
        if x > 0:
            return 1
        else:
            return -1
```

mais les choses peuvent devenir compliquées quand le nombre de cas augmente.
Le programme suivant détermine la mention du bac.

```
def afficheMention(note):
    """Entrée : la note du bac
       Sortie : la mention obtenue"""
    if note < 10:
        return 'Non admis'
    else:
        if note < 12:
            return 'Admis sans mention'
        else:
            if note < 14:
                return 'Mention assez bien'
            else:
                if note < 16:
                    return 'Mention bien'
                else:
                    return 'Mention tres bien'
```

Python propose une structure raccourcie[1] qui évite d'imbriquer les conditions.

Définition : séparations de cas

On peut traiter des cas qui s'excluent par

- *l'instruction* `if` *suivi d'une expression booléenne puis de* `:`,
- *un bloc d'instructions indenté qui sera effectué si la l'expression booléenne vaut* `True`
- *un certain nombre d'instructions* `elif` *suivies d'une expression booléenne puis du symbole* `:`, *toutes au même niveau d'indentation que* `if`
- *pour chacune un bloc d'instructions indenté qui sera effectué si la l'expression booléenne vaut* `True` *et les précédentes valent* `False`
- `else:` *au même niveau d'indentation que* `if`
- *un bloc d'instructions indenté qui sera effectué si toutes les expressions booléennes valent* `False`

On peut donc écrire l'algorithme des mentions de manière plus lisible

```
def mention(note):
    """Entrée : la note du bac
       Sortie : la mention obtenue"""
    if note < 10:
        return 'Non admis'
    elif note < 12:
        return 'Admis sans mention'
    elif note < 14:
        return 'Mention assez bien'
    elif note < 16:
        return 'Mention bien'
    else:
        return 'Mention très bien'
```

1.4 Complexité : 2

Les instructions conditionnelles introduisent une incertitude lors du calcul du nombre d'instructions ; en effet ce nombre peut être différent selon le résultat du test. On est donc amené à calculer un **majorant** de la complexité, plutôt qu'une expression de celle-ci. On essaiera, dans la mesure du possible, de donner un majorant qui peut être atteint pour certaines valeurs des paramètres.

1. On parle de **sucre syntaxique**.

2 Les expressions booléennes

Une structure conditionnelle nécessite la définition d'une **condition** dont le résultat est booléen (True ou False) : une expression booléenne. Celles-ci s'obtiennent en deux temps : on crée des résultats booléens à l'aide de comparaisons et on les combine avec des opérateurs booléens.

2.1 Les opérateurs de comparaison

Un premier test possible est l'identité : une variable (ou un résultat) est-il égal à une autre ?
Le test d'égalité ne s'écrit pas `=` qui est l'opérateur d'affectation mais `==`.
On l'a utilisé pour comparer à 0 dans la fonction `signe`.
La différence est signifiée [2] par `!=`.

Les variables qui sont comparables, principalement les nombres, peuvent être testées selon leur ordre.

	inférieur	supérieur
Strictement	`<`	`>`
ou égal	`<=`	`<=`

2.2 Les opérateurs booléens

Les résultats des comparaisons peuvent combinés de la même manière que les nombres peuvent être combinés par les opérations usuelles d'addition, soustraction, division, ...
Les opération de base utilisées dans python sont

- `and` (et) : `test1 and test2` vaut `True` si et seulement si `test1` **et** `test2` sont évalués à `True`,
- `or` (ou) : `test1 or test2` vaut `True` si et seulement si au moins un des deux test est évalué à `True`, le ou n'est pas exclusif, si `test1` et `test2` valent `True` alors `test1 or test2` vaut `True`
- `not` (négation) : inverse le résultat.

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

2.3 Évaluation paresseuse

Les définitions données ci-dessus sont correctes mathématiquement mais ce n'est pas ce qu'utilisent les langages de programmation. Lors de l'évaluation de `a or b` on sait que le résultat est vrai dès que la proposition `a` est vérifiée. Python ne calculera pas le résultat de `b` dans ce cas. La même remarque s'applique lors de l'évaluation de `a and b` : si `a` est faux, python renvoie `False` sans chercher à calculer `b`.
Cela rend le calcul non commutatif :

```
>>> True or (1/0 == 1)
True

>>> (1/0 == 1) or True
ZeroDivisionError: division by zero
```

Ce comportement est en fait très utile, nous l'utiliserons avec les boucles `while`.

2. On peut penser le symbole comme la déconstruction du signe mathématique $\neq$: une barre et le signe égal.

Chapitre VI

Listes : 1

Résumé

Jusqu'à présent nous avons utilisé des données simples : entiers, réels, booléens.
Nous avons aussi vu la possibilité de faire plusieurs calculs.
Il est naturel de vouloir manipuler des suites de données à l'aide de suites d'instructions.
Nous allons définir et apprendre à utiliser un assemblage simple de valeurs sous forme d'une suite finie.

1 Introduction

Un ensemble de données peut être vu de différentes manières

- on peut le voir comme un casier dans lequel sont rangées, à une position repérable, les différentes données
- on peut le voir comme un sac dans lequel on place les données, on peut alors les en sortir selon différents critères :
  - on sort en premier le dernier arrivé, on a une structure de **pile**, c'est un entassement d'objets
  - on sort en premier le plus ancien, on a une structure de **file d' attente**
  - on muni les élément d'une priorité et on sort en premier celui qui a la plus forte priorité

Chacune de ces structure a des avantages et des contraintes.
Nous allons définir la première structure, le tableau de rangement. Mais on verra dans un chapitre suivant que son implémentation en Python est très souple et qu'on peut l'utiliser de plusieurs manières.
D'autres structures seront vues en seconde année.

2 Définitions

2.1 Tableaux

La majorité des langages de programmation définissent des assemblages appelés **tableaux** :

- ils contiennent un nombre d'éléments fixé à la création,
- les éléments doivent être de même type, on n'y mélange pas les entiers et les chaînes de caractère, par exemple,
- les éléments sont repérés par leur adresse, un entier, qui varie, selon les langages entre 1 et n ou entre 0 et $n-1$ pour les tableaux de n éléments,
- le tableau est modifiable : non seulement on peut lire la valeur enregistrée à la position i mais on peut aussi la modifier, écrire une nouvelle valeur

Ce type de données correspond à un usage très proche de la structure des ordinateurs, ce qui était indispensable dans les débuts de l'informatique. Les ordinateurs modernes permettent à un langage récent comme Python peut définir un type beaucoup plus souple.
Nous verrons plus tard dans l'année que ce type de structure est celle des tableaux **numpy**.

2.2 Listes python

Python définit une sorte de tableaux, les **listes**, qui ont les propriétés ci-dessus mais qui libèrent la contrainte de taille fixée, nous le verrons dans un prochain chapitre, et qui n'impose pas la cohérence des données : on peut y placer tout type de valeurs et modifier le type d'élément que l'on place dans une case.

- La longueur d'une liste `L`, son nombre d'éléments est donné par la fonction `len`.
- Les indices des éléments, les positions, d'une liste de longueur n vont de 0 à $n-1$, on remarquera que ce sont les entiers fournis par `range(n`
- Les éléments d'une liste `L` sont accessibles en lecture ou en écriture (modification) par `liste[i]` pour les entiers i compatibles avec la taille.

Exemple : on peut calculer simplement la moyenne des termes d'une liste

```
def moyenne(liste):
    """Entrée : une liste de nombres
      Sortie : la moyenne des termes de la liste"""
    n = len(liste)
    somme = 0
    for i in range(n):
        somme = somme + liste[i]
    moy = somme/n
    return moy
```

La structure de la fonction

- calcul de la longueur,
- initialisation,
- parcours de la liste,
- traitement final
- retour du résultat

est générique, on écrira de nombreuses fonction sous cette forme.

2.3 Tuples

Si on encadre une suite d'éléments par des parenthèses, on définit un **tuple**.
En fait les parenthèses sont facultatives.
On accède aux éléments d'un tuple et à sa longueur de la même manière qu'une liste.
Cependant les éléments **ne sont pas modifiables**.

```
>>> a = 2, 3, 7
>>> a
(2, 3, 7)
>>> len(a)
3
>>> a[2]
7
>>> a[1] = 0
TypeError: 'tuple' object does not support item assignment
```

Renvoyer plusieurs élément dans un `return` renvoie en fait un tuple.

3 Opérations

3.1 Création

Pour manipuler une liste, il faut l'avoir créée. Plusieurs méthodes sont possibles

Définition par extension

On peut créer une liste en écrivant tous ses éléments séparés par des virgules et encadrés par des crochets.

```
carres = [1, 4, 9, 16, 25, 36, 49]
```

Cette méthode n'est possible que pour les petites longueurs.

Définition par remplissage

Une méthode plus efficace consiste à créer un casier et le remplir :
- on crée une liste "neutre",
- on la remplit pas-à-pas à l'aide d'une boucle.

La création d'une liste de taille n peut se faire par `[0]*n`.
Dans l'exemple, on choisit de donner la création d'une liste de carrés par une fonction.

```
def listeCarres(n):
    """Entrée : un entier n
       Sortie : la liste des carrés des entiers de 1 à n"""
    carres = [0]*n
    for i in range(n):
        carres[i] = (i+1)**2
    return carres
```

Définition par compréhension

Dans un exemple ci-dessus, on a créé les valeurs d'une liste en appliquant une fonction, ici $i \mapsto (i+1)^2$, aux indices produits par une instruction `for`. La construction peut être synthétisée en python de manière simple.

```
liste = [(i+1)**2 for i in range(n)]
```

Le cas particulier `[i for i in range(n)]` peut s'écrire plus simplement comme une conversion vers une liste, de type `list`. Les fonctions de conversion portent le nom du type souhaité : `list(range(i))`.

Définition par adjonctions

À voir dans un prochain chapitre.

3.2 Opérations algébriques

Python permet la manipulation de listes : dans ces opérations ce sont les listes qui sont transformées, par leurs éléments.

Addition

On peut additionner (on dit **concaténer**) deux listes.

```
>>> carres = [1, 4, 9, 16, 25]
>>> cubes = [1, 8, 27, 64]
>>> carres + cubes
[1, 4, 9, 16, 25, 1, 8, 27, 64]
```

Multiplication

Ajouter une liste plusieurs fois à elle-même revient à la multiplier par un entier

```
>>> cubes = [1, 8, 27, 64]
>>> cubes*3
[1, 8, 27, 64, 1, 8, 27, 64, 1, 8, 27, 64]
```

On peut aussi noter `n*liste` ; il est nécessaire que le facteur multiplicatif soit entier.

3.3 Extraction

On peut extraire une sous-liste d'une liste Python. L'extraction d'une tranche se fait en indiquant le premier indice choisi et le premier indice non pris séparés par un " :".

```
>>> liste = [3, 1, 4, 1, 5, 9, 2, 6, 5, 2]
>>> liste1 = liste[2 : 7]
>>> liste1
[4, 1, 5, 9, 2]
```

On peut noter que `liste[a : b]` contient $b - a$ éléments.

Valeurs par défaut

Si on n'indique pas le premier terme il prendra la valeur 0 par défaut.
Si on n'indique pas le second terme il prendra la valeur n par défaut où n est la longueur de la liste. En particulier `liste[:]` reproduit la liste entière.

```
>>> liste = [3, 1, 4, 1, 5, 9, 2, 6, 5, 2]
>>> liste2 = liste[ : 5] # correspond à liste[0 : 5]
>>> liste2
[3, 1, 4, 1, 5]
>>> liste3 = liste[5 : ] # correspond à liste[5 : 10]
>>> liste3
[9, 2, 6, 5, 2]
>
```

On remarque que `liste[:k]` et `liste[k:]` définissent une partition de la liste : la somme des deux extractions reconstitue la liste initiale.

Pas

On peut même sélectionner les termes en les prenant séparés par un pas constant.
On ajoute le pas en troisième paramètre après un " :".

```
>>> l = [3, 1, 4, 1, 5, 9, 2, 6, 5, 2]
>>> liste1 = liste[1:8:2]
>>> liste1
[1, 1, 9, 6]
```

Dans le cas d'un pas négatif les valeurs par défaut sont
n (la longueur) pour le premier terme et -1 pour le deuxième.
En particulier on peut calculer la liste avec l'ordre des termes inversé par `[ : : -1]`.

```
>>> l = [3, 1, 4, 1, 5, 9, 2, 6, 5, 2]
>>> liste[ : : -1]
[2, 5, 6, 2, 9, 5, 1, 4, 1, 3]
```

La syntaxe d'extraction permet aussi de définir une portion de liste à modifier :

```
>>> liste1 = [1,1,1,1,1,1,1,1,1,1]
>>> liste2 = [2,2,2]
>>> liste1[2:5] = liste2
>>> liste1
[1,1,2,2,2,1,1,1,1,1]
```

```
>>> liste1 = [9,8,7,6,5,4,3,2,1]
>>> liste1[6:] = liste1[:8:2]
>>> liste1
[1,1,2,2,2,1,9,7,5,3]
```

3.4 Particularités de Python

Python permet des constructions supplémentaires qui ne sont pas usuelles dans les autres langages.

Indices négatifs

On peut accéder aux éléments depuis la fin : le dernier élément a l'indice -1, le précédent l'indice -2 et on continue jusqu'au premier qui a l'indice $-n$ où n est la longueur de la liste.
Par exemple pour la liste [4, 9, 2, 3, 7]

élément	4	9	2	3	7
indice	0	1	2	3	4
indice négatif	-5	-4	-3	-2	-1

- En particulier on accède alors au dernier élément sans calculer la longueur : `list[-1]`.
- Pour une liste de longueur n, les indices i et $i - n$ définissent le même élément ($0 \leqslant i < n$).

Itération sur une liste

Le langage Python permet de parcourir la liste sans passer par les indices avec une boucle `for`.

```
def somme(liste):
    """Entrée : une liste
       Sortie : la somme des termes"""
    som = 0
    for x in liste:
        som = som + x
    return som
```

Cela est utile si on n'a pas besoin des indices.
Cela permet une création par extension d'une liste obtenue en appliquant une fonction à une liste :

```
def f(x):
    return x**2 - 3*x +4

l = [2, 4, 6, 7]
ll = [f(x) for x in l]
print(ll)
----------
[2, 8, 22, 32]
```

4 Modification de listes

Voici un comportement qui peut surprendre.

```
>>> liste1 = [3, 2, 5]
>>> liste2 = liste1
>>> liste2
[3, 2, 5]
>>> liste1[1] = 3
>>> liste1
[3, 3, 5]
>>> liste2
[3, 3, 5]
```

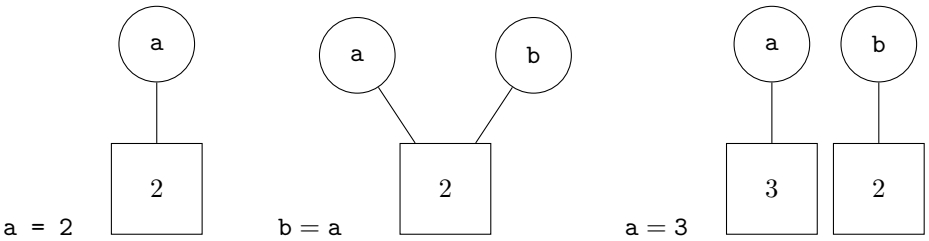
Une modification de `liste1` influe sur `liste2` !
Nous allons esquisser l'origine de ce comportement et en tirer des conséquences.

4.1 Subtilité de l'affectation

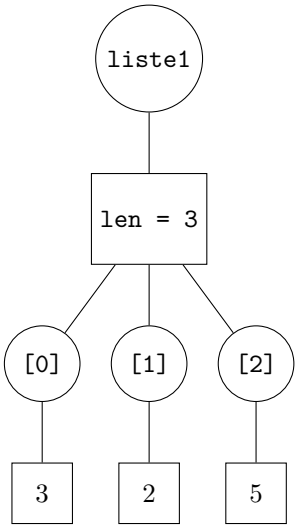
Nous avons vu le fonctionnement de l'affectation `variable = expression`.

1. L'expression est évaluée, le résultat est stocké à une adresse `ad`.
2. Un nouveau nom de variable est défini (même s'il existait déjà avant, dans ce cas l'ancien est oublié)
3. `ad` est lié au nom de la variable.

Il y a, en fait, une exception : dans le cas où l'expression est en fait une variable simple, `b = a`, alors le fonctionnement ci-dessus change. `a` et `b` sont maintenant associés à la même adresse. Ce sont deux synonymes d'une même variable. Dans le cas de variables simples cela n'a pas d'incidence car la modification d'une variable en crée une nouvelle.

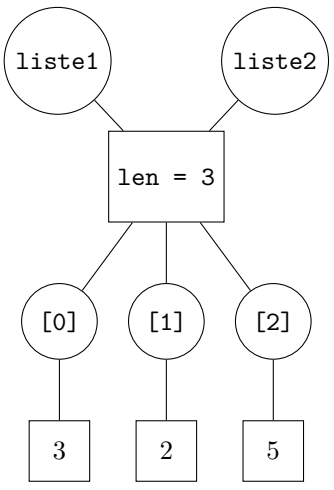


4.2 Cas des listes

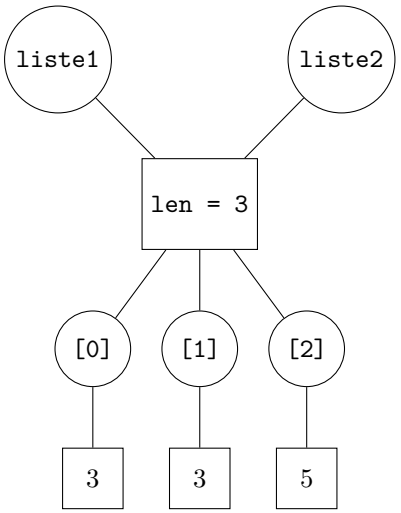


Dans le cas des listes le processus est le même mais les listes sont modifiables en interne tout en demeurant le même objet liste.
Voici ce que produisent les instructions

```
liste1 = [3, 2, 5]
liste2 = liste1
```



L'affectation `liste1[1] = 3`
ne fait que modifier un emplacement dans la liste et les deux listes associées ont toujours des valeurs égales.
On en conclut que
l'affectation `liste1 = liste2` **ne crée pas une copie.** On peut



4.3 Copie de listes

Comment alors copier une liste ?
On peut utiliser les calculs qui créent des nouvelles listes dans des cas où le résultat est semblable à l'original :

```
liste1 = liste2*1
liste1 = liste2 + []
liste1 = liste2[ : ]
```

Mais ces moyens ne sont pas fiables dans tous les cas, on retrouvera des situations semblables à celle décrite ci-dessus dans le cas, par exemple de listes dont des éléments sont des listes.
Si on veut dupliquer une liste le moyen recommandé est d'utiliser une fonction à importer.

```
from copy import deepcopy
```

La fonction `deepcopy` est alors disponible et `liste2 = deepcopy(liste1)` crée une nouvelle liste `liste2` qui a les mêmes valeurs que `liste1` mais qui est indépendante.

4.4 Modification d'une liste dans une fonction

Une fonction peut avoir pour but de modifier certaines valeurs d'une liste passée en paramètre. Dans ce cas la variable employée dans la fonction est associée à la liste initiale et les modification seront conservées à la fin de la fonction. On n'a pas besoin d'instruction `return` : le résultat est porté par la liste donnée en paramètre.
Un exemple classique est l'échange de deux termes dans une liste.

```
def echange(liste,i,j):
    """Entree : une liste et deux indices i, j
       Requis : 0 <= i,j < len(liste)
       Sortie : les termes i et j sont échangés"""
    temp = liste[i]
    liste[i] = liste[j]
    liste[j] = temp
```

On verra aussi les fonctions de tri, très importantes.

5 Exercice

Exercice VI.1 — Extraction
On définit `liste = [2, 8, 4, 6, 2, 7, 3, 5]`
Dans chacun des cas suivants prévoir le résultat de la commande.

1. `liste[1]`
2. `liste[-3]`
3. `liste[8]`
4. `liste[-8]`
5. `liste[3 : 6]`
6. `liste[3 : 7 : 2]`
7. `liste[6 : 2]`
8. `liste[6 : 2 :-1]`
9. `liste[-2 : -5]`
10. `liste[-2 : -5: -1]`
11. `liste[-5 : 6]`
12. `liste[4 : 8] = liste[0 : 4]`
    `liste`

6 Solutions

Solution de l'exercice VI.1 -

1. `liste[1] -> 8`
2. `liste[-3] -> 7`
3. `liste[8] ->` **IndexError** : list index out of range
4. `liste[-8] = 2`
5. `liste[3:6] -> [6, 2, 7]`
6. `liste[3:7:2] -> [6, 7]`
7. `liste[6:2] -> []`
8. `liste[6:2:-1] -> [3, 7, 2, 6]`
9. `liste[-2:-5] -> []`
10. `liste[-2:-5:-1] -> [3, 7, 2]`
11. `liste[-5:6] -> [6, 2, 7]`
12. `liste[4:8] = liste[0:4]`
    `liste -> [2, 8, 4, 6, 2, 8, 4, 6]`

Chapitre VII

Boucles conditionnelles

Résumé *Dans ce chapitre, on introduit la possibilité de calculs répétés sans connaissance préalable du nombre d'itérations qui seront nécessaires.*

1 Interrompre une boucle

Dans le cas d'une recherche d'un élément dans une liste, l'algorithme de base donne

```
def appartient(x, liste):
    reponse = False # On n'a pas encore trouvé x
    for y in liste:
        if x == y:
            reponse = True
    return reponse
```

On peut regretter que l'on continue à chercher même quand on a trouvé une réponse certaine (on a trouvé l'élément). Une modification possible est d'interrompre la boucle dès qu'on a trouvé l'élément. Pour cela on utilise l'instruction **`break`** qui impose au programme de sortir de la boucle.

```
def appartient(x, liste):
    reponse = False
    for y in liste:
        if x == y:
            reponse = True
            break
    return reponse
```

On peut aussi, lorsqu'aucune instruction n'est exécutée après la boucle, interrompre celle-ci avec un **`return`**. La variable n'est alors plus nécessaire car on renvoie `True` quand on a trouvé et on renvoie `False` après la boucle si l'élément recherché n'a pas été trouvé.

```
def appartient(x, liste):
    for y in liste:
        if x == y:
            return True
    return False
```

Il arrivera dans certains cas que le nombre d'itérations d'une boucle ne soit pas connu d'avance, comme ici, mais qu'en plus, on ne sache pas majorer simplement ce nombre d'itérations. Nous allons exposer une autre instruction de répétition qui permet de gérer ce cas.

2 Définition

2.1 Un exemple

On veut chercher le nombre de chiffres dans l'écriture en base 10 d'un entier naturel non nul.
On peut écrire le programme suivant

```
def nbChiffres(nombre):
    if nombre == 0:
        return 0
    elif nombre < 10:
        return 1
    elif nombre < 100:
        return 2
    elif nombre < 1000:
        return 3
    elif nombre < 10000:
        return 4
    else:
        print("Nombre trop grand")
```

`nbChiffres(438)` renverra 3.
On a comparé le nombre n avec des puissances de 10, la première fois que n est majoré par 10^p, on a $10^{p-1} \leqslant n < 10^p$ donc n a p chiffres.
Le problème est que la fonction ne donne pas de résultat pour les grands entier : même si on ajoute d'autres cas, il existera des entiers dont le résultat n'est pas calculable.
On peut améliorer le programme précédent en utilisant utilisant une variable dont la valeur sera la puissance de 10 (et une autre correspondant à l'exposant). On demande alors de faire la comparaison entre le nombre et la puissance de 10 de manière répétée :

- si le nombre est encore trop grand, on multiplie la puissance par 10 et on ajoute 1 au nombre de chiffres puis on recommence,
- si on a dépassé alors le nombre de chiffres était le bon.

On fait donc un calcul **tant que** le nombre est inférieur à la puissance de 10.
La traduction python est directe, on utilise **`while`** :

```
def nbChiffres(nombre):
    puissance10 = 1 # On initialise, pour 0
    chiffres = 0    # 0, seul nombre < 1, a 0 chiffre
    while nombre >= puisance10:
        puissance10 = 10*puissance10
        chiffres = chiffres + 1
    return chiffres
```

2.2 Syntaxe

Définition : Boucle conditionnelle
La suite d'instructions

```
    while exp_bool:
        instruction 1 à répéter
        instruction 2 à répéter
        ...
        instruction p à répéter
    suite des instructions après répétition
```

1. *teste l'expression booléenne* `exp_bool`
2. *si celle-ci est évaluée à* `True`*, les instructions 1 à p sont exécutées et le programme revient au test*

3. si elle est évaluée à `False`*, le programme passe directement à la suite des instructions.*

Une des difficultés lors de l'écriture des programmes qui utilisent les boucles `while` est la gestion de l'expression booléenne : on ne souhaite pas, en général, laisser l'ordinateur répéter une suite d'instructions sans fin [1].
Il faudra s'assurer qu'au moins une des variables est modifiée à chaque passage pour que, après un nombre fini d'itérations, la condition puisse devenir fausse.
Dans l'exemple ci-dessus la variable `puissance10` prend les valeurs $1, 10, 100, \ldots, 10^p, \ldots$successivement. Comme la limite de la suite (10^p) est l'infini, on est assuré qu'après un nombre fini d'étapes cette variable prendra une valeur supérieure au nombre donné en entrée donc le test donnera le résultat `False` et la boucle cessera..

3 Usages

3.1 `for` et `while`

Il y a donc deux techniques pour faire répéter une suite d'instructions.
En fait une seule suffirait. En effet on peut remplacer les boucles `for` par des boucles `while`.
Le code suivant remplace une boucle `for i in range(a, b, c)`.

```
i = a
while i < b: # pour c négatif, écrire while i > b:
    instructions(i)
    i = i + c
instructions suivantes
```

On voit qu'on doit initialiser la variable `i` et la modifier dans le corps de la boucle.

Ainsi la boucle `while` est plus générale que la boucle `for`.
Cependant quand on connaît à l'avance le nombre d'itérations que doit effectuer une boucle il est recommandé d'utiliser une boucle `for` : le principal avantage est qu'une boucle `for` permet d'éviter les possibles erreurs de programmation qui font qu'une boucle `while` peut tourner indéfiniment.

3.2 Limite de suites adjacentes

Si deux suites u et v sont adjacentes (avec $u_n \leqslant v_n$) on sait que la limite ℓ vérifie $u_n \leqslant \ell \leqslant v_n$. Pour calculer la limite avec une précision $\varepsilon > 0$ il suffit de calculer u_n et v_n jusqu'à obtenir $v_n - u_n \leqslant \varepsilon$.
Par exemple la constante d'Euler, γ est la limite des suites adjacentes définies Par

$$u_n = \left(\sum_{k=1}^{n} \frac{1}{k}\right) - \ln(n+1) \text{ et } v_n = \left(\sum_{k=1}^{n} \frac{1}{k}\right) - \ln(n)$$

.

```
from math import log
def gamma(epsilon):
    n = 1               # on commence par u1 et v1
    sigma = 1           # somme des 1/k
    u = sigma - log(2) # valeur de u1
    v = sigma           # valeur de v1
    while v - u  > epsilon:
        n = n +1
        sigma = sigma + 1/n
        u = sigma - log(n+1)
        v = sigma - log(n)
    return (u + v)/2
```

1. Un contre-exemple : l'interface graphique d'un ordinateur est en fait une boucle infinie qui attend les commandes et les exécute, on ne souhaite pas que le système cesse.

3.3 Recherche dans une liste

On revient sur le problème de la recherche d'un élément dans une liste : on peut utiliser la boucle `while` de manière astucieuse en évitant l'usage de la variable `dedans`. En effet la valeur de `i` lorsque la boucle s'arrête est un indicateur : si i est strictement inférieur à n, c'est qu'on a trouvé `x` dans `liste[i]`, si i vaut n c'est qu'on n'a pas trouvé.

```
def appartient3(x, liste):
    n = len(liste)
    i = 0
    while (i < n) and liste[i] != x:
        i = i + 1
    return i < n
```

On notera que l'on doit placer la condition `i < n` **avant** l'autre condition car, pour $i = n$, la deuxième condition n'a pas de sens. L'évaluation paresseuse du `and` fait qu'alors `liste[i]` n'est pas évalué si on a $i = n$.

4 Recherche dans une liste triée

4.1 Idée

Si un élément x n'appartient pas à une liste et qu'on le recherche dans cette liste, les algorithmes précédents vont comparer x avec tous les éléments de la liste. Il semble difficile de faire autrement : pour être sur qu'un élément n'est pas dans une liste il faut être certain qu'il soit différent de tous les éléments.
Cependant si la liste est triée on peut utiliser cette information supplémentaire : si x est strictement supérieur à `liste[i]` alors il ne peut pas être égal à `liste[j]` pour tout $j \leqslant i$ (quand la liste est triée par ordre croissant).
Le principe est alors similaire au jeu classique :
Agnès (**A**) demande à Bernard (**B**) de penser à un nombre entre 1 et 100 et lui annonce qu'elle va le deviner en posant au plus 7 questions auxquelles Bernard répondra en disant *trouvé*, *trop petit* ou *trop grand*. Par exemple (**B** a pensé à 88)

- **A** : "*50*" — **B** : "*trop petit*"
- **A** : "*75*" — **B** : "*trop petit*"
- **A** : "*87*" — **B** : "*trop petit*"
- **A** : "*94*" — **B** : "*trop grand*"
- **A** : "*91*" — **B** : "*trop grand*"
- **A** : "*89*" — **B** : "*trop grand*"
- **A** : "*88*" — **B** : "*trouvé*"

Au départ **A** sait que le nombre est entre 1 et 100. La première réponse diminue cet intervalle à $[51; 100]$, la seconde à $[76; 100]$ et ainsi de suite jusqu'à la dernière question qui n'en est pas vraiment une car **A** sait que le nombre est entre 88 et 88.
On remarque que **A** teste un nombre qui est la moyenne approchée[2] des bornes connues, cela permet de diminuer l'intervalle au mieux dans tous les cas.

L'algorithme de recherche va suivre ces idées.

1. On initialise deux bornes `a` et `b` avec les bornes de la liste.
2. On définit le milieu de `a` et `b`, `c`, et on compare le terme d'indice `c` de la liste à l'élément recherché.
3. S'il y a égalité on renvoie `True`, on a trouvé.
4. Si l'élément recherché est strictement plus grand, on remplace `a` par `c+1` ; en effet x ne peut pas être égale à `liste[i]` pour $i \leqslant m$.
5. Si l'élément recherché est plus strictement petit, on remplace `b` par `c`.
6. On recommence en 2 si on peut encore chercher, sinon on renvoie `False`.

2. Car on veut que le nombre testé reste un entier.

4.2 Écriture

L'algorithme ci-dessus s'exécute tant qu'il reste des éléments parmi lesquels chercher, comme on cherche entre `a` et `b`, la condition est `a <= b`.

```
def recherche(x, liste):
    a = 0
    b = len(liste) - 1
    while a <= b:
        c = (a + b)//2 # on doit avoir un indice entier
        if liste[c] == x:
            return True
        elif liste[c] < x:
            a = c + 1
        else:
            b = c -1
    return False
```

4.3 Étude théorique

L'algorithme ci-dessus, la **recherche par dichotomie**, n'est pas simple. C'est un cas particulier d'une méthode appelée "*diviser pour régner*".
L'étude qui suit est destinée à étudier cet algorithme.

Notations

- n est la longueur de la boucle.
- a_i et b_i les valeurs des variables `a` et `b` au départ du i-ième passage.
- Initialement on a $a_0 = 0$ et $b_0 = n - 1$.
- On note $c_i = \left\lfloor \frac{a_i + b_i}{2} \right\rfloor$, la valeur de `c` lors de la i-ième itération.

On sort de la boucle `while`

La première étape est de prouver que l'algorithme ne reste pas indéfiniment dans la répétition.

- Si on a $a_i > b_i$, on sort de la boucle.
- Si on a $a_i \leqslant b_i$, on calcule c_i
  (1) *Prouver qu'on a* $a_i \leqslant c_i \leqslant b_i$.
- Si `liste[c]` vaut x, le programme s'arrête.
- Sinon on remplace a_i par $a_{i+1} = c_i + 1$ ou b_i par $b_{i+1} = c_i - 1$. Dans le premier cas on a $b_{i+1} = b_i$ et dans le second, $a_{i+1} = a_i$.
  (2) *Montrer que, dans les deux cas,* $b_{i+1} - a_{i+1} < b_i - a_i - 1$.
- Ainsi l'écart entre a_i et b_i diminue d'au moins 1 lors de chaque passage dans la boucle `while`. Comme l'écart initial est $n-1$ où n est la longueur de la liste, on arrive à la condition `a > b` après n passages au plus si on n'est jamais arrivé à trouver x.

On obtient le bon résultat

- Si `x` n'est pas un élément de la liste alors la boucle `while` n'est pas interrompue par une instruction `return`. la démonstration ci-dessus montre qu'on finit par sortir de la boucle `while` donc la fonction va alors renvoyer `False`.
- On suppose maintenant que x est une des valeurs de la liste, il faut prouver que la fonction va alors renvoyer `True` ; k est un indice tel que `liste[k]` vaut x. **(3)** *Prouver qu'on a* $a_i <= k <= a_i$ *à chaque début de passage de la boucle.*
- **(4)** *Conclure.*

L'algorithme est plus efficace

On a remplacé un algorithme simple qui ne fait qu'une comparaison par élément de la liste donc au plus n opérations au total si n est la longueur de la liste par un algorithme moins intuitif qui effectue 1 addition, 1 division et 2 comparaisons à chaque passage. Il est légitime de se demander si cela vaut la peine.
On va donc essayer de majorer le nombre de passages dans la boucle `while`.
On note p ce nombre de passages on a vu qu'on avait $p \leqslant n$.
Une difficulté provient du fait qu'on ne coupe pas toujours en 2, si $b_i - a_i$ est pair, $b_{i+1} - a_{i+1}$ n'est pas le même selon que l'on cherche à droite de c_i ou à gauche.

- On commence par un cas particulier où la division est toujours régulière.
  On suppose que n est de la forme $n = 2^m - 1$, $m \geqslant 1$.
  (5) *Prouver qu'on a* $b_i - a_i = 2^{m-i} - 2$ *pour tout* i *tel que* a_i *et* b_i *sont définis.*
- On suppose maintenant qu'on majore $n < 2^m$, $m \geqslant 1$.
  (6) *Prouver qu'on a* $b_i - a_i \leqslant 2^{m-i} - 2$ *pour tout* i *tel que* a_i *et* b_i *sont définis.*
- **(7)** *En déduire qu'on a* $p \leqslant m + 1$.
- **(8)** *En choisissant* m *tel que* $2^{m-1} \leqslant n < 2^m$ *prouver qu'on a* $p \leqslant \log_2(n) + 2$.

La majoration du nombre d'opérations par $4\big(\log_2(n) + 2\big)$ est très intéressante.
Pour $n = 10^6$ on obtient un nombre d'opérations de l'ordre de 88 à comparer aux 10^6 opérations dans l'algorithme simple. Chaque fois qu'on multiplie n par 1000 le nombre d'opérations augmente seulement de 40.

4.4 Réponses

(1) On a $2a_i \leqslant a_i + b_i \leqslant 2b_i$ d'où $a_i \leqslant \frac{a_i+b_i}{2} \leqslant b_i$ puis $a_i \leqslant \left\lfloor \frac{a_i+b_i}{2} \right\rfloor = c_i \leqslant b_i$ car a_i et b_i sont des entiers.

(2) On a soit $b_{i+1} - a_{i+1} = b_i - (c_i + 1) = b_i - c_i - 1 \leqslant b_i - a_i - 1$ car $c_i \geqslant a_i$,
soit $b_{i+1} - a_{i+1} = (c_i - 1) - a_i = c_i - 1 - a_i \leqslant b_i - a_i - 1$ car $c_i \leqslant b_i$.

(3) On prouve la résultat par récurrence sur i.

1. On a $0 \leqslant k < n$ donc $a_0 \leqslant k \leqslant b_0$.
2. On suppose qu'on a $a_i \leqslant k \leqslant b_i$.
   Si `liste[c] < x = liste[k]` on sort de la boucle et il n'y a rien à prouver.
   Si `liste[c] > x = liste[k]` alors, comme la liste est croissante, $c_i > k$ puis $b_{i+1} = c_i - 1 \geqslant k$. On a aussi $k \geqslant a_i = a_{i+1}$ on trouve bien $a_{i+1} \leqslant k \leqslant b_{i+1}$.
   Si `liste[c] < x = liste[k]` alors $c_i < k$ puis $a_{i+1} = c_i + 1 \leqslant k$.
   On a aussi $k \leqslant b_i = b_{i+1}$.
3. On a bien $a_i \leqslant k \leqslant b_i$ pour tous les cas où a_i et b_i existent.

(4) Comme on a toujours $a_i \leqslant k \leqslant k$ on n'aboutit jamais à une condition `a > b` qui permet de sortir de la boucle. Cependant on sait que le programme donne un résultat, ce ne peut donc être que par une sortie `return True` dans la boucle.

(5) On fait encore une récurrence (finie).

1. On a $b_0 - a_0 = n - 1 = 2^m - 1 - 1 = 2^{m-0} - 2$.
2. On suppose qu'on a $b_i - a_i = 2^{m-i} - 2$ et qu'on revient dans la boucle `while`.
   On a $c_i = \left\lfloor \frac{a_i+b_i}{2} \right\rfloor = \left\lfloor \frac{2a_i+2^{m-i}-2}{2} \right\rfloor = a_i + 2^{m-1-i} - 1$.
   On en déduit qu'on a selon les cas, $b_{i+1} - a_{i+1} = c_i - 1 - a_i = 2^{m-1-i} - 2$ ou $b_{i+1} - a_{i+1} = b_i - (c_i + 1) = a_i + 2^{m-i} - (a_i + 2^{m-i-1}) = 2^{m-1-i} - 2$.
3. La propriété est démontrée pour tout i.

(6) La démonstration est semblable, avec une subtilité.

1. On a $b_0 - a_0 = n - 1 \leqslant 2^m - 1 - 1 = 2^{m-0} - 2$.
2. On suppose qu'on a $b_i - a_i \leqslant 2^{m-i} - 2$ et qu'on revient dans la boucle `while`.
   On commence par le cas $b_i - a_i$ pair : $b_i = a_i + 2q$ avec $q \leqslant 2^{m-i-1} - 1$.
   On a $c_i = a_i + q$ donc $b_{i+1} - a_{i+1} = q - 1 \leqslant 2^{m-i-1} - 2$
3. Si on a $b_i - a_i$ impair, $b_i - a_i = 2p + 1$, l'inégalité $b_i - a_i \leqslant 2^{m-i} - 2$ pour $b_i - a_i$ impair impose $b_i - a_i \leqslant 2^{m-i} - 3$ donc $q \leqslant 2^{m-i-1} - 2$. On a encore $c_i = a_i + q$ donc $c_i - 1 - a_i = q - 1$ et $b_i - (c_i + 1) = q$.
   Dans les deux cas on a $b_{i+1} - a_{i+1} \leqslant q \leqslant 2^{m-i-1} - 2$.
4. La propriété est démontrée pour tout i.

(7) Tous les passages sauf peut-être le dernier valident la condition du `while`. On a donc $b_i - a_i \geqslant 0$ pour $0 \leqslant i \leqslant p - 2$ (le dernier passage correspond à $i = p - 1$). En particulier $0 \leqslant b_{p-2} - a_{p-2} = 2^{m-(p-2)} - 2$ donc $2^{m-p+2} \geqslant 2 = 2^1$ puis $m - p + 2 \geqslant 1 : p \leqslant m + 1$.

(8) Si on a $2^{m-1} \leqslant n < 2^m$ alors $m-1 \leqslant \log_2(n)$ donc $p \leqslant m+1 \leqslant \log_2(n)+2$. Comme on effectue au plus 4 opérations dans chaque passage de la boucle, on obtient la majoration demandée.

Chapitre VIII

Listes : 2

Résumé

Nous avons défini le type `list` *de python et les fonctions de base : on les retrouve dans la plupart des langages de programmation, avec un type très souvent appelé* **tableau** *(array). Dans ces langages apparaît souvent un autre type d'assemblage, appelé liste, qui se construit et s'utilise pas-à-pas. Dans le langage python ces deux types sont combinés et nous allons étudier les actions supplémentaires sur les listes.*

1 Création de liste par augmentations

1.1 Définition

Jusqu'à présent nous utilisions les listes avec une longueur fixée : on pouvait changer les valeurs de chacune des positions mais on ne pouvait pas ajouter un élément supplémentaire.
On peut créer une **nouvelle** liste avec un élément de plus :

```
>>> l = [4, 3, 7, 6, 0]
>>> l1 = l + [5]
>>> l1
[4, 3, 7, 6, 0, 5]
>>> l
[4, 3, 7, 6, 0]
```

Cette opération ne modifie pas la liste initiale mais crée une nouvelle liste : cela nécessite de copier tous les éléments et a donc un coût que l'on souhaite éviter.
Python a prévu cette possibilité sous la forme d'une méthode.

Définition : Méthode
Une méthode est une fonction d'un type spécial qui est invoquée en ajoutant son nom **après** *l'objet auquel elle s'applique avec un point de séparation. Une méthode peut avoir des paramètres, ils seront placés classiquement entre les parenthèses qui suivent le nom de la méthode.*

Définition : append
L'instruction `liste.append(x)` *attache l'élément* `x` *au bout de la liste* `liste`*. C'est une instruction élémentaire qui procède sans affectation, elle ne fait que modifier sans créer.*

```
>>> l = [4, 3, 7, 6, 0]
>>> l.append(5)
>>> l
[4, 3, 7, 6, 0, 5]
```

Cette opération a un coût constant [1], il ne dépend pas de la longueur de la liste initiale.

1.2 Usage

On obtient ainsi une autre méthode pour construire une liste.
En voici quelques exemples

Application d'une fonction aux éléments d'une liste

```
def carre(liste):
    n = len(liste)
    l2 = [0]*n
    for x in liste:
        l2[i] = x**2
    return l2
```

```
def carre(liste):
    l2 = []
    for x in range(n):
        l2.append(x**2)
    return l2
```

On voit que, même si on utilise la lecture directe des éléments de la liste (`for x in liste`), la méthode classique nécessite de connaître la longueur de la liste. Cependant la méthode la plus directe est d'utiliser la construction Python :

```
def carre(liste):
    return [x**2 for x in liste]
```

Calcul des termes d'une suite

```
def fibonacci(n):
    F = [0]*(n+1)
    F[0] = 0
    F[1] = 1
    for i in range(2, n+1):
        F[i] = F[i-1] + F[i-2]
    return F
```

```
def fibonacci(n):
    F = [0, 1]
    for i in range(2, n+1):
        F.append(F[i-1] + F[i-2])
    return F
```

Dans le cas des adjonction on peut remplacer le calcul par `F.append(F[-1] + F[-2])`.

Filtrage d'une liste

On veut extraire d'une liste ses termes positifs.

```
def positifs(liste):
    pos = []
    for x in liste:
        if x >= 0:
            pos.append(x)
    return pos
```

Ici l'écriture classique n'est pas raisonnable, il faudrait commencer par calculer le nombres de termes positifs puis les écrire dans une liste de taille adaptée.

Exercice VIII.1
Écrire cette fonction.

Ici encore, Python fournit une construction très efficace.

```
def positifs(liste):
    return [x for x in liste if x >= 0]
```

1. Cela n'est vrai qu'en moyenne, on parle de coût amorti.

2 Autres méthodes

L'opération inverse de `append` est notée `pop` ; l'instruction `liste.pop()` a deux effets
1. elle modifie la liste en lui enlevant son dernier élément,
2. elle renvoie ce dernier élément. Il sera souvent utile de conserver ce dernier élément dans une variable :
   `a = liste.pop()`.

Il est indispensable d'écrire les parenthèses.

Il existe d'autres méthodes qui modifient une liste. Ce ne sont plus des opérations élémentaires, leur temps d'exécution dépend de la taille.
1. `liste.pop(k)` qui enlève l'élément d'indice k de la liste et le renvoie.
2. `liste.remove(x)` qui enlève la première apparition de x dans la liste. Si x n'est pas présent dans la liste, la méthode renvoie une erreur.
3. `liste.insert(i,x)` qui insère x dans la liste à la position i en décalant les termes suivants.
4. `liste.reverse()` qui retourne la liste.
5. `liste.sort()` qui trie la liste.

On peut simuler le comportement de `liste.pop(k)` à l'aide d'une fonction.

```
def enlever(k, liste):
    a = liste[k]
    n = len(liste)
    for i in range(k, n-1):
        liste[i] = liste[i+1]
    liste.pop()
    return a
```

On voit qu'on effectue $n-1=k$ affectations, le temps d'exécution dépend de la longueur.

Exercice VIII.2
Simuler de même les méthodes `remove` *et* `insert`.

3 Décomposition en base 2

La représentation interne des entiers dans un ordinateurs se fait en base 2.
Elle s'appuie sur la possibilité d'écrire tout entier positif $n < 2^{p+1}$ de manière unique sous la forme

$$n = \sum_{k=0}^{p} \epsilon_k 2^k \text{ avec } \epsilon_k \in \{0, 1\}$$

Pour calculer les ϵ_k, on utilise les propriétés suivantes
1. Comme 2^k est pair pour $k \geqslant 0$, ϵ_0 vaut 1 si et seulement si n est impair, c'est le **bit de parité**. ϵ_0 est donc la valeur de `n%2`.
2. Pour $0 \leqslant q \leqslant p$, la décomposition de la division entière de n par 2^q est $\sum_{k=q}^{p} \epsilon_k 2^{k-q}$.
   ϵ_q est donc la valeur de `(n//2**q)%2`.
3. Pour $q > p$ la division entière de n par 2^q donne 0, il n'y a plus de composante à calculer. On peut donc calculer la liste des ϵ_k utiles sans connaître p à l'avance en s'arrêtant dès que le quotient `n//2**q` vaut 0.

On va donc calculer la liste des coefficients [e0, e1, ..., ep] par adjonctions successives. On remarque que l'on obtient le coefficient de 2^0 (de poids faible) en premier ; si on a besoin des coefficients dans l'ordre inverse, on retournera la liste obtenue.

Voici une première écriture possible.

```
def base2(n):
    b2 = []
    q = 0
    while n//2**q > 0:
        e = (n//2**q)%2
        b2.append(e)
        q = q + 1
    return b2
```

On remarque que l'on fait des calculs inutiles :

1. `n//2**q` est calculé deux fois,
2. plutôt que le calculer, on peut passer de `n//2**q` à `n//2**(q+1)` en divisant par 2.

```
def base2(n):
    b2 = []
    while n > 0:
        e = n%2
        b2.append(e)
        n = n//2
    return b2
```

4 Solutions

Solution de l'exercice VIII.1 -

```
def positifs(liste):
    nb_pos = 0
    for x in liste:
        if x >= 0:
            nb_pos = nb_pos + 1
    pos = [0]*nb_pos
    k = 0
    for x in liste:
        if x >= 0:
            pos[k] = x
            k = k + 1
    return pos
```

Solution de l'exercice VIII.2 -

```
def oter(x, liste):
    n = len(liste):
    k = 0
    while k < n
      and liste[k] != x:
        k = k + 1
    if k < n:
        for i in range(k, n-1):
          liste[i] = liste[i+1]
        liste.pop()
```

```
def inserer(x, k, liste):
    n = len(liste):
    liste.append(x)
    for i in range(n, k, -1):
        liste[i] = liste[i-1]
    liste[k] = x
```

Chapitre IX

Textes et fichiers

1 Les chaînes de caractères

Une chaîne de caractères est un assemblage de caractères, plus précisément :
Les chaînes de caractères Python sont des assemblages :

1. homogènes, ils ne contiennent que des caractères,
2. dont les éléments sont accessibles par leur indice,
3. non mutables : on ne peut en modifier ni la longueur, ni les éléments, contrairement aux listes.

- On définit une chaîne en écrivant les caractères entre guillemets simples ou doubles : `mot = 'bonjour'` ou `nom = "Tournesol"`. La chaîne vide s'écrit donc `''` ou `""`.
- Le type des chaînes de caractères en Python est **`str`** :

```
>>> type('oui')
<class 'str'>
```

- On accède aux caractères d'une chaîne par leur numéro, sous la forme `ch[i]`. La numérotation démarre à 0. Le nombre de caractères d'une chaîne est donné par `len(ch)`.
- Dans une boucle, on peut parcourir les caractères constituant une chaîne par leurs indices

```
n=len(ch)
for i in range(n):
    x=ch[i]
    ...
```

mais on peut aussi le faire sous la forme :

```
for x in ch:
    ...
```

- Par contre on ne peut pas remplacer un caractère par un autre.

```
>>> ch = "Bonjaur"
>>> ch[4] = 'o'
TypeError: 'str' object does not support item assignment
```

2 Caractères (rappels)

Dans Python, les caractères usuels sont associées à un entier entre 0 et 255 selon un encodage ASCII sur 8 bits avec les fonctions

- `ord`, qui donne le code d'un caractère,
- `chr`, qui donne le caractère associé à un entier.

Les caractères seront utilisés le plus souvent sous la forme d'une assemblage de caractères, la **chaîne de caractères** de type `str`.
Le symbole `'\'` n'est pas un caractère, il est un symbole d'échappement et permet de définir des caractères spéciaux dans une chaînes de caractères, ils seront interprétés par la fonctions `print`.

- `\'` est remplacé par une apostrophe,
- `\\` est remplacé par le symbole \,
- `\"` est remplacé par une apostrophe double,
- `\n` est remplacé par un retour à la ligne,
- `\t` est remplacé par une tabulation ...

Ces caractères spéciaux sont signifiés à l'aide de deux signes mais sont considérés chacun comme un seul caractère.

3 Fonctions, opérations et méthodes sur les chaînes

Les fonctions imposées par le programme sont la création, l'accès à un caractère, la concaténation et les conversions de type.

- La concaténation permet de joindre deux chaînes et d'en créer une nouvelle.

```
>>> ch1 = "aujourd'hui,"
>>> ch2 = 'il fait beau'
>>> ch3 = ch1 + ch2
>>> print(ch3)
aujourd'hui,il fait beau
```

- L'extraction d'une tranche se fait comme ce qui a été vu pour les listes.

```
>>> \type{ch} = "Il faut imaginer Sisyphe heureux"
>>> ch1=ch[1:4]
>>> ch2=ch[20:]
>>> print(ch1)
l f
>>> print(ch2)
yphe heureux
```

- Voici maintenant quelques exemples de changement de type :

```
>>> str(2.5)
'2.5'
>>> int('54')
54
>>> float('3.7')
3.7
>>> list('bonjour')
['b','o','n','j','o','u','r']
```

- On peut convertir les caractère en leur code ASCII et réciproquement.
  Pour récupérer le code ASCII de la lettre a, on demande `ord('a')`, qui renvoie `97`.
  L'instruction `chr(65)` renvoie `'A'`, c'est la lettre de code ASCII égal à `65`.

Python définit un grand nombre de méthodes dont l'argument est une chaîne de caractères.
Nous allons en explorer certaines. Il n'est pas nécessaire de les connaître mais les outils acquis vont nous permettre de les écrire nous-même sous forme de fonction.

Décompte d'une lettre
`ch.count(a)` compte le nombre d'occurrences du caractère `a` dans la chaîne `ch`.

Première occurrence
`ch.index(a)` renvoie l'indice de la première occurrence du caractère `a` dans la chaîne `ch`.

Passage en majuscules
`ch.upper()` convertit tous les caractères alphabétiques de la chaîne `ch` en majuscules.

Découpage d'une chaîne
Dans le traitement des fichiers, nous aurons souvent besoin de découper des chaînes de caractères selon un critère. Par exemple, des phrases sont des mots séparés par des blancs, les données d'un fichier CSV sont séparés par des points-virgule.
`ch.split(a)` est une méthode qui fournit la liste des chaînes de caractère extraites de `ch` qui étaient délimitées par le caractère `a` (ou le début ou la fin de la chaîne) ; le caractère `a` est enlevé. Par exemple

```
>>> 'Il fait beau et chaud'.split(' ')
['Il', 'fait', 'beau', 'et', 'chaud']
```

Exercice IX.1 — Écriture des méthodes
Écrire les fonctions

1. `compte(a, ch)` *qui renvoie le nombre d'apparitions d'une lettre* `a` *dans une chaîne de caractères,*
2. `premier_indice(a, ch)` *qui renvoie le premier indice d'apparition d'une lettre dans une chaîne de caractères ; la fonction renverra* -1 *si le caractère n'est pas dans la chaîne,*
3. `majuscules(ch)` *qui renvoie une chaîne de caractères obtenue à partir de ch en remplaçant toutes les minuscules non accentuées par des majuscules ; les majuscules et les minuscules se suivent dans le même ordre dans la liste des caractères donnés par la fonction* `ord`*,*
4. `separe(ch, a)` *qui renvoie la liste des chaînes extraites de* `ch` *en découpant de part et d'autre du caractère* `a`.

4 Les fichiers

Les fichiers permettent, entre autre, de stocker des résultats ou d'en récupérer : acquisitions physiques de données, recueil de statistiques,...
Nous allons ici simplement définir un moyen de lire et écrire des données dans un fichier texte simple accessible depuis l'ordinateur. Les moyens d'accéder aux répertoires dépendent du système et de l'environnement ; ils ne seront pas étudiés ici.

4.1 Ouvrir un fichier

Pour permettre à Python d'accéder à un fichier, on utilise l'instruction
`fichier = open(nomDuFichier,mode)`. Pour cette syntaxe,

- `fichier` est le nom que l'on donne à la variable qui est associée au fichier
- `nomDuFichier` est une chaîne de caractères qui indique où trouver le fichier. Si on a su indiquer comme répertoire courant celui qui contient le fichier, il suffit de donner son nom (`'nombresPremier.txt'` par exemple).
  Mais en général, il faudra donner l'arborescence globale, du style
  `'/home/moi/travail/python/nombresPremiers.txt'` dans une arborescence Linux.
- Le mode permet de préciser l'action souhaitée ; les principaux mode sont :
  - `'r'` pour *read* . C'est le mode lecture. On lit les données du fichier.
  - `'w'` pour *write*. C'est le mode écriture. C'est un mode destructif : l'ancien fichier est détruit et remplacé par ce que l'on met. Si le fichier n'existait pas, il est créé.
  - `'a'` pour *append*. C'est le mode ajout. On ajoute ce que l'on écrit à ce qui existait précédemment.

Après l'avoir utilisée, on doit fermer la communication avec le fichier par l'instruction `fichier.close()`.
Un fichier ouvert doit toujours être fermé.

4.2 Écrire dans un fichier

Un fichier est considéré comme une chaîne de caractères. Lorsque l'on utilise le mode `'w'`, cette chaîne initiale est vide (et le fichier est créé si nécessaire) ; lorsque l'on utilise le mode `'a'`, cette chaîne initiale est le contenu du fichier au moment de son ouverture.

L'instruction **`fichier.write(chaineCaractères)`** ajoute la chaîne de caractères passée en argument au fichier. En général, on structure un peu le fichier, notamment avec des passages à la ligne (`'\n'`).
Dans l'exemple suivant, on suppose que le répertoire de travail a été bien défini.

```
def prem(n): # on suppose n plus grand que 5
    l = [2]
    for k in range(3,n):
        premier = True
        for j in range(2,k-1):
            if k%j == 0:
                premier = False
        if premier:
            l.append(k)
```

Programme IX.1 – Les premiers nombres premiers

```
fichier=open('nombresPremiers.txt','w') # creation

liste = prem(20)
for n in liste:
    ch= str(n)+'\n'    # on cree la chaine
    fichier.write(ch)  # on l'ajoute
fichier.close()        # on ferme le fichier
# on veut rajouter 23
fichier = open('nombresPremiers.txt','a')
ch=str(23)+'\n'
fichier.write(ch)
fichier.close()
```

4.3 Lire un fichier

Il y a plusieurs méthodes pour lire un fichier ouvert en mode lecture.

- **`ch`** = fichier.read() lit tout le fichier et l'affecte à la variable **`ch`**.
- L'exemple ci-dessus donne

  ```
  '2\n3\n5\n7\n11\n13\n17\n19\n23\n'
  ```

- On peut lire tout le fichier grâce à **`liste = fichier.readlines()`**.
  Ceci crée la liste des lignes du fichier (les lignes sont délimitées par `'\n'`).
- L'exemple ci-dessus donne

  ```
  ['2\n', '3\n', '5\n','7\n','11\n','13\n','17\n','19\n','23\n']
  ```

  `len(liste)` donne alors le nombre de lignes présentes dans le fichier.
- On peut lire le fichier ligne par ligne . **`ch = fichier.readline()`** lit les caractères du fichier depuis la position actuelle jusqu'à `'\n'` (ou jusqu'au bout s'il n'y a plus ce caractère). On peut alors parcourir tout le fichier ligne par ligne grâce à une boucle (le point précédent nous donne le nombre de lignes à considérer).
  L'instruction **`for ligne in fichier`** permet d'accomplir cela directement.

Programme IX.2 – Lecture des premiers nombres premiers

```
fichier = open('nombresPremiers.txt','r') #m ode lecture
liste = []
for ligne in fichier:
    p = len(ligne)
    ch = int(ligne[0:p-1])  # on enleve le caractere \n
    liste.append(ch)
fichier.close()
print(liste)

[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

5 Recherche naïve d'un mot dans une chaîne de caractères

Supposons `texte` une chaîne de caractères et `motif` une chaîne de caractères de longueur inférieure à celle de `texte`. On s'intéresse à l'apparition du motif dans le texte.

- `motif` est-il une sous-chaîne de `texte` ?
- Si oui, combien de fois `motif` apparaît-il si on lit `texte` en entier ?
- À quel(s) endroit(s) de `texte` rencontre-t-on `motif` ?

Quelles sont les positions possibles ?

Si on note n la longueur du texte et p celle du motif, le motif peut démarrer au maximum à l'indice i avec $i + p - 1 \leqslant n - 1$, soit $i = n - p$. On utilisera donc une boucle `for i in range(n-p+1)`.

Comment comparer ?

La méthode de base consiste à comparer le caractère i du texte avec le caractère 0 du motif, puis le caractère $i + 1$ du texte avec le caractère 1 du motif et ainsi de suite jusqu'au dernier caractère du motif. En Python, cela revient à comparer la chaîne extraite `texte[i:i+p]` avec le motif. Bien qu'écrit en une seule instruction, ceci demande p opérations élémentaires (des comparaisons).

Les fonctions

On commence par repérer les apparitions du motif.

Programme IX.3 – Compter les occurences d'un motif

```
def nombreOcc(texte, motif):
    """ Entrée : 2 chaînes de caractères"
          Sortie : le nombre d'apparitions
                   de la seconde chaîne dans la première"""
    n = len(texte)
    p = len(motif)
    compt = 0   # compteur initialisé à 0
    for i in range(n - p + 1):
        if motif == texte[i:i + p]:
            compt = compt + 1
    return compt
```

Exercice IX.2 — Autres fonctions
Écrire les fonctions

1. `listeOcc(texte, motif)` *qui renvoie la liste des positions en lesquelles le motif apparaît dans le texte,*
2. `present(texte, motif)` *qui renvoie* `True` *ou* `False` *selon que le motif est présent ou non dans le texte.*

6 Solutions

Solution de l'exercice IX.1 -

1. Méthode `count`

```
def compte(a, ch):
    compteur = 0
    for x in ch:
        if x == a:
            compteur = compteur + 1
    return compteur
```

2. Méthode `index`

```
def premier_indice(a, ch):
    resultat  = -1
    n = len(ch)
    position  = 0
    while position < n and resultat = -1:
        if ch[position] == a:
            resultat = position
        position  = position + 1
    return resultat
```

3. Méthode `upper`

```
def majuscules(ch):
    ecart = ord("A") - ord("a")
    resultat  = ""
    for x in ch:
    k = ord(x)
        if ord("a") <= k and k <= ord("z"):
            y = chr(ord(x) + ecart)
        else:
            y = x # On ne change que a .. z
        resultat = resultat + y
    return resultat
```

4. Méthode `split`

```
def separe(ch, a):
    reponse = []
    mot = ""
    for x in ch:
        if x != a:
            mot = mot + x
        else:
            reponse.append(mot)
            mot = ""
    return reponse
```

Solution de l'exercice IX.2 -

1. Liste des occurences

```
def listeOcc(texte,motif):
    """ Entrée : 2 chaînes de caractères"
          Sortie : la liste des positions d'apparitions
                   de la seconde chaîne dans la première"""
    n = len(texte)
    p = len(motif)
    liste = []
    for i in range(n - p + 1):
        if motif == texte[i:i + p]:
            liste.append(i)
    return liste
```

2. Tester la présence d'un motif.

```
def present(texte, motif):
    n = len(texte)
    p = len(motif)
    for i in range(n-p+1):
        if motif == texte[i:i+p]:
            return True
    return False

def present(texte, motif):
    return nombreOcc(texte,motif) > 0
```

Chapitre X

Entiers

On peut noter les deux points suivants :

- une machine n'a qu'un nombre fini d'unités de mémoire,
- une unité de mémoire ou **bit** peut-être vue comme un paramètre pouvant valoir 1 ou 0.

On en déduit que N bits peuvent représenter 2^N objets distincts.
En général N sera un multiple de 8 : 8 bits sont assemblés en un **octet**.
Voici quelques utilisations typiques :

- 8 bits ($2^8 = 256$) : codage d'une couleur dans les formats JPEG, Blu-ray ...
- 10 bits ($2^{10} = 1\,024$) : code d'une couleur dans certains formats d'images professionnels.
- 16 bits ($2^{16} = 65\,536$) : codage des sons dans les formats audio grand public (CD, MP3...)
- 24 bits ($2^{24} = 16\,777\,216$) : codage des sons dans les formats audio professionnels (mastering).
- 32 bits ($2^{32} = 4\,294\,967\,296$) : codage des entiers dans la plupart des langages de programmation.
- 64 bits ($2^{64} = 18\,446\,744\,073\,709\,551\,616$) : codage des flottants dans la plupart des langages de programmation.

Cela permet, de manière générale, de stocker des ensembles finis de valeurs : on choisira N, le nombre de bits, suffisamment grand pour pouvoir majorer le nombre de valeurs distinctes par 2^N.
Par exemple si un voltmètre donne des résultats entre -10 V et 10 V avec une précision de 3 chiffres après la virgule, on veut pouvoir coder 20000 valeurs, il faudra alors utiliser 15 bits au minimum car on a $2^{14} = 16\,384$ et $2^{15} = 32\,768$.

1 Entiers non signés

Un entier est représenté par la suite de 0 et de 1 de son écriture en base 2.

Théorème :
Pour tout entier appartenant à $\{0, 1, 2, \ldots, 2^N - 1\}$ *il existe une suite unique* $(\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{N-1})$ *appartenant à* $\{0, 1\}^N$ *telle que* $n = \sum_{k=0}^{N-1} \varepsilon_k 2^k$

En général, un entier se voit allouer 32 bits pour sa valeur. [1] Cela permet de stocker 2^{32} valeurs.
On peut donc ainsi coder en mémoire les entiers naturels de 0 à $2^{32} - 1$. Ces entiers sont appelés **non signés** car ils ont le même signe (positif).
L'ordre le plus courant des bits est de placer les coefficients dans l'ordre décroissant des puissances : par exemple $142 = 2 + 4 + 8 + 128 = 2^1 + 2^2 + 2^3 + 2^7$ sera représenté par la suite 10001110 sur 8 bits et par la suite 00000000 10001110 sur 16 bits
Comme l'entier est découpé en octets pour le stockage, il y a deux grandes stratégies :

1. Si cette taille ne convient pas, certains langages obligent à demander explicitement la taille voulue dès le début ; d'autres, comme Python, peuvent dynamiquement réallouer de l'espace si nécessaire.

- La stratégie *little-endian*, la plus courante dans les microprocesseurs : l'octet stocké en premier est celui de poids le plus faible. 142 sur 16 bits sera codé par 10001110 00000000.
- La stratégie *big-endian*, la plus courante dans les communications réseau : l'octet stocké en premier est celui de poids le plus fort. 142 sur 16 bits sera codé par 00000000 10001110.

L'écriture que nous utiliserons correspond à la stratégie big-endian.

Le calcul de la décomposition peut s'obtenir en remarquant que si $n = \sum_{k=0}^{N-1} \varepsilon_k 2^k$ alors

- $n = \varepsilon_0 + 2.\left(\sum_{k=1}^{N-1} \varepsilon_k 2^{k-1}\right)$ donc ε_0 est le reste de la division de n par 2 (`n % 2`)
- $\sum_{k=1}^{N-1} \varepsilon_k 2^{k-1}$ est la décomposition du quotient de n par 2 (`n // 2`).

On peut remarquer qu'on obtient ainsi les coefficients "à l'envers", on doit donc remplir le tableau des coefficicents à partir de la droite.

```
def binaire(n,p):
    """Entrée : deux entiers
       Requis : 0 <= n < 2**p
       Sortie : la liste des p coefficients de l'écriture en
          base 2
                de n commençant le bit de poids fort"""
    bin = [0]*p
    for i in range(p):
        bin[p-1-i] = n%2
        n = n//2
    return bin
```

1.1 Incrémentation

L'opération la plus simple pour les entiers et l'incrément de 1 : ajouter 1 à n.

Pour ajouter 1 à $n = \sum_{k=0}^{N-1} \varepsilon_k 2^k$:

- si $\epsilon_0 = 1$, on le remplace par 0,
- on continue le remplacement de 1 par 0 tant que $\epsilon_i = 1$,
- on remplace le premier coefficient nul par 1.

Par exemple 151 est représenté par `[1, 0, 0, 1, 0, 1, 1, 1]`, son successeur, 152, est représenté par `[1, 0, 0, 1, 1, 0, 0, 0]`.

```
from copy import deepcopy

def increment(bin):
    """ Entrée : la représentation en base 2 d'un entier n
        Sortie : la liste représente n + 1"""
    p = len(bin)
    b =deepcopy(bin)
    k = p -1
    while k >= 0 and b[k] == 1:
        b[k] = 0
        k = k - 1
    if k >= 0:
        b[k] = 1
    return b
```

Le dernier test est destiné au cas où la liste d'entrée est une suite de 1, codant $2^p - 1$, le successeur est alors 0 ; l'incrémene se fait modulo 2^p.

2 Entiers signés

Les entiers relatifs ont une valeur absolue et un signe, ce sont les entiers signés. Il faut réserver un bit pour stocker le signe, de sorte que l'on «perd un bit» pour stocker la valeur absolue. Sur N bits toujours les valeurs absolues seront codées sur $N-1$ bits.
L'usage général est de coder les entiers positifs entre 0 et $2^{N-1}-1$ comme dans le cas des entiers non signés : le premier bit(ε_{N-1}) sera toujours 0. Le codage des entiers négatifs peut être fait de deux manières (au moins).

- On peut coder la valeur absolue sur $N-1$ bits et donner la valeur 1 au bit de poids fort. Calculer l'opposé d'un entier est alors très simple (on change le bit de signe) mais l'addition est compliquée.
- On lui préfère le décalage : un entier p compris entre -2^{N-1} et -1 est codé par l'entier non signé 2^N+p qui est compris entre 2^{N-1} et 2^N-1. L'addition revient alors à calculer l'addition des entier non signés mais la multiplication par -1 n'est pas directe. On retrouve p en soustrayant 2^N.

Pour illustrer les calculs nous allons utiliser des entiers codés sur 8 bits : on code donc des entiers entre -128 et 127.

2.1 Addition

L'addition des entiers non signés se fait comme l'addition étudiée à l'école primaire ; on additionne des 0 et des 1 dont une retenue éventuelle dès que la somme dépasse 2.

1. 12 est codé par 00001100 et 25 par 00011001 d'où la somme

12	0	0	0	0	1	1	0	0
25	0	0	0	1	1	0	0	1
$12+25$	0	0	1	0	0	1	0	1

   On trouve bien le résultat $32+4+1=37$
2. -17 est codé par $256-17=239$ sous la forme 11101111

-17	1	1	1	0	1	1	1	1
25	0	0	0	1	1	0	0	1
$-17+25$	0	0	0	0	1	0	0	0

   On obtient 8, on remarque que la dernière retenue n'a pas été utilisée
3. De même

12	0	0	0	0	1	1	0	0
-17	1	1	1	0	1	1	1	1
$-17+12$	1	1	1	1	1	0	1	1

   On obtient le codage de $128+64+32+16+8+2+1=251$ qui correspond à $251-256=-5$

2.2 Opposé

- Si p est compris entre 1 et $2^{N-1}-1$ (on exclut le cas $p=0$) alors son opposé, $-p$, est codé par 2^N-p.
- Si p est compris entre $-2^{N-1}+1$ et -1 (on exclut le cas $p=-2^{N-1}$) alors il est codé par $q=2^n+p$ donc $-p$ vaut 2^N-q qui est codé directement.

Dans les deux cas on voit p et $-p$ sont représentés par des entiers non signés p' et $2^N-p'$.

Pour calculer $2^N-p'$ on peut remarque qu'on a $2^N=1+(2^N-1)=1+\sum_{k=0}^{N-1}2^k$.

Or, pour $p'=\sum_{k=0}^{N-1}\varepsilon'_k 2^k$, on a $(2^N-1)-p'=\sum_{k=0}^{N-1}(1-\varepsilon'_k)2^k$.

Ainsi, pour déterminer la représentation de $-p$:

1. on détermine la représentation de p,
2. on change chaque bit : 0 devient 1, 1 devient 0, on parle de complément à 2, c'est plutôt le complément à 1 (on remplace ε par $1-\varepsilon$),
3. on ajoute 1 pour obtenir la représentation de $-p$

Par exemple, pour -52 sur 8 bits

54	0	0	1	1	0	1	0	0
complément	1	1	0	0	1	0	1	1
ajouter 1	1	1	0	0	1	1	0	0

On retrouve bien la représentation de $128+64+8+4 = 204 = 256-52$

2.3 Problèmes d'overflow

Dans un des calculs ci-dessus on a vu qu'une retenue pouvait disparaître, dans le cas considéré on obtenait cependant le bon résultat. Lorsque la somme des deux entiers non signés est supérieure à 2^N, le résultat est en fait tronqué (il manque le bit correspondant à 2^N).
Mathématiquement, les additions se font modulo 2^N, c'est-à-dire que lors d'une opération on ajoute ou retranche un multiple de 2^n pour avoir un résultat appartenant à l'intervalle utilisé, ici $[-2^{N-1}; 2^{N-1}-1]$. Cela peut engendrer des résultats faux.

1. Si on ajoute deux entiers positifs dont la somme est supérieure à 2^{N-1} on va aboutir à un entier négatif :

100	0	1	1	0	0	1	0	0
50	0	0	1	1	0	0	1	0
$50+100$	1	0	0	1	0	1	1	0

   On aboutit à la représentation de 150 qui est celle de $150-256 = -106$.
2. De même si on somme deux entiers négatifs avec un résultat inférieur à -2^{N-1} on obtient un entier positif :

-55	1	1	0	0	1	0	0	1
-99	1	0	0	1	1	1	0	1
$-55-99$	0	1	1	0	0	1	1	0

   Ici encore la dernière retenue n'est pas utilisée, on apboutit à $102 = -55-99+256$.

Quand on manipule des entiers très grands en valeur absolue, il faut surveiller d'éventuels problèmes de dépassement (**overflow**). Le problème est particulièrement sensible avec la multiplication.

Chapitre XI

Réels

On a vu que les entiers étaient représentés par leur écriture en base deux, formée de 0 et de 1.
On a vu aussi que les lettres étaient représentées par des entiers
(la correspondance peut se voir à l'aide des fonctions `ord` et `chr`).
Les nombres utilisés par les sciences sont, le plus souvent, des réels. Il faut imaginer une représentation de ces nombres qui permette de les stocker en utilisant toujours le même nombre de bits afin de les manipuler plus simplement.
Cependant toute représentation des réels se heurte au problème de la finitude : les réels, comme les entiers, ne sont pas bornés et il faudra limiter les valeurs possibles. Mais il y a aussi une infinité de réels dans un intervalle et il faudra représenter les réels par des valeurs approchées : tous les réels dans un intervalle de la forme $]a-h;a+h[$ seront représenté par le même flottant a.

Sur ces problèmes d'arrondi on pourra lire :

```
http://www-users.math.umn.edu/~arnold/disasters/patriot.html
https://fr.wikipedia.org/wiki/Vol_501_d'Ariane_5
```

1 Représentations possibles sur 8 bits

Nous allons étudier divers types de représentations possibles des réels et décrire leurs inconvénients. Dans les exemples étudiés on se restreint à l'usage de 8 bits (un `octet`) pour la représentation. Cette taille n'est pas réaliste pour un usage normal des flottants mais elle permet de visualiser ce qui se passe.

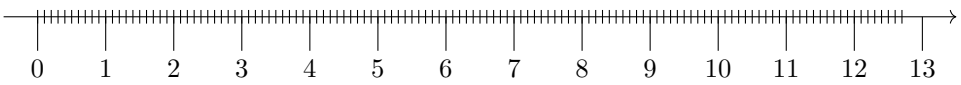
1.1 Nombre à virgule fixe

La première idée qui peut venir à l'esprit est de définir des approximations avec un nombre fixe de chiffres après la virgule. C'est ce qui est utilisé dans les tableurs.
Pour 8 bits on peut choisir

- le premier bit indique le signe,
- il reste 7 bits qui permettent de représenter $2^7 = 128$ valeurs,
- les entiers sont divisés par 10.

L'octet $\boxed{b_0 \mid b_1 \mid b_2 \mid b_3 \mid b_4 \mid b_5 \mid b_6 \mid b_7}$ $(b_i \in \{0,1\})$ représente donc

$$(-1)^{b_0}\frac{b_1+2b_2+4b_3+8b_4+16b_5+32b_6+64b_7}{10}$$



Ce format est en fait une représentation par des entiers avec un facteur d'échelle.
La précision est constante et il est très bien adapté dans le cas de calculs simples :

- échanges entre différents composants,
- calculs avec une seule unité de mesure : finance, tailles standardisées, ...

Les inconvénients sont nombreux

- La précision est constante : on approche les grandes valeurs avec la même précision que les petites. On a le plus souvent besoin d'une précision relative.
- La multiplication fait vite sortir de l'ensemble des valeurs possibles.
- On ne peut pas gérer des très petites valeurs ou des très grandes valeurs.

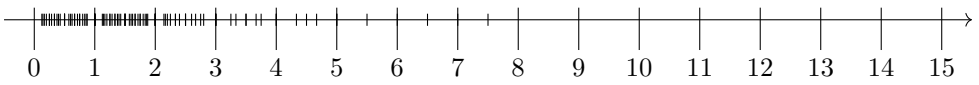
1.2 Nombres rationnels

On peut ne pas se limiter à des quotient à dénominateur fixe en codant par des nombres rationnels.
Pour 8 bits on peut choisir

- le premier bit indiquer le signe,
- 4 bits pour le numérateur,
- 3 bits pour le dénominateur, on ajoutera 1 pour éviter un dénominateur nul.

L'octet $\boxed{b_0}\boxed{b_1}\boxed{b_2}\boxed{b_3}\boxed{b_4}\boxed{b_5}\boxed{b_6}\boxed{b_7}$ ($b_i \in \{0,1\}$) représente donc

$$(-1)^{b_0}\frac{b_1+2b_2+4b_3+8b_4}{1+b_5+2b_6+4b_7}$$



Ce format est bien adapté à certains calculs algébriques
Mais il conserve de nombreux inconvénients.

- Les valeurs sont très irrégulièrement espacées.
- Il y a de nombreux doublons ($\frac{2}{1}$, $\frac{4}{2}$, $\frac{6}{3}$, ...)
- Les calculs des fonctions usuelles (sin, cos, ln, exp, ...) ne sont pas aisés.

1.3 Nombres à virgule flottante

On peut enfin utiliser une virgule flottante : au lieu de considérer des entiers divisés par une constante, on considère les nombres de la forme $\pm n.b^k$ avec n entier strictement positif, b fixé (la base) et k entier relatif, l'exposant réel.
Dans la suite on ne considérera que la base 2 : les nombres sont de la forme $\pm n.2^k$. Les nombres représentés seront toujours des fractions de la forme $\frac{n}{2^s}$.
On utilisera donc

- un bit pour le signe
- p bits pour n
- q bits pour k
- avec $1+p+q$ égal au nombre de bits utilisés pour la représentation.

Les p bits permettent de définir 2^p valeurs possibles pour n.
On remarque qu'il peut exister plusieurs représentations pour un même nombre : par exemple $1,75$ est représenté par 7.2^{-2} mais aussi par 14.2^{-3}, 28.2^{-4} ...
Pour obtenir une représentation unique on va imposer n compris entre 2^p et $2^{p+1}-1$, il y a bien 2^p valeurs.
Ainsi les p bits définissent un entier non signé N_1, q autres bits définissent un entier N_2 et cela se traduit en le réel $(1)^s\big(2^p+N_1\big).2^{N_2-h} = \left(1+\frac{N_1}{2^p}\right)2^{N_2-h+p}$.
Le problème le plus immédiat est qu'on ne peut pas représenter 0, on verra dans la partie suivante une correction possible.

Pour 8 bits on peut choisir
- le premier bit indiquer le signe,
- $p = 3$ bits pour la mantisse.
- $q = 4$ bits pour l'exposant, que l'on décale en retranchant 7 pour obtenir des exposants entre -7 et 8.

L'octet $\boxed{b_0}\boxed{b_1}\boxed{b_2}\boxed{b_3}\boxed{b_4}\boxed{b_5}\boxed{b_6}\boxed{b_7}$ $(b_i \in \{0,1\})$ représente donc

$$(-1)^{b_0}\left(1+\frac{m}{8}\right)2^{e-7} \text{ avec } m = b_7 + 2b_6 + 4b_5 \text{ et } e = b_4 + 2b_3 + 4b_2 + 8b_1$$

Exemples

1. Le plus petit nombre positif représentable est $\left(1+\frac{0}{8}\right)2^{0-7} = 0,0078125$.
2. Le suivant est $\left(1+\frac{1}{8}\right)2^{0-7} \simeq 0,00879$.
3. Le grand petit nombre représentable est $\left(1+\frac{7}{8}\right)2^{15-7} = 480$.
4. `01001011` donne $e = 9$ et $m = 3$ donc représente $\left(1+\frac{3}{8}\right)2^{9-7} = 5,5$.
5. Pour représenter π on commence par se placer entre 1 et 2 donc on considère $\frac{\pi}{2}$.
   On veut $\frac{\pi}{2}$ proche de $1+\frac{m}{8}$ donc m entier proche de $8\left(\frac{\pi}{2}-1\right) = 4,56...$.
   On choisit d'arrondir au plus proche. [1]
   On approche donc π par $\left(1+\frac{4}{8}\right)2^1 = 3,0$.
   On a $m = 3$ et $e = 8$ donc π est représenté par `01000011`.
6. 17 n'est pas représentable, il est approché par 16.
7. Pour représenter 300, on approche $\frac{300}{256}$ par une fraction de la forme $1+\frac{m}{8}$ donc m proche de $\frac{11}{8}$: on choisit $m = 1$ et 300 est approché par $\left(1+\frac{1}{8}\right)2^8 = 286$ qui est représenté par `01111001`.

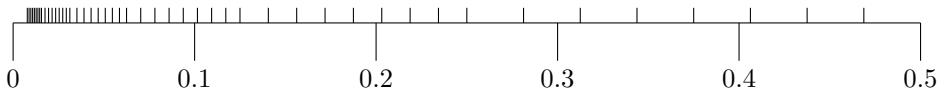


Figure XI.1 – Nombres représentables en deça de 0.5

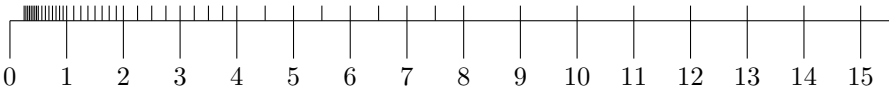


Figure XI.2 – Nombres représentables entre 0.25 et 15

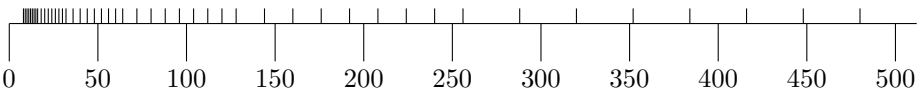


Figure XI.3 – Nombres représentables au dela de 10

1.4 Dénormalisation

Les nombres à virgule flottante permettent une meilleure répartition des nombres représentables : il y a le même nombre de valeurs entre a et $2a$ pour tout a. On parvient ainsi à une meilleure étendue des valeurs pour un nombre de bits déterminé.
Cependant la figure XI.1 montre un problème important : il n'y a pas de réel représentable autour de 0 alors que la précision est bonne au delà du premier réel représenté.
Pour obtenir un meilleur résultat on choisit de **dénormaliser** les nombres représentés lorsque l'exposant e est nul.

1. Pour approcher un nombre de la forme $\frac{n}{2}$, on choisira l'entier pair le plus proche.

Dans le cas d'une représentation avec 8 bits on remplace $(-1)^{b_0}\left(1+\frac{m}{8}\right)2^{0-7}$ par $(-1)^{b_0}\frac{m}{8}2^{0-7+1}$.
La précision est moindre pour les petits flottants mais on approche des nombres autour de 0.

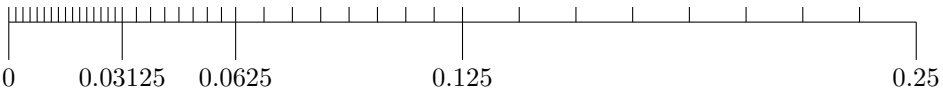


Figure XI.4 – Représentables dénormalisés

1. Le plus petit nombre positif représentable est maintenant $\frac{1}{16}2^{-2} = 0,015625$.
2. $0,015625$ est le pas pour les premiers flottants.
3. Les nombres représentables au-delà de $0,25$ sont les mêmes.

Pour des calculs scientifiques, cette représentation est celle qui est choisie.
Elle a de nombreux avantages mais il reste des problèmes qui rendent l'interprétation des calculs parfois difficiles. On verra dans la suite quelques-uns de ces problèmes.

2 Les nombres flottants dans Python

Les langages modernes utilisent des réels définis en virgule flottante sur 32 bits (simple précision) ou sur 64 bits (double précision) en respectant un standard mondial : la norme IEEE-754.
Python, dans la version 3, n'utilise que des flottants sur 64 bits.
Dans cette représentation les bits de l'exposant sont placés avant ceux de la mantisse.

2.1 Les normes IEEE-754

La norme est définie, pour 32 bits, par

- un bit pour le signe,
- 8 bits pour l'exposant donc e est compris entre 0 et 255,
- le décalage est $2^7 - 1 = 127$ et les exposant 0 et 255 ne sont pas utilisés, ils sont réservés pour des nombres dénormalisés,
- il reste 23 bits pour la mantisse.

Les 4 octets $\boxed{b_0 \mid b_1 \mid \cdots \mid b_{30} \mid b_{31}}$ représentent donc

$$(-1)^{b_0}\left(1+\frac{m}{2^{23}}\right)2^{e-127} \text{ avec } m=\sum_{k=9}^{31} b_k.2^{31-k} \text{ et } e=\sum_{k=1}^{8} b_k.2^{8-k}$$

La norme est définie, pour 64 bits, par

- un bit pour le signe,
- 11 bits pour l'exposant donc e est compris entre 0 et 2047,
- le décalage est $2^{10} - 1 = 1023$ et les exposants 0 et 2047 ne sont pas utilisés, ils sont réservés pour des nombres dénormalisés,
- il reste 52 bits pour la mantisse.

Les 8 octets $\boxed{b_0 \mid b_1 \mid \cdots \mid b_{62} \mid b_{63}}$ représentent donc

$$(-1)^{b_0}\left(1+\frac{m}{2^{52}}\right)2^{e-1023} \text{ avec } m=\sum_{k=12}^{63} b_k.2^{63-k} \text{ et } e=\sum_{k=1}^{11} b_k.2^{11-k}$$

2.2 Exemples

Pour coder 11.0, on l'écrit

$$11.0=\frac{11}{8}.2^3=\frac{8+3}{8}.2^{1026-1023}=\left(1+\frac{3.2^{49}}{2^{52}}\right).2^{1026-1023}$$

avec $3=2^0+2^1$ et $1026=2^1+2^{10}$ donc le codage sur 64 bits est

Pour coder le nombre d'Avogadro $N = 6.0210^{23}$,

1. $2^{78} \leqslant N < 2^{79}$
2. $N.2^{-78} = 1.9918509150276906$
3. $0.9918509150276906 * 2^{52} = 4466899411325793$
4. 4466899411325793 en vase 2 s'écrit
   `1111110111101001111100010000101010001101001101100001`
   ce qui donne les 52 derniers bits
5. le premier est 0, le signe est positif
6. les 12 suivants codent 75+1023 : `10001001101`

```
>>> (1+4466899411325793/2**52)*2**78
6.02e+23
```

Voici quelques valeurs obtenues avec la représentation sur 64 bits

- L'exposant $e = 1$ permet d'obtenir la valeur $\left(1 + \frac{0}{2^{52}}\right) 2^{-1023} \sim 10^{-308}$
- Les nombres dénormalisés sont de la forme $\frac{k}{2^{52}}.2^{-1023}$. Le plus petit flottant dénormalisé strictement positif est donc $2^{-1075} \sim 10^{-324}$
- Le plus grand réel représentable est $\left(1 + \frac{2^{52}-1}{2^{52}}\right) 2^{1023} \sim 10^{308}$,
  le précédent diffère de $2^{971} \sim .10^{292}$.
- La précision pour les réels entre 1 et 2 est de $2^{-52} \sim 2.10{-}16$.
- Le plus petit entier qui n'est pas représenté sans erreur est $2^{53} + 1$.

```
>>> 2**53+1
9007199254740993
>>> float(2**53+1)
9007199254740992.0
```

2.3 Problèmes d'arrondi

Comparaison à zéro

Deux nombre trop proches l'un de l'autre seront considérés comme égaux.

```
>>> i=1
>>> while 1.0+10**(-i)>1.:
...     i+=1
>>> i
16
```

Cela signifie qu'une variation relative de 10^{-16} n'est par perceptible.

Exercice XI.1 — Equation de degré 2
Ecrire une fonction `racine(a,b,c)` qui donne le nombre de racines de l'équation $ax^2 + bx + c = 0$. La tester pour $a = 0.1$, $b = 0.6$, $c = 0.9$.

Le résultat de la soustraction de deux nombres a et b proches peut être de l'ordre de l'imprécision des calculs.
On a vu dans le paragraphe précédent que l'incertitude semble de l'ordre de 10^{-16} pour un nombre de l'ordre de 1 donc si on soustrait des nombres de l'ordre de 10^{16} on devrait avoir une imprécision de l'ordre de 1.

Sommes et ordre de sommation

Si on doit ajouter 3 nombres, on commencera par ajouter les deux plus petits puis on ajoutera le troisième, l'erreur générée sera alors moins grande car elle est proportionnelle au plus grand des termes de chaque addition.
Par exemple, on sait que $\sum_{k=1}^{n} \frac{1}{k^4}$ a pour limite $\frac{\pi^4}{90}$.
Si on opère dans l'ordre, on commence par ajouter les plus grands termes, si on opère en partant de la fin, on commence par ajouter les plus petits.
Voici ce que donne le calcul :

```
from math import pi
def zeta(n):
    l=[1/i**4 for i in range(1,n+1)]
    somme=0
    for i in range(n):
        somme=somme+l[i]
    sommeinverse=0
    for i in range(n):
        sommeinverse=sommeinverse+l[-i-1]
    return somme-pi**4/90,sommeinverse-pi**4/90

print(zeta(66))
print(zeta(2**18))
```

```
(-1.1333517326850284e-06,-1.1333517324629838e-06)
(-2.7688962234151404e-13,2.220446049250313e-16)
```

2.4 Le module fractions

Si on utilise le module `fractions` on peut travailler avec des valeurs fractionnaires des nombres et éviter beaucoup des problèmes précédents. Le prix à payer en termes d'efficacité est par contre élevé et de toutes façons, ça ne règle pas le problème de la représentation des nombres irrationnels.

```
from fractions import Fraction
a=Fraction(16,-10)
```

Le module *decimal* permet aussi de travailler avec des nombres décimaux.

```
from decimal import Decimal
b=Decimal('1.1') #noter l usage des guillemets
```

On peut aussi choisir la précision (le nombre maximal de décimales).

```
from decimal import getcontext
getcontext().prec=100
```

Forcément, choisir 100 décimales rend les calculs moins rapides.

Chapitre XII

Zéros de fonctions

Un des problèmes numériques les plus connus est celui de la résolution des équations : on se donne un équation sous la forme $f(x) = 0$ est on cherche, une valeur de x, x_0, telle que $f(x_0) = 0$. Dans la suite on nommera **zéro** de f une telle valeur.

On voit très vite qu'on a besoin de préciser le contexte :

- un zéro doit-il appartenir à un ensemble donné ?
- cherche-ton un zéro ou tous les zéros ?
- sait-on s'il existe un zéro, s'il est unique ?

On supposera ici que la fonction f est définie sur une partie de $\mathbb{R}$ et à valeurs réelles.

Ce problème est au cœur de nombreux résultats mathématiques :

- existence d'un zéro : théorème de Rolle
- unicité du zéro : injectivité
- résolution avec extractions de racines dans le cas des polynômes de degré 2, puis 3, puis 4
- fonctions réciproques, par exemple $\arcsin(a)$ est l'unique solution de $\sin(x) = a$ sur $[-\pi/2; \pi/2]$
- ...

Résoudre une équation est parfois considéré comme la recherche d'une expression du zéros à l'aide d'un ensemble de fonctions considérées comme connues. On considère plutôt les équations de la forme $f(x) = a$ avec f bijective et on cherche la fonction f^{-1}. Une forme d'expertise en mathématique serait alors de savoir résoudre le plus possible d'équations.

Cependant si l'objectif est d'avoir un valeur numérique du zéro, on n'aura pratiquement jamais une valeur exacte car les fonction utilisées ne sont exactement calculables. L'objectif de ce chapitre sera, inversement, de calculer une valeur approchée du zéro pour un ensemble très général d'équations, contenant celles que l'on estime résolubles mathématiquement.

On se place dans le cadre des fonctions continues sur un intervalle $[a; b]$ et on cherche des approximations d'une solution de $f(x) = 0$; on suppose que f admet au moins un zéro sur $[a; b]$.

Il sera souvent utile de se restreindre à un intervalle dans lequel on a prouvé que la fonction admet une racine unique : la figure XII.1 représente ce type de cas.

Nous étudions dans ce chapitre deux méthodes :

1. la méthode de dichotomie est valable pour les fonctions continues et donne une approximation dans tous les cas où on a su encadrer un zéro de la fonction,
2. la seconde méthode, de Newton, nécessite d'avoir une fonction dérivable et peut ne pas donner d'approximation. Cependant, en cas de convergence, elle est très rapide. De plus elle se généralise au cas de recherche de zéros d'une fonctions de $\mathbb{R}^n$ vers $\mathbb{R}^n$.

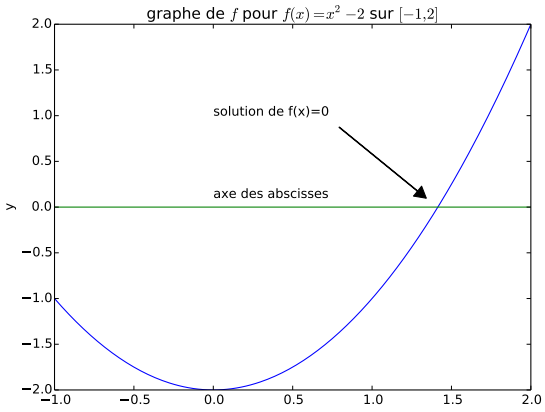


Figure XII.1 – Fonction à zéro unique

1 Méthode de dichotomie

1.1 La méthode mathématique

On travaille ici sur un intervalle $I = [a, b]$ en supposant $a < b$ et $f(a)f(b) < 0$.
L'idée est d'utiliser le théorème des valeurs intermédiaires pour diminuer la largeur de l'intervalle d'étude par 2.

1. On pose $c = \frac{a+b}{2}$ et on calcule $f(c)$,
2. Si $f(a)f(c) < 0$, alors f s'annule sur $]a; c[$.
3. Sinon f s'annule sur $[c; b[$ et on étudie sur $[c, b]$

On peut alors itérer cette idée en notant $a_0 = a$ et $b_0 = b$ et en construisant trois suites (a_n), (b_n) et (c_n) de la façon suivante :

1. on pose $c_n = \frac{1}{2}(a_n + b_n)$ et on calcule $f(c_n)$,
2. si $f(a_n)f(c_n) \leqslant 0$, alors on définit $a_{n+1} = a_n$ et $b_{n+1} = c_n$,
3. sinon on définit $a_{n+1} = c_n$ et $b_{n+1} = b_n$.

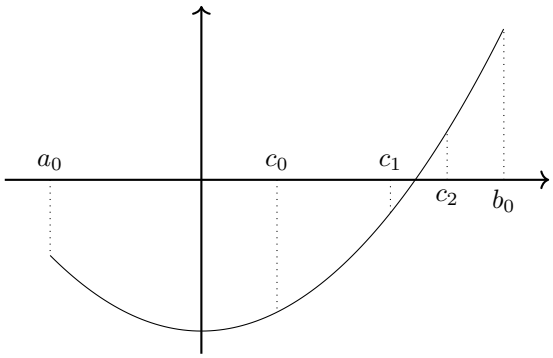


Figure XII.2 – Premières valeurs de c_n

Pour l'exemple initial on a représenté, dans la figure XII.2, les premières valeurs de c_n.
On a $a_1 = c_0$, $a_2 = a_3 = c_1$, $b_0 = b_1 = b_2$ et $b_3 = c_2$.

Théorème :
Les deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ sont adjacentes et leur limite commune est un zéro de f.
De plus on a $b_n - a_n = \dfrac{b_0 - a_0}{2^n}$.

1.2 Algorithme

Programme XII.1 – Calcul d'un zéro par dichotomie

```
def dichotomie(f, a, b, epsilon):
    """Entrées : un fonction, 3 flottants
       Requis  : a < b, f s'annule sur [a; b],  epsilon > 0
       Sortie  : une valeur approchée de f sur [a; b]
                 à epsilon près"""
    while b - a > epsilon:   # precision non atteinte
        c = (a+b)/2          # calcul du milieu de a et b
        if f(a) * f(c) <= 0:
            b = c            # on change l'extremite droite
        else:
            a = c            # on change l'extremite gauche
    return (a+b)/2
```

1.3 Analyse de l'algorithme

L'algorithme termine

Il faut prouver que l'on sort de la boucle `while`.
On a $b_n - a_n = \dfrac{b_0 - a_0}{2^n}$. Ainsi $\lim\limits_{n\to+\infty} b_n - a_n = 0$ donc $b_n - a_n \leqslant \varepsilon$ pour n supérieur à un entier N.
`b - a` vaut $b_n - a_n$ après n passages dans la boucle donc la condition `b - a > epsilon` est fausse pour $n \geqslant N$: on est sûr que l'on n'itère la boucle qu'un nombre fini de fois.

L'algorithme est correct

Il faut prouver qu'il existe un réel r tel que $f(r = 0)$ et $|r - c| < \varepsilon$ si c est la valeur renvoyée par `dichotomie(f, a, b, epsilon)`. La limite commune, r, des suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ est un zéro de f et vérifie $a_n \leqslant r \leqslant b_n$ on a donc $a - \dfrac{a+b}{2} \leqslant r - \dfrac{a+b}{2} \leqslant b - \dfrac{a+b}{2}$ à chaque étape d'où $\left|r - \dfrac{a+b}{2}\right| \leqslant \dfrac{b-a}{2} \leqslant \dfrac{1}{2}\varepsilon < \varepsilon$ à la sortie de la fonction.

Nombre d'opérations

On peut déterminer le nombre de parcours de la boucle en cherchant le premier entier k tel que $\dfrac{b-a}{2^k} \leqslant \epsilon$. On a alors $\dfrac{b-a}{\epsilon} \leqslant 2^k$ c'est-à-dire $\ln\left(\dfrac{b-a}{\epsilon}\right) \leqslant k * \ln(2)$. Il suffit donc $k = \left\lfloor \dfrac{\ln\left(\frac{b-a}{\epsilon}\right)}{\ln(2)} \right\rfloor + 1$ opérations à effectuer pour avoir une valeur approchée de r à ϵ près.

1.4 Remarques

- En pratique, on ne connaît pas toujours la fonction f en tout point mais on dispose seulement d'un nombre fini de ses valeurs (penser par exemple à l'acquisition d'un son numérique) sous la forme d'une liste $y = [y_0, y_1, \cdots, y_{n-1}]$ de longueur n qui représente la liste des valeurs prises par la fonction.
  Si on suppose la fonction croissante (ou décroissante), une bonne approximation de la racine r sera alors y_q avec q le premier entier tel que $y_q > 0$ (y_{q-1} serait aussi une bonne approximation sans qu'on puisse vraiment les départager).
  On est donc amené à chercher, dans une liste ordonnée, le premier terme qui dépasse 0. Une méthode coûteuse serait de parcourir la liste jusqu'à trouver un terme positif, on utilisera plutôt une dichotomie comme on l'a vu dans le cours sur les recherches dans des listes triées.
- Le fait que l'algorithme termine est prouvé mathématiquement mais il faut tenir compte des erreurs dues à l'usage des flottants. Il vaut mieux éviter des valeur de `epsilon` trop petite pour que le calcul de `(a+b)/2` donne un résultat distinct de `a`. Dans la pratique on choisira ε supérieur à 10^{-12} fois l'ordre de grandeur de a et b.
- La valeur renvoyée n'est correcte que si les erreurs d'arrondi ne perturbent pas les tests.
  En effet un test de comparaison à 0 n'est pas fiable pour des flottants.
  Par exemple la fonction $f : x \mapsto \sin(x) - x + \dfrac{x^3}{6} + \dfrac{x^5}{120}$ est plate autour de son unique racine et le signe de $f(x)$ n'est pas bien calculé pour des valeurs de x proches de 0.

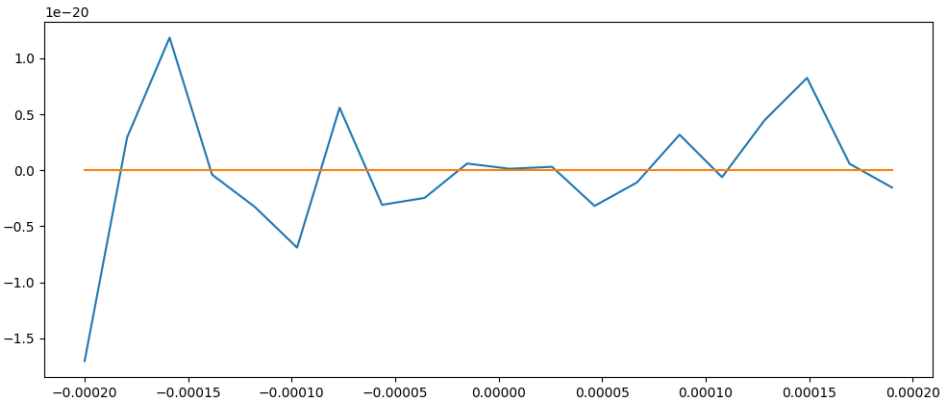


Figure XII.3 – Signes non exacts de $x \mapsto \sin(x) - x + \frac{x^3}{6} + \frac{x^5}{120}$

`Dichotomie(f, -0.4, 0.6, 1e-6)` renvoie `0.000207996` qui n'est pas une valeur approchée à ε près.

2 Méthode de Newton

2.1 La méthode

On suppose pour cette méthode f de classe $\mathcal{C}^1$ sur un intervalle I
avec f s'annulant en au moins un point de I.
On part de $x_0 \in I$; la tangente à la courbe représentative de f en $\big(c, f(c)\big)$ recoupe l'axe des abscisses en un point $(x_1, 0)$. Si x_1 appartient à I, on réitère le procédé avec le point $(x_1, f(x_1))$.
On continue ainsi (quand c'est possible).
On remarque dans l'exemple de la figure XII.4 que x_2 est proche d'une racine. Dans le graphe la tangente en $\big(x_2, f(x_2)\big)$ est indiscernable de la courbe donc x_3 sera une excellente approximation.

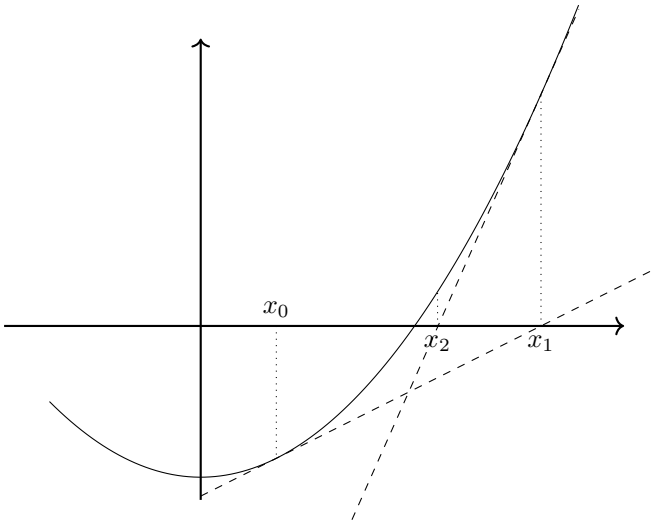


Figure XII.4 – 2 étapes dans la méthode de Newton

La tangente en $(c, f(c))$, T_c, admet pour équation $y - f(c) = f'(c)(x - c)$.
La relation de récurrence qui définit la suite (x_n) est donc

$$x_0 = c \text{ et } \forall n \in \mathbf{N}, \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

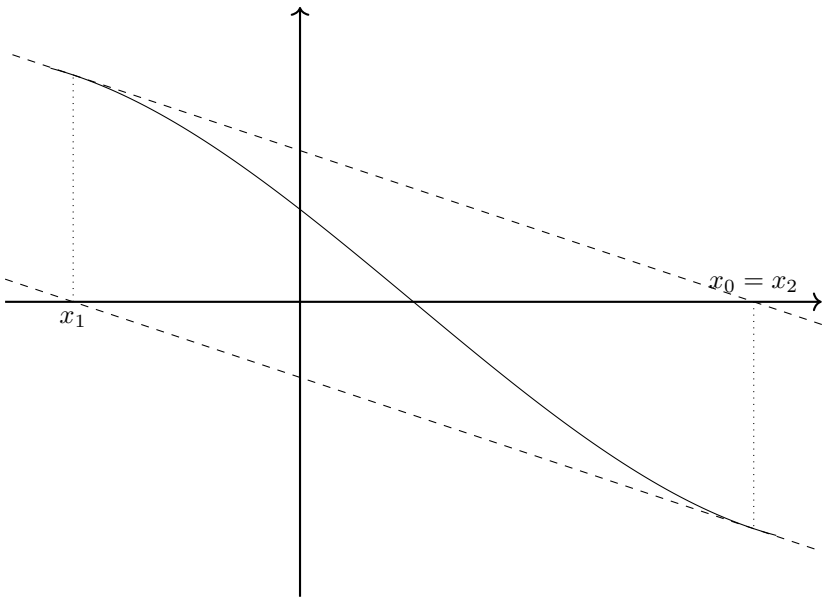
2.2 Convergence

La convergence est démontrable dans un cas particulier.

Théorème :
f est de classe $\mathcal{C}^2$ sur $[a;b]$ avec
- *f' et f'' de signe constant*
- *$f(a).f(b) < 0$.*

Si $x_0 = b$ quand f' et f'' sont de même signe ou $x_0 = a$ pour f' et f'' de signe opposés alors la suite (x_n) définie par $x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$ est bien définie et monotone dans $[a;b]$.
Cette suite converge vers l'unique racine de f dans $[a;b]$.
On a, de plus $x_{n+1} - x_n \sim r - x_n$ quand n tend vers l'infini et $|r - x_{n+1}| \leqslant \dfrac{M_2}{m_1}(r - x_n)^2 = K(r - x_n)^2$ pour M_2 majorant de $|f''|$ sur $[a;b]$ et m_1 minorant de $|f'|$ sur $[a;b]$.

1. Dans le cas général, il peut ne pas y avoir de convergence :
   - soit parce que $f'(x_n) = 0$, par exemple pour $f(x) = \frac{1-x}{x^2}$ et $x_0 = 2$
   - soit parce que la valeur de x_n n'appartient plus à un intervalle où on sait calculer f, par exemple pour $f(x) = 4\sqrt{x} - x$ et $x_0 = 1$
   - soit parce que la suite n'admet pas de limite, par exemple pour $f : x \mapsto \frac{1}{27}\big(2x^3 - 3x^2 - 21x + 11\big)$ et $x_0 = 2$. On trouve alors $x_1 = -1$ puis $x_2 = 2$, la suite est 2-périodique.



2. Si le zéro, r, de la fonction n'est pas un minimum ou un minimum, alors les conditions du théorème sont vérifiées dans un intervalle autour de r. Dans la pratique on pourra approcher un zéro par quelques étapes de dichotomie et on continuera ensuite avec la méthode de Newton.
3. En effet, si on a $|x_0 - r| \leqslant \frac{1}{2K}$, l'inégalité $|r - x_{n+1}| \leqslant K(r - x_n)^2$ donne $|r - x_n| \leqslant 1K.2^{2^n}$: la convergence mathématique est très rapide, le nombre de chiffres exact double à chaque étape. Cela signifie que si x_0 est une valeur approchée à 10^{-1}, il suffit de 4 étapes pour obtenir la précision maximale pour les flottants de 10^{-16}.

2.3 Codage de la méthode de Newton

La fonction calcule x_n en prenant en paramètres ϵ, x_0, f et f'. On doit définir f et f' par des fonctions python car on ne sait pas calculer la dérivée de f en Python.
L'équivalent $x_{n+1} - x_n \sim r - x_n$ du théorème permet de donner un critère de sortie de la boucle `while` sans connaître r.

Programme XII.2 – Calcul d'un zéro par la méthode de Newton

```
def Newton(x0, f, df, epsilon):
    """Entrées : un réel, deux fonctions et un réel
       Requis : df est la dérivée de f, epsilon > 0
       Sortie : une valeur approchée d'un zéro de f"""
    x = x0
    difference = epsilon +1 # pour faire au moins un calcul
    while difference > epsilon:
        x_old = x
        x = x - f(x)/df(x)
        difference = abs(x - x_old)
    return x
```

3 fsolve

Le module `scipy` contient, dans la sous-bibliothèque `optimize` une fonction `fsolve` qui résout efficacement les équations. C'est la méthode à utiliser quand la résolution d'une équation est un outil d'un projet plus large.

```
from scipy.optimize import fsolve
```

- `fsolve` reçoit pour paramètres une fonction f dont on cherche un zéro et un valeur initiale autour de laquelle on cherche une solution.

```
def f(x):
    return x**2 - 2
>>> fsolve(f, 1)
array([1.41421356])
```

- On remarque que la réponse est une liste (en fait un tableau `numpy`) avec un seul élément : on devra sortir la réponse.

```
a = fsolve(f, 1)
zero = a[0]
```

- La raison est que `fsolve` peut chercher un zéro d'une fonction de $\mathbb{R}^n$ vers $\mathbb{R}^n$

```
def F(u):
    x, y = u
    return x + y - 3, x*y -2
>>> fsolve(F, (1, 1))
array([1., 2.])
```

- Si la fonction dont on recherche un zéro contient des paramètres on peut les définir lors de la résolution à l'aide de l'argument optionnel nommé par `args`.

```
def g(x, a):
    return x**a -2
>>> fsolve(g, 1, args = 2.5)
array([1.31950791])
```

- On peut, par exemple, chercher un zéro d'un polynôme de degré 3 en donnant les coefficients.

```
def poly3(x, a, b, c):
    return x**3 + a*x**2 + b*x + c
>>> fsolve(poly3, 1, args = (1, 2, 3))
array([-1.2756822])
```

Chapitre XIII

Intégration numérique

Le but de ce chapitre est de calculer, de diverses manières l'intégrale d'une fonction f sur un segment $[a;b]$. On supposera, lors des calculs que la fonction f est suffisamment régulière.
On va utiliser deux outils :

1. découper l'intervalle d'intégration en n petits intervalles,
2. approcher l'intégrale en remplaçant la fonction à intégrer par une fonction simple dont on connaît les primitives.

Dans le chapitre f est une fonction (au moins) continue sur un segment $[a;b]$ et on note

$$I = \int_a^b f(t)\mathrm{d}t$$

1 Subdivision

Une subdivision de $[a;b]$ est une suite finie $(a_0, a_1, \ldots, a_n)$ qui vérifie

1. $a_0 = a$,
2. $a_n = b$,
3. $a_0 < a_1 < a_2 < \ldots < a_{n-1} < a_n$

La subdivision est **régulière** si on a $a_1 - a_0 = a_2 - a_1 = \cdots = a_n - a_{n-1}$.
La valeur constante de $a_k - a_{k-1}$ est le **pas** de la subdivision et il vaut $\dfrac{b-a}{n}$.
On a alors $a_k = a + k\dfrac{b-a}{n}$.
Dans toute la suite on ne considérera que les subdivisions régulières de $[a;b]$ et on calculera, à l'aide de la formule de Chasles, pour n fixé

$$I = \int_a^b f(t)\mathrm{d}t = \sum_{k=0}^{n-1} \int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t$$

2 Méthode des rectangles

On est donc amené à approcher des intégrales $\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t$ avec $a_{k+1} - a_k = \frac{b-a}{n}$ qui devient "petit" si on choisit n assez grand.
La continuité de f [1] implique qu'on a $|f(x) - f(y)| \leqslant \varepsilon$ pour $x, y \in [a_k; a_{k+1}]$ dès que n est assez grand. On peut ainsi approcher

$$\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t \simeq \int_{a_k}^{a_{k+1}} f(c_k)\mathrm{d}t = (a_{k+1} - a_k)f(c_k)$$

avec c_k appartenant à $[a_k; a_{k+1}]$.
On aboutit à $I = \sum_{k=0}^{n-1}(a_{k+1} - a_k)f(c_k) = \frac{b-a}{n}\sum_{k=0}^{n-1} f(c_k)$, ce sont les **sommes de Riemann**.

La méthode des rectangles à gauche consiste à choisir $c_k = a_k$: $R_{g,n} = \frac{b-a}{n}\sum_{k=0}^{n-1} f\left(a + k\frac{b-a}{n}\right)$,

la méthode des rectangles à droite consiste à choisir $c_k = a_{k+1}$: $R_{d,n} = \frac{b-a}{n}\sum_{k=1}^{n} f\left(a + k\frac{b-a}{n}\right)$.

Pour $f : x \mapsto= 3\sin\left(\frac{x+1}{2}\right)$ sur $[0; 5]$ avec $n = 10$ on obtient les figures ci-dessous.
La partie hachurée représente la surface utilisée comme approximation de l'intégrale.

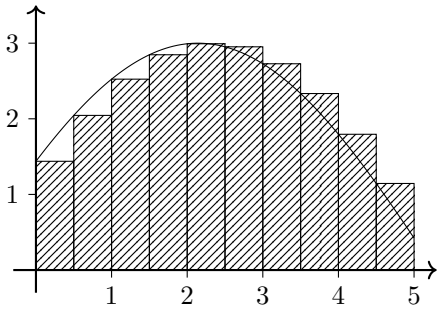


Figure XIII.1 – Rectangles à gauche, $n = 10$

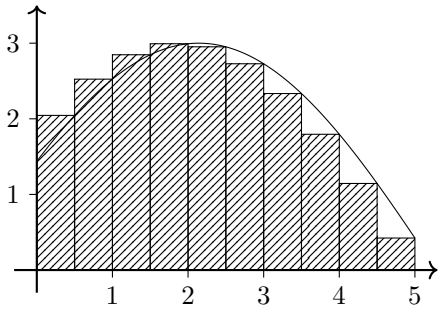


Figure XIII.2 – Rectangles à droite, $n = 10$

Programme XIII.1 – Méthode des rectangles (à gauche)

```
def rectangleG(f, a, b, n):
    pas = (b - a)/n
    somme = 0
    for i in range(n):
        somme = somme + f(a + i*pas)
    return somme*pas
```

Exercice XIII.1 — Mesure de l'erreur
Prouver que si f est de classe $\mathcal{C}^1$ sur $[a; b]$ et si $M_1 = \sup\{|f'(t)| \ ; \ t \in [a; b]\}$, alors

$$\left|\int_{a_k}^{a_{k+1}} f(t)dt - (a_{k+1} - a_k)f(a_k)\right| \leqslant \frac{M_1}{2}(a_{k+1} - a_k)^2$$

En déduire qu'on a $\left|I - R_{g,n}\right| \leqslant \frac{M_1.(b-a)^2}{2n}$.

1. Plus précisément la continuité uniforme de f sur $[a; b]$

3 Méthode des trapèzes

Dans la méthode précédente on a approché f par une constante, c'est-à-dire un polynôme de degré 0. On va maintenant approcher f par un polynôme de degré 1, une fonction affine de la forme $x \mapsto \alpha x + \beta$.
On choisit d'approcher f sur $[a_k; a_{k+1}]$ par une fonction affine g_k avec les conditions $g_k(a_k) = f(a_k)$ et $g_k(a_{k+1}) = f(a_{k+1})$.
L'intégrale sur $[a_k; a_{k+1}]$ est alors approchée par la surface d'un trapèze.

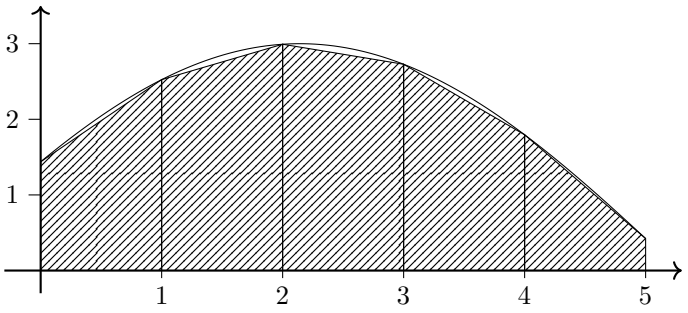


Figure XIII.3 – Méthode des trapèzes, $n = 5$

Exercice XIII.2
Prouver que si f est une fonction affine alors $\displaystyle\int_u^v f(t)dt = \frac{v-u}{2}\big(f(u)+f(v)\big)$.

On approche donc $\displaystyle\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t$ par $\displaystyle(a_{k+1}-a_k)\frac{f(a_k)+f(a_{k+1})}{2}$ d'où

$$I \simeq T_n = \frac{b-a}{n}\sum_{k=0}^{n-1}\frac{f(a_k)+f(a_{k+1})}{2} = \frac{b-a}{n}\left(\frac{f(a)}{2} + \sum_{k=1}^{n-1} f(a_k) + \frac{f(b)}{2}\right)$$

Programme XIII.2 – Méthode des trapèzes

```
def trapeze(f, a, b, n):
    pas = (b - a)/n
    somme = 0
    for i in range(n):
        somme = somme + (f(a + i*pas) + f(a + (i+1)*pas))/2
    return somme*pas
```

Exercice XIII.3 — Mesure de l'erreur
On suppose que f est de classe $\mathcal{C}^2$ sur $[a;b]$ et on note $M_2 = \sup\big\{|f''(t)| \ ; \ a \leqslant t \leqslant b\big\}$.
Prouver que $\displaystyle\left|\int_{a_k}^{a_{k+1}} f(t)dt - (a_{k+1}-a_k)\frac{f(a_k)+f(a_{k+1})}{2}\right| \leqslant \frac{M_2}{12}(a_{k+1}-a_k)^3$.
En déduire qu'on a $\big|I - T_n\big| \leqslant \dfrac{M_2.(b-a)^3}{12n^2}$.

Cette erreur ne prend en compte que l'erreur mathématique, si on ajoute les erreurs d'arrondis, l'erreur totale est de l'ordre de $\frac{\alpha}{n^2} + \beta.n.10^{-16}$. Il ne sert donc à rien de prendre des valeurs trop grandes pour n.

4 Compléments hors-programme : méthode de Simpson

On peut continuer dans la démarche et approcher f sur $[a_k; a_{k+1}]$ par une fonction polynomiale de degré 2. On choisit une fonction qui prend les mêmes valeurs que f aux points a_k, a_{k+1} et $m_k = \frac{1}{2}(a_k + a_{k+1})$.

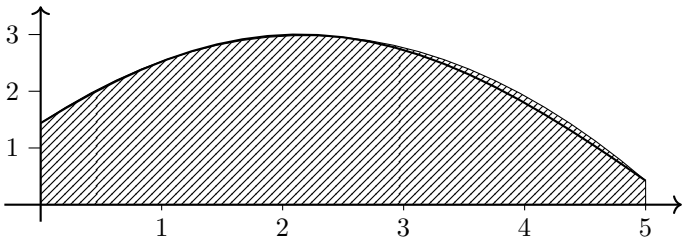


Figure XIII.4 – Méthode de Simpson, n = 1

Exercice XIII.4
Prouver que si f est une fonction polynomiale de degré 2 alors

$$\int_u^v f(t)dt = \frac{v-u}{6}\left(f(u) + 4f\left(\frac{u+v}{2}\right) + f(v)\right)$$

La méthode de Simpson consiste donc à faire l'approximation, en notant $m_k = \dfrac{a_k + a_{k+1}}{2}$,

$$\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t \simeq (a_{k+1} - a_k)\frac{f(a_k) + 4f(m_k) + f(a_{k+1})}{6}$$

On note S_n l'approximation obtenus ainsi pour I.

Exercice XIII.5
Écrire une fonction `Simpson(f, a, b, n)` *car approche l'intégrale de f entre a et b en utilisant cette méthode.*

On prouve que l'erreur commise est, si f est de classe $\mathcal{C}^4$, de l'ordre de $\frac{K}{n^4}$.
En raison des erreurs d'arrondis, on se restreindra à $n \leqslant 1000$.

Exercice XIII.6 — Lien entre les expressions
Prouver qu'on a

- $R_{n,d} = R_{n,g} + \dfrac{b-a}{n}\big(f(b) - f(a)\big)$
- $T_n = \dfrac{R_{n,d} + R_{n,g}}{2}$
- $T_n = R_{n,g} + \dfrac{b-a}{2n}\big(f(b) - f(a)\big)$
- $T_n = R_{n,d} + \dfrac{b-a}{2n}\big(f(a) - f(b)\big)$
- $S_n = \dfrac{4T_{2n} - T_n}{3}$

5 Solutions

Solution de l'exercice XIII.1 - $\displaystyle\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t - (a_{k+1}-a_k)f(a_k) = \int_{a_k}^{a_{k+1}} \big(f(t)-f(a_k)\big)\mathrm{d}t.$
Or l'inégalité des accroissement finis donne $\big|f(t)-f(a_k)\big| \leqslant |t-a_k|M_1$ donc on a
$\displaystyle\left|\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t - (a_{k+1}-a_k)f(a_k)\right| \leqslant \int_{a_k}^{a_{k+1}} M_1(t-a_k)\mathrm{d}t = \left[\frac{M_1}{2}(t-a_k)^2\right]_{a_k}^{a_{k+1}} = \frac{M_1}{2}(a_{k+1}-a_k)^2.$
En sommant, on obtient
$\displaystyle\left|\int_a^b f(t)\mathrm{d}t - R_{g,n}\right| \leqslant \sum_{k=0}^{n-1} \frac{M_1}{2}\left(\frac{b-a}{n}\right)^2 = \frac{M_1}{2}\frac{(b-a)^2}{n}$

Solution de l'exercice XIII.2 - $f(t)=\alpha t+\beta$.

$$\begin{aligned}\int_u^v f(t)\mathrm{d}t &= \int_u^v (\alpha t+\beta)\mathrm{d}t = \left[\alpha\frac{t^2}{2}+\beta t\right]_u^v = \alpha\frac{v^2-u^2}{2}+\beta(v-u)\\ &= \frac{v-u}{2}\big(\alpha u+\alpha v+2\beta\big) = \frac{v-u}{2}\big(f(u)+f(v)\big)\end{aligned}$$

Solution de l'exercice XIII.3 -

1. On pose $g(x) = \displaystyle\int_{a_k}^{x} f(t)\mathrm{d}t - (x-a_k)\frac{f(a_k)+f(x)}{2} + A(x-a_k)^3$ avec A choisi tel que $g(a_{k+1}) = 0$.
2. On a $g(a_{k+1}) = g(a_k) = 0$ donc il existe $c_1 \in]a_k; a_{k+1}[$ tel que $g'(c_1) = 0$.
3. $g'(x) = f(x) - (x-a_k)\frac{f'(x)}{2} - \frac{f(a_k)+f(x)}{2} + 3A(x-a_k)^2 = \frac{f(x)-f(a_k))}{2} - (x-a_k)\frac{f'(x)}{2} + 3A(x-a_k)^2$
4. $g'(a_k) = g'(c_1) = 0$ donc il existe $c_2 \in]a_k; c_1[$ tel que $g''(c_2) = 0$.
5. $g''(x) = -(x-a_k)\frac{f''(x)}{2} + 6A(x-a_k)2$ donc $g''(c_2) = 0$ avec $c_2 - a_l \neq 0$ donne $A = \frac{1}{12}f''(c_2)$
6. $g(a_{k+1}) = 0$ donne $\displaystyle\left|\int_{a_k}^{a_{k+1}} f(t)\mathrm{d}t - (a_{k+1}-a_k)\frac{f(a_k)+f(a_{k+1})}{2}\right| = \frac{(a_{k+1}-a_k)^3}{12}|-f''(c_2)| \leqslant \frac{M_2}{12}(a_{k+1}-a_k)^3$

En sommant, on obtient $\displaystyle\left|\int_a^b f(t)\mathrm{d}t - T_n\right| \leqslant \sum_{k=0}^{n-1}\frac{M_2}{12}\left(\frac{b-a}{n}\right)^3 = \frac{M_2}{12}\frac{(b-a)^3}{n^2}$

Solution de l'exercice XIII.4 - $f(t)=At^2+Bt+C$.

$$\begin{aligned}\int_u^v f(t)\mathrm{d}t &= \int_u^v (At^2+Bt+C)\mathrm{d}t = \left[A\frac{t^3}{3}+B\frac{t^2}{2}+Ct\right]_u^v = A\frac{v^3-u^3}{3}+B\frac{v^2-u^2}{2}+C(v-u)\\ &= \frac{v-u}{6}\big(Au^2+A(u^2+2uv+v^2)+Av^2+Bu+2B(u+v)+Bv+C+4C+C\big)\\ &= \frac{v-u}{6}\left(f(u)+4f\left(\frac{u+v}{2}\right)+f(v)\right)\end{aligned}$$

En fait l'égalité est vraie aussi pour les polynômes de degré 3.
On peut le prouver en la vérifiant pour la fonction $t \mapsto (2t-a-b)(t-a)(t-b)$.

Solution de l'exercice XIII.5 - $S_n = \dfrac{b-a}{6n}\displaystyle\sum_{k=0}^{n-1}\big(f(a_k)+4fm_k)+f(a_{k+1})\big)$.

```
def Simpson(f, a, b, n):
    pas = (b - a)/n
    somme = 0
    for i in range(n):
        u = f(a + i*pas)
        v = f(a + (i+1/2)*pas)
        w = f(a + (i+1)*pas)
        somme = somme + (u + 4*v + w)/6
    return somme*pas
```

Solution de l'exercice XIII.6 -

On note $h = \dfrac{b-a}{n}$ et $K_n = h\displaystyle\sum_{k=1}^{n-1} f(a_k)$. On a alors $R_{n,g} = hf(a) + K_n$, $R_{n,d} = K_n + hf(b)$ et $T_n = \frac{h}{2}f(a) + K_n + \frac{h}{2}f(b)$ ce qui donne les 4 premières égalités.

On note $h' = \dfrac{b-a}{2n} = \dfrac{h}{2}$ le pas de la subdivision pour $2n$

et a'_k les points correspondants : $a'_k = a + kh'$. On a $a'_{2k} = a + 2kh' = a + kh = a_k$

et $a'_{2k+1} = a + (2k+1)h' = a + kh + h' = a_k + \dfrac{h}{2} = a_k + \dfrac{a_{k+1}-a_k}{2} = \dfrac{a_{k+1}+a_k}{2} = m_k$.

$$\begin{aligned}\text{On a } S_n &= \frac{h}{6}\sum_{k=0}^{n-1}\big(f(a_k)+4f(m_k)+f(a_{k+1})\big) = \frac{h}{6}\sum_{k=0}^{n-1} f(a_k) + \frac{2h}{3}\sum_{k=0}^{n-1} f(m_k) + \frac{h}{6}\sum_{k=0}^{n-1} f(a_{k+1}) \\ &= \frac{1}{6}R_{n,g} + \frac{2h}{3}\sum_{p=0}^{n-1} f(a'_{2k+1}) + \frac{1}{6}R_{n,d} = \frac{1}{3}T_n + \frac{2h}{3}\sum_{p=0}^{n-1} f(a'_{2k+1})\end{aligned}$$

$$\begin{aligned}T_{2n} &= \frac{h'}{2}f(a) + h'\sum_{k=1}^{2n-1} f(a'_k) + \frac{h'}{2}f(b) = \frac{h}{4}f(a) + \frac{h}{2}\sum_{p=0}^{n-1} f(a'_{2p+1}) + \frac{h}{2}\sum_{p=1}^{n-1} f(a'_{2p}) + \frac{h}{4}f(b) \\ &= \frac{h}{2}\sum_{p=0}^{n-1} f(m_p) + \frac{h}{4}f(a) + \frac{h}{2}\sum_{p=1}^{n-1} f(a_p) + \frac{h}{4}f(b) = \frac{h}{2}\sum_{p=0}^{n-1} f(a'_{2p+1}) + \frac{1}{2}T_n\end{aligned}$$

On peut combiner : $S_n - \frac{4}{3}T_{2n} = \frac{1}{3}T_n - \frac{2}{3}T_n = -\frac{1}{3}T_n$.

Chapitre XIV

ÉQUATIONS DIFFÉRENTIELLES

Depuis Newton les lois de la physique sont souvent écrites sous formes d'équations reliant des fonctions à déterminer et leurs dérivées.
Nous allons, dans ce chapitre, préciser les notions d'équations différentielles générales et de solutions, particulièrement de solution approchée et donner un algorithme simple qui permet de déterminer des solutions approchées.
Le but sera de résoudre de manière approchée la plupart des équations, pas de résoudre mathématiquement un petit nombre d'équations.

1 Présentation

1.1 Introduction

On appelle équation différentielle une relation entre une ou plusieurs fonctions inconnues et leurs dérivées.
L'*ordre* d'une équation différentielle correspond au degré maximal de dérivation auquel l'une des fonctions inconnues a été soumise.
Voici quelques exemples.

Radioactivité On note $N(t)$ le nombre de noyaux radioactifs à l'instant t, c'est-à-dire se désintégrant selon un processus aléatoire.
Il existe un nombre λ dit *constante radioactive* tel que $N'(t) = -\lambda N(t)$.
L'ordre est ici 1. On peut écrire $N'(t) = f\big((t, N(t)\big)$ avec $f : (t, u) \mapsto -\lambda u$.

Oscillations libres Un ressort exerce une force opposée et proportionnelle à l'allongement du ressort $F = -kx$.
Si on néglige les frottements, l'équation $m\vec{a} = \sum \vec{F}$ donne $mx'' = -kx$.
En notant $\omega_0 = \sqrt{\frac{k}{m}}$ on peut aussi écrire l'équation sous la forme $x'' + \omega_0^2 x = 0$: c'est une équation d'ordre 2.

Pendule L'équation du pendule pesant peut s'écrire $\theta'' = \dfrac{g}{l}\sin(\theta)$ où θ est l'angle avec la verticale et l est la longueur. C'est une équation d'ordre 2 et elle n'est pas linéaire : l'expression n'est pas linéaire en la fonction inconnue et ses dérivées.

Nous allons nous restreindre dans ce chapitre au cas des équations d'ordre 1. [1] et résolues en y'.

1. Nous verrons que cette limite est surmontable.

1.2 Définitions

Nous traitons donc des équations de la forme

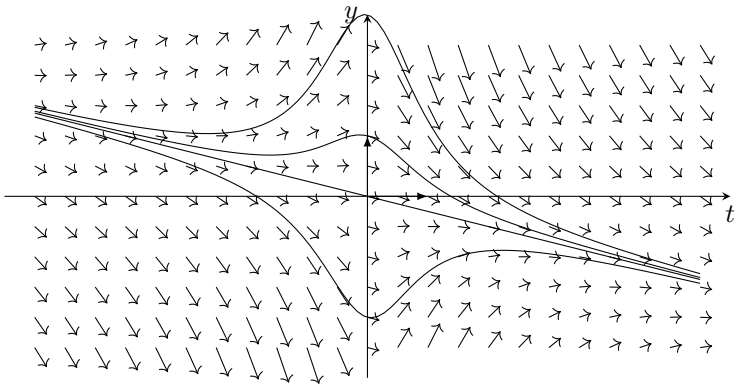
$$(E) \qquad y' = \varphi(y,t)$$

où φ est définie et continue sur une partie de $\mathbb{R}^2$ de la forme $U =]a;b[\times]c;d[$ [2] et à valeur dans $\mathbb{R}$.

Une **solution** de (E) est la donnée d'un intervalle I de $\mathbb{R}$ et d'une fonction f de classe $\mathcal{C}^1$ sur I tels que

- $\big(t,\varphi(t)\big) \in \mathcal{U}$ pour tout $t \in I$,
- $f'(t) = \varphi\big(f(t),t\big)$ pour tout $t \in I$.

Géométriquement cela signifie qu'une solution est telle que sa tangente en un point $\big(t,f(t)\big)$ admet $\varphi\big(f(t),t\big)$ pour pente.



Une équation aura beaucoup de solutions possibles. On spécialise l'équation (E) en ajoutant des **conditions initiales**, c'est un couple $(t_0,y_0) \in U$ et on cherche une solution passant par ce point : une solution doit vérifier $f(t_0) = y_0$. On obtient un **problème de Cauchy** :

$$(E_0) \qquad \begin{cases} y' = \varphi(y,t) \\ y(t_0) = y_0 \end{cases}$$

Si φ est suffisamment régulière, le théorème de Cauchy-Lipschitz affirme l'existence d'un intervalle ouvert I contenant t_0 et d'une unique fonction f de classe $\mathcal{C}^1$ sur I tels que (I,f) est une solution de (E_0).

Le problème que nous chercherons à résoudre sera de déterminer une approximation de la solution sur un intervalle $[t_0,t_f]$ (ou $[t_f;t_0]$) avec les conditions initiales (t_0,y_0). On admettra qu'une telle solution existe : en particulier nous supposerons que φ vérifie les conditions de régularité.

1.3 Vers la résolution

Que veut-on calculer ?

On a déjà vu une équation différentielle particulière : calculer l'intégrale $\displaystyle\int_a^b g(t)\mathrm{d}t$ revient à calculer la valeur en b de la solution de

$$(E_{int}) \qquad \begin{cases} y' = f(t) \\ y(a) = 0 \end{cases}$$

En effet, les solutions de $y' = f(t)$ sont les primitives de f, et si on choisit la primitive F de f qui s'annule en a, on a $\displaystyle\int_a^b g(t)\mathrm{d}t = F(b)$.

2. Plus généralement sur un ouvert de $\mathbb{R}^2$.

Lors du calcul de l'intégrale on n'a renvoyé que cette valeur mais lors du calcul, on a calculé un grand nombre de valeurs intermédiaires qui permettent en fait de calculer les valeurs approchées de $\int_a^{a_k} f(t)\mathrm{d}t$ avec $a_k = a + k\dfrac{b-a}{n}$.
Si on garde ces valeurs dans une liste, on obtient une liste de valeurs approchées des $F(a_k)$.
Voici une possibilité avec la méthode des trapèzes.

Programme XIV.1 – Valeurs d'une primitive

```
def primitive(f, a, b, n):
    """Entree : une fonction f,
                deux bornes,
                un entier
       Sortie : deux listes,
                la première contient les points de la
                   subdivision,
                la seconde, les valeurs de la primitive en ces
                    points"""
    pas = (b-a)/n
    T = [0]*(n+1)        # liste des points de subdivision
    F = [0]*(n+1)        # liste des valeurs de la primitive
    T[0] = a             # initialisation
    F[0] = 0             # initialisation
    for k in range(n):   # il reste n valeurs
        T[k+1] = a + (k+1)*(pas
        F[k+1] = F[k] + pas*(f(T[k]) + f(T[k+1]))/2
    return T, F
```

On va garder ce même principe mais, au lieu de donner les paramètres a, b et n, nous allons utiliser comme paramètre la liste des points en lesquels on veut une valeur approchée de la solution. Les méthodes renverront simplement la liste des valeurs en ces points.

1. On définit une liste `T` de taille n avec $t_0 =$ `T[0]` et $t_f =$ `T[n-1]`
2. On cherche une liste `Y` de taille n aussi telle que `Y[k]` est une valeur approchée de $f\big($`T[k]`$\big)$

Remarques

1. Le plus souvent la suite des valeurs de `T` sera régulièrement espacée mais ce ne sera pas imposé.
2. Les suites `T` et `Y` permettent de visualiser le graphe de la solution.

```
plt.plot(T, Y)
plt.show()
```

La représentation trace les segments délimités par les points de coordonnées `(T[k], Y[k])`.

2 Quelques méthodes

On va écrire des fonctions de résolution de la forme

```
def solution(phi, y0, T):
    ...
    return Y
```

On rappelle que l'on cherche à approcher la solution de

$$(E_0) \qquad \begin{cases} y' = \varphi(y,t) \\ y(t_0) = y_0 \end{cases}$$

sur un intervalle $[t_0; t_f]$ ou $[t_f; t_0]$.

2.1 Paramètres utilisés

`phi` : fonction de l'équation On doit donner la fonction qui définit l'équation différentielle.
Exemple : l'équation $\tau y'(t) + y(t) = Ke_0$ soit : $y'(t) = \frac{Ke_0 - y(t)}{\tau}$ pourra s'écrire

```
def phi(y,t):
    K = 2.5
    e0 = 1.7
    tau = 0.25
    return (y - K*e0)/tau
```

On remarquera que les variables sont `y` suivie de `t` ; c'est la convention prise dans les modules qu'on utilisera parfois, en particulier `scipy.integrate`.
Il arrivera très souvent dans les équations qui seront utilisées que la variable t n'apparaisse pas explicitement dans φ, comme ici. Il faudra néanmoins introduire t comme paramètre.

`y0` : condition initiale La valeur de t_0 est donnée par la première valeur de `T`. Il reste donc à donner la valeur initiale du problème de Cauchy : y_0.

`T` : liste des temps L'intervalle de temps est discrétisé en une suite finie de temps. On notera t_k la valeur de `T[k]`. Le plus souvent les valeurs de la liste seront séparées par un pas constant : $t_k = t_0 + k\frac{t_f - t_0}{n-1} = t_0 + k(t_1 - t_0)$ si n est la longueur de `T`.

`Y` : résultat C'est la liste des valeurs approchées des $f(t_k)$. On note $y_k =$ `Y[k]`.

2.2 Principe

Pour définir la suite Y à partie de T, y_0 et φ nous allons procéder pas-à-pas.

1. On remplace l'équation différentielle par une équation intégrale
   $f(t_{k+1}) = f(t_k) + \displaystyle\int_{t_k}^{t_{k+1}} f'(t)\mathrm{d}t = f(t_k) + \int_{t_k}^{t_{k+1}} \varphi\big(f(t),t\big)\mathrm{d}t.$
2. On calcule, pour chaque intégrale, une valeur approchée $\displaystyle\int_{t_k}^{t_{k+1}} \varphi\big(f(t),t\big)dt \simeq \delta_k$.
3. On définit $y_{k+1} = y_k + \delta_k$ pour $0 \leqslant k < N$.

Les différentes méthodes correspondent à différents algorithmes d'approximation d'une intégrale.

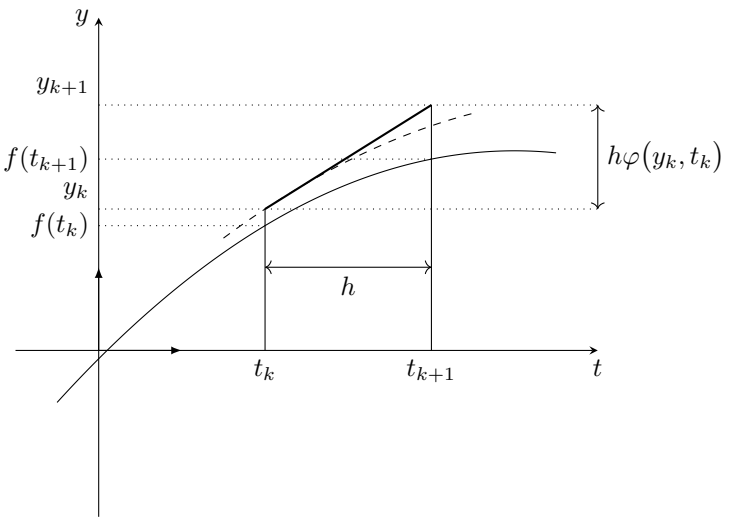
2.3 Méthode d'Euler explicite

Elle correspond à la méthode des rectangles à gauche.

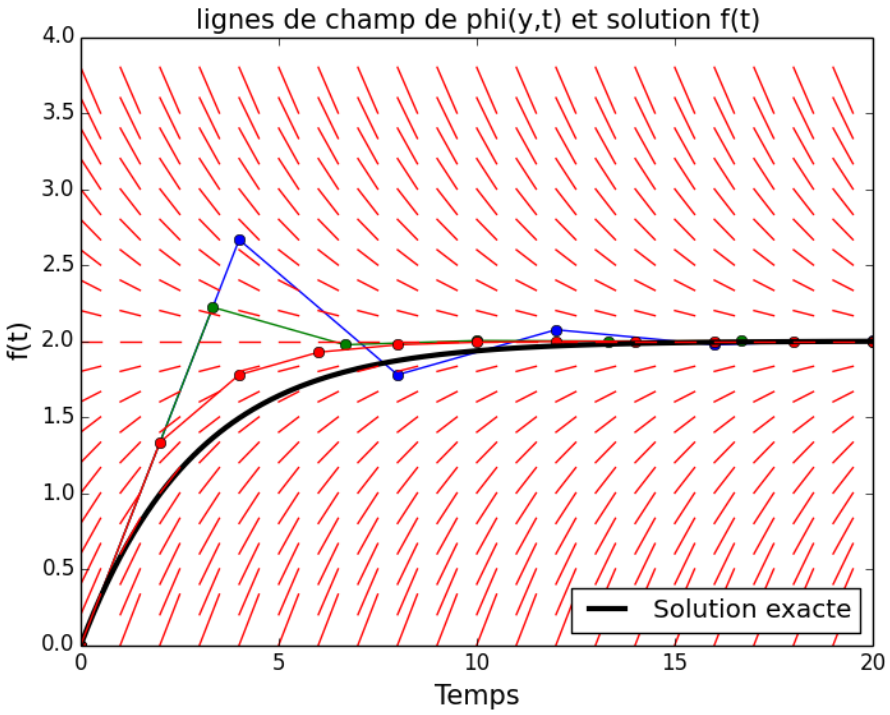
$$\int_{t_k}^{t_{k+1}} \varphi\big(f(t),t\big)\mathrm{d}t \simeq (t_{k+1} - t_k)\varphi\big(f(t_k),t_k\big)$$

On connaît une valeur approchée de $f(T_k)$, c'est y_k. Ainsi on aboutit à

$$y_{k+1} = y_k + (t_{k+1} - t_k)\varphi\big(y_k,t_k\big)$$



On peut remarquer que $t \mapsto y_k + (t - t_k)\varphi(y_k, t_k)$ est l'équation de la tangente en (t_k, y_k) de la solution de condition initiales t_k et y_k.
La méthode revient à suivre la tangente des solutions sur des intervalles $[t_k; t_{k+1}]$.
On approche donc la solution par une fonction affine sur chaque intervalle.



Programme XIV.2 – Euler explicite

```
def euler(f, y0, T):
    """Entree : une fonction f de 2 variables
                un reel y0, la condition initiale en t0
                une liste de réels, monotone, dont le premier
                   terme est t0
       Sortie : une liste de  points définissant
                  une solution approchee de y' = f(y, t)
                  avec les conditions initiales (t0,y0)
                  y(T[i]) est approché par Y[i]"""
    n = len(T)            # n valeurs à calculer
    Y = [0]*n             # création de la liste
    Y[0] = y0             # ordonnee initiale
    for k in range(n-1): # il reste (n-1) points à trouver
        pas = T[k+1] - T[k]
        pente = f(Y[k],T[k])
        Y[k+1] = Y[k] + pas*pente # On applique la formule
    return Y
```

On notera qu'on calcule le pas à chaque étape : il n'est peut-être pas constant.

2.4 Méthode d'Euler implicite

Si on approche la fonction à intégrer, $\varphi\big(f(u),u\big)$, par sa valeur finale sur l'intervalle $[t_k;t_{k+1}]$ c'est-à-dire $\varphi\big(f(u),u\big) \simeq \varphi\big(f(t_{k+1}),t_{k+1}\big) \simeq \varphi\big(y_{k+1},y_{k+1}\big)$ on obtient

$$y_{k+1} = y_k + \int_{y_k}^{y_{k+1}} \varphi\big(y_{k+1},y_{k+1}\big)\mathrm{d}t \simeq y_k + (t_{k+1}-t_k)\varphi\big(y_{k+1},t_{k+1}\big)$$

On voit que y_{k+1} apparaît aussi dans le second membre : il est défini implicitement c'est-à-dire qu'on ne peut pas le calculer directement.
Pour déterminer y_{k+1} en fonction de y_k il faut donc résoudre, à chaque étape, l'équation $g_k(y)=0$ avec $g_k(y) = y - y_k - (t_{k+1}-t_k)\varphi(y,t_{k+1})$.
On peut, par exemple, en déterminer une valeur approchée avec des méthodes d'analyse numérique comme celle de Newton.

Parfois on peut résoudre directement l'équation. Par exemple si on veut résoudre l'équation différentielle $y' = y^2+t^2$ pour $y(t_0)=y_0$ sur $[t_0;t_1]$ par la méthode d'Euler implicite, on doit doit calculer les y_k avec $y_{k+1} = y_k + h\big(y_{k+1}^2 + t_{k+1}^2\big)$ où h est le pas supposé constant.
Ainsi y_{k+1} est solution de $hX^2 - X + \big(y_k + ht_{k+1}^2\big) = 0$.
Les racines sont $\dfrac{1 \pm \sqrt{1-4hy_k-4h^2t_{k+1}^2}}{2h}$; on choisit celle qui est proche de y_k

$$Y_{k+1} = \frac{1-\sqrt{1-4hy_k-4h^2t_{k+1}^2}}{2h}$$

```
def solution_particuliere(y0, T):
    n = len(T)
    Y = [0]*n
    Y[0] = y0
    for k in range(n-1):
        pas = T[k+1] - T[k]
        delta = 1 - 4*pas*(Y[k]+pas*T[k+1]**2)
        Y[k+1] = (1-delta**(1/2))/(2*h)
    return Y
```

Sans rentrer dans la théorie, on peut retenir que la méthode implicite est souvent moins précise et plus compliquée à utiliser mais elle est plus stable que la méthode explicite, elle diverge moins rapidement. Pour simplifier grossièrement : la méthode implicite est moins précise à court terme mais plus précise à long terme.

Utilisation de la méthode de Newton.

On rappelle la méthode de Newton

```
def newton(f, df, x0, epsilon = 1e-8):
    h = 1 + epsilon
    x = x0
    while h > epsilon:
        x_old = x
        x = x - f(x)/df(x)
        h = abs(x - x_old)
    return x
```

L'équation à résoudre est $f(y) = y - y_k - (t_{k+1} - t_k)\varphi(y, t_{k+1}) = 0$.

On a $f'(y) = 1 - (t_{k+1} - t_k)\dfrac{\partial\varphi(y, t_{k+1})}{\partial y}$: il faudra donc définir aussi la fonction python associée à la dérivée partielle.

Par exemple pour $y' = \dfrac{\sin(y)}{t^2 + 0.0001}$ on écrit

```
def phi(y, t):
    return -sin(y)/(t**2+0.001)

def dphi(y, t):
    return -cos(y)/(t**2+0.001)
```

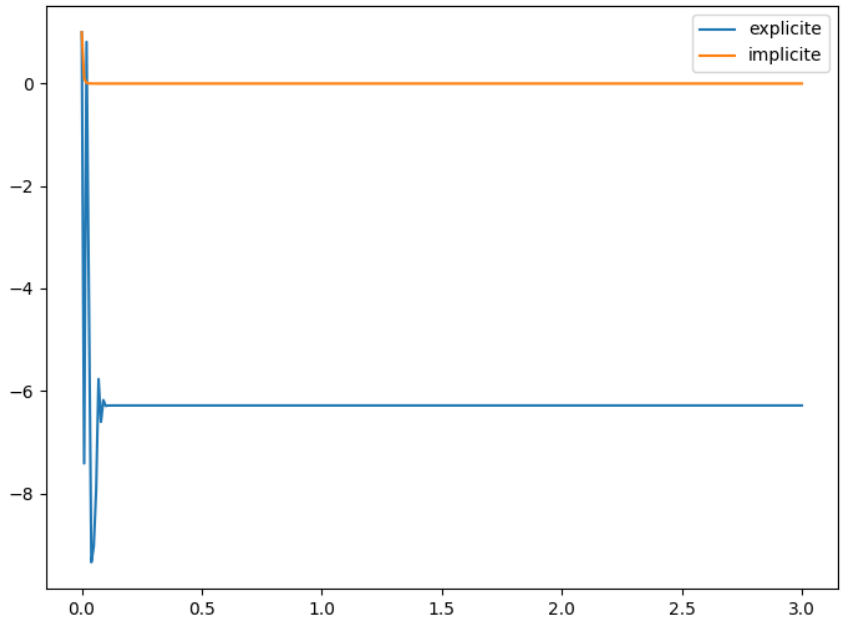
La fonction a été choisie pour mettre en évidence l'amélioration de la stabilité par la méthode implicite. La solution attendue est celle fournie par la méthode implicite.

Les fonctions g et g' dépendent des valeurs calculées t_{k+1} et y_k : on les définira à l'intérieur de la fonction de résolution, cela est possible dans python.

On prendra y_k pour valeur initiale de la recherche, c'est une valeur proche de y_{k+1}.

Programme XIV.3 – Euler implicite

```
1  def euler_imp(phi, dphi, y0, T):
2      n = len(T)
3      Y = [0]*n
4      Y[0] = y0
5      for k in range(N-1):
6          pas = T[k+1] - T[k]
7          def f(y):
8              return y - Y[i] - pas*phi(y, T[k+1])
9          def df(y):
10             return 1 - pas*dphi(y, T[k+1])
11         Y[k+1] = newton(f, df, Y[k])
12     return Y
```



2.5 Utilisations du module `scipy`

2.6 odeint

Le module `scipy` contient, dans la sous-bibliothèque `integrate` une fonction `odeint` qui résout efficacement les équations différentielles. C'est la méthode à utiliser quand la résolution d'une équation est un outil d'un projet plus large.

```
from scipy.integrate import odeint

odeint(phi,y0,T)
```

2.7 fsolve

Dans le cas de la méthode d'Euler implicite la résolution de l'équation implicite peut se faire avec la fonction `fsolve` (voir **XII.3**).
L'usage des paramètres envoyés permet de ne définir qu'une seule fois une fonction.

Programme XIV.4 – Euler implicite avec `fsolve`

```
1  from scipy.optimize import fsolve
2
3  def euler_imp(phi, y0, T):
4      def equ(y, y_old, t):
5          return y - y_old - pas*phi(y_old, t)
6      n = len(T)
7      Y = [0]*n
8      Y[0] = y0
9      for k in range(N-1):
10         pas = T[k+1] - T[k]
11         Y[k+1] = fsolve(equ, Y[k], args = (Y[k], T[k+1]))
12     return Y
```

Chapitre XV

Module Numpy

1 Listes Python

1.1 Avantages

Les liste python sont un type de données très souple :

- on accède aux éléments directement aux éléments par leur indice,
- les valeurs sont modifiables,
- on peut combiner plusieurs types de données dans les valeurs, entiers, booléens, flottants, ...
- on peut ajouter des éléments à une extrémité pour un coût faible : `append`,
- on peut combiner des listes en les ajoutant,
- on peut répéter une liste en la multipliant par un entier positif,
- on peut extraire une portion de liste : `liste[a : b]`,
- on peut itérer sur une liste ; `for x in liste:`,
- il existe de nombreuses méthodes de modification : `pop`, `remove`, `del`, ...

Comme exemple de cette souplesse, il existe au moins 3 façons d'appliquer une fonction à une liste.

```
def appliquer1(f, liste):
    n = len(liste)
    out = [0]*n
    for i in range(n):
        out[i] = f(liste[i])
    return out
```

```
def appliquer2(f, liste):
    n = len(liste)
    out = []
    for i in range(n):
        out.append(f(liste[i]))
    return out
```

```
def appliquer3(f, liste):
    return [f(x) for x in liste]
```

1.2 Inconvénients

Cette richesse fonctionnelle ne va pas sans quelques problèmes.

- Les méthodes `pop`, `remove`, `del` et autres ont un coût caché qui rend leur utilisation dangereuse en raison même de leur simplicité apparente.
- La richesse de la structure rend le temps de calcul sur des grandes listes plus lent que dans d'autres langages.
- La création d'une liste neutre de n éléments ne semble pas "naturelle".
- On pourrait souhaiter que l'application d'une fonction à une liste s'écrive simplement `f(liste)`.

Cependant ce qui manque le plus pour un usage scientifique de Python est l'inaptitude des listes à représenter simplement des vecteurs. On aura souvent besoin de considérer les éléments d'une liste comme les composantes d'un vecteur auquel on voudrait appliquer les lois d'un espace vectoriel : addition de deux vecteurs et multiplication par un scalaire.

2 Le module Numpy

2.1 Présentation

Il existe un module qui permet d'utiliser des listes de manière plus efficace dans le cadre du calcul scientifique : `numpy`. C'est un ensemble de fonctions centrées autour d'un nouveau type de données, des tableaux nommés `array`.
Ce type présente des limitations par rapport aux listes de python :

- la longueur est fixée, on ne peut pas utiliser la méthode `append`,
- le type des composantes est unique et fixé à l'avance, tous les éléments seront soit des flottants, soit des entiers.

En contrepartie il y a des avantages :

- les tableaux peuvent être additionnés terme-à-terme comme des vecteurs,
- on peut appliquer une fonction à un tableau, cela applique la fonction à tous les termes,
- en particulier on peut multiplier un tableau par un scalaire, cela multiplie chaque composante,
- les calculs sont plus rapides.

L'accès aux éléments, la longueur et l'extraction sont écrits de la même façon que dans le cas des listes python.

2.2 Premières fonctions

Le module doit être importé

```
import numpy as np
```

Il est recommandé de ne pas importer le module de manière anonyme (`from numpy import *`) et ne nom abrégé, `np`, est conventionnel.
Le module contient les fonctions mathématiques usuelles, il est préférable d'utiliser ces fonctions plutôt que celles du module `math`.

Création

Pour créer un tableau, plusieurs moyens sont possibles.

1. On peut convertir, par la fonction `np.array`, une liste python

```
>>> m = np.array([1, 2, 5, 3])
>>> m
array([1, 2, 5, 3])
```

   L'affichage par `print` supprime les virgules,

```
>>> print(m)
[1 2 5 3]
```

2. On peut créer un tableau de valeurs nulles avec `np.zeros` :

```
>>> np.zeros(4)
array([ 0.,  0.,  0.,  0.])
```

   On remarquera que les valeurs sont des flottants.
3. On peut forcer le type des valeurs avec le paramètre optionnel `dtype`

```
>>> np.zeros(4, dtype = int)
array([ 0,  0,  0,  0])
```

   On peut choisir des types entiers spécifiques, signés/non signés, sur 8, 16, 32 ou 64 bits.
4. On peut créer un tableau de valeurs unitaires avec `np.ones` :

```
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
```

5. On notera que le type des données est contraignant

```
>>> m =  np.zeros(4, dtype = int)
>>> m[1] = 2.6
>>> m
array([0, 2, 0, 0])
```

   Lorqu'un tableau est crée avec des valeurs entières alors qu'on veut faire des calculs flottants, on devra mettre au moins un des termes sous forme flottante.

```
>>> m = np.array([1.0, 2, 5, 3])
>>> m
array([1., 2., 5., 3.])
```

6. La fonction `np.linspace(a, b, n)` crée un tableau de taille n avec des valeurs de a à b également espacées.

```
>>> np.linspace(0, 1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

7. La fonction `np.arange(a, b, r)` crée le tableau des valeurs qui commencent par a, espacées de r et qui ne dépassent pas b (b est exclu).

```
>>> np.arange(0, 1, 0.2)
array([0. , 0.2, 0.4, 0.6, 0.8])

>>> np.arange(0, 1, 0.3)
array([0. , 0.3, 0.6, 0.9])
```

8. La fonction `np.copy(a)` crée une copie indépendante de l'original.

Opérations

On utilisera les tableaux `a = np.array([1.2, 2.0, -1.4])` et `b = np.array([2.0, 4, 3])`.

1. La somme de deux tableaux **de même longueur** est le tableau des sommes.

```
>>> a + b
array([3.2, 6. , 1.6])
```

   La somme de deux tableaux de longueurs différentes engendre une erreur.
2. Le produit de deux tableaux **de même longueur** est le tableau des produit.

```
>>> a*b
array([ 2.4,  8. , -4.2])
```

   On peut aussi diviser un tableau par un autre.

3. On peut ajouter un scalaire à un tableau, on obtient le tableau dans lequel le scalaire a été ajouté à chaque terme.

```
>>> a + 2.7
array([3.9, 4.7, 1.3])
```

4. On peut calculer le tableau des puissances des termes d'un tableau

```
>>> a**2
array([1.44, 4.  , 1.96])
```

   On peut donc appliquer un polynôme à un tableau

```
def P(x):
    return x**3 - 2*x**2 + 7*x + 4

>>> P(a)
array([ 11.248,  18.   , -12.464])
```

5. Plus généralement on peut appliquer une **fonction** `numpy` à un tableau

```
>>> np.sin(a)
array([ 0.93203909,  0.90929743, -0.98544973])
```

   Cela est possible pour une fonction définie à partir des fonctions `numpy`.

```
def f(x):
    return 1 + 2.5*np.sin(3*x) - np.cos(np.pi*x)

>>> f(a)
array([ 0.70271589, -0.69853875,  3.48795643])
```

6. Une fonction définie avec des instructions conditionnelles ne s'applique pas directement ; elle n'est pas une fonction universelle (**`ufunc`**) pour `numpy`. On peut la transformer pour qu'elle puisse agir sur les tableaux `numpy` avec l'instruction `np.vectorize`.

```
def signe1(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1

>>> signe1(a)
...
ValueError: ...

signe = np.vectorize(signe1)

>>> signe(a)
array([ 1,  1, -1])
```

7. On peut affecter une valeur à tous les termes d'une extraction : **on modifie alors le tableau**.

```
>>> a[1:3] = 2.5
>>> a
array([1.2, 2.5, 2.5])
```

Chapitre XVI

ÉQUATIONS DIFFÉRENTIELLES : COMPLÉMENTS

Ce chapitre présente des développements du chapitre sur les équations différentielles qu'une lecture raisonnable du programme peut considérer comme hors-programme mais que les épreuves des concours et les calculs en T.I.P.E. utilisent couramment.
Le texte utilise la fonction `Euler` de résolution mais on peut lui substituer toute autre méthode vue dans le T.P. Il est en fait recommander d'utiliser la fonction `odeint` du module `scipy.integrate` qui se substitue efficacement aux fonctions écrites.

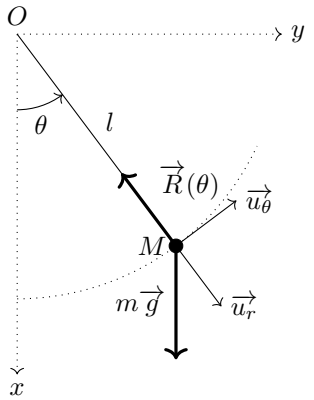
1 Équations d'ordre 2

1.1 Exemple

Le pendule pesant est soumis à la force de la pesanteur et à la réaction du fil ($\overrightarrow{R}$). La formule de Newton donne, en projetant dans une base orthonormée directe $(\overrightarrow{u_r}, \overrightarrow{u_\theta})$ telle que $\overrightarrow{OM} = l\overrightarrow{u_r}$,

$$-ml\left(\frac{d\theta}{dt}\right)^2 = -R + mg\cos(\theta) \qquad \text{(XVI.1)}$$

$$ml\frac{d^2\theta}{dt^2} = -mg\sin(\theta) \qquad \text{(XVI.2)}$$



On remarque que la la seconde équation, obtenue en projetant sur $\overrightarrow{u_\theta}$ ne fait intervenir que l'angle comme fonction du temps mais elle fait intervenir la dérivée seconde.
La première équation, obtenue en projetant sur $\overrightarrow{u_r}$, permet de calculer $R(t)$ si on connaît la fonction $\theta(t)$.

1.2 Un solution

Une équation différentielle d'ordre 2 est une équation qui exprime la dérivée seconde en fonction des valeurs de la fonction, de sa dérivée et de la variable.

$$y'' = \psi(y, y', t)$$

Dans l'exemple ci-dessus $\psi(y, y', t) = -\frac{g}{l}\sin(y)$.

Une propriété importante des équation différentielle d'ordre 2 est qu'une des conditions pour qu'existe une solution unique est qu'on doit donner des conditions initiales en t_0 pour la fonction et pour sa dérivée [1].
On peut alors résoudre de manière approchée l'équation avec des approximations d'Euler à 2 étages : on incluant le calcul des dérivées aux temps t_i.
On cherchera donc à définir deux listes `Y`, pour les valeurs de la fonctions, et `V`, pour les valeurs de sa dérivée. La valeur de `Y[i]`, y_i, et celle de `V[i]`, v_i seront des valeurs approchées de $f(t_i)$ et $f'(t_i)$ respectivement où f est la solution telle que $f(t_0) = y_0$ et $f'(t_0) = v_0$.
Si on note $h_i = t_{i+1} - t_i$, on calcule donc itérativement

$$y_{i+1} \simeq f(t_{i+1}) \simeq f(t_i) + \delta_i f'(t_i) \simeq y_i + h_i v_i \text{ et}$$

$$v_{i+1} \simeq f'(t_{i+1}) \simeq f'(t_i) + h_i f''(t_i) = f'(t_i) + h_i \psi\big(f(t_i), f'(t_i), t_i\big) \simeq v_i + h_i \psi(y_i, v_i, t_i)$$

```
def Euler2(psi, y0, v0, T):
    n = len(T)
    Y = [0]*n
    Y[0] = y0
    V = [0]*n
    V[0] = v0
    for i in range(n-1):
        pas = T[i+1] - T[i]
        Y[i+1] = Y[i] + pas*V[i]
        V[i+1] = V[i] + pas*psi(Y[i], V[i], T[i])
    return Y, V
```

Les deux listes `Y` et `V` permettent de tracer le **portrait de phase** où, au lieu de représenter y en fonction de t, on représente la trajectoire des points de coordonnées $\big(y(t), y'(t)\big)$.

1.3 Traitement de l'exemple

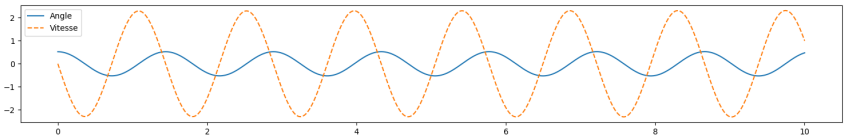
On revient à l'exemple $\theta''(t) = -\frac{g}{l}\sin\big(\theta(t)\big)$.

```
import numpy as np
import matplotlib.pyplot as plt

g = 9.81
l = 0.5
t_fin = 10 # temps final pour l'étude
N  = 100000  # nombre de points
T = np.linspace(0, t_fin, N)

def psi(y, v, t):
    return -g/l*np.sin(y)
```

1. Le cas des conditions aux limites, $y(t_0) = y_0$ et $y(t_{final}) = y_{final}$, plus difficile, ne sera pas traité ici.

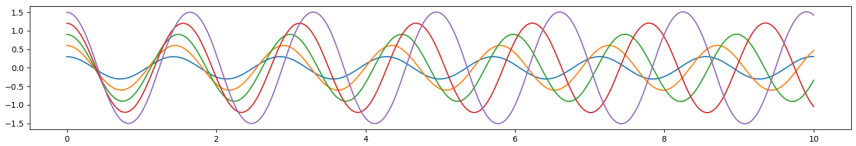
- On peut tracer une solution et la courbe des vitesses, par exemple pour $\theta_0 = \frac{\pi}{6}$ et $v_0 = 0$.

```
y0 = np.pi/6
v0 = 0
Y, V = Euler2(psi, y0, v0, T)

plt.plot(T, Y, label = "Angle")
plt.plot(T, V, linestyle = "dashed", label = "Vitesse")
plt.legend()
plt.show()
```



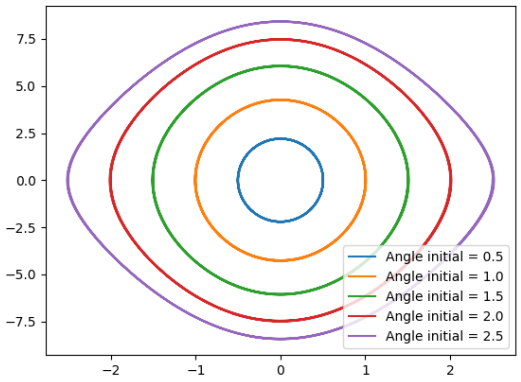
- On peut tracer plusieurs solution, on remarque que, contrairement à l'oscillateur harmonique, la période n'est pas constante.

```
Y0 = [0.3, 0.6, 0.9, 1.2, 1.5]
v0 = 0
for y0 in Y0:
    Y, V = Euler2(psi, y0, v0, T)
    plt.plot(T, Y)
plt.show()
```



- On peut tracer plusieurs portraits de phases.

```
Y0 = [0.5, 1.0, 1.5, 2.0, 2.5]
v0 = 0
for y0 in Y0:
    Y, V = Euler2(psi, y0, v0, T)
    plt.plot(Y, V)
plt.show()
```



2 Équations couplées

2.1 Exemple

Le fructose et le glucose sont deux sucres d'origine naturelle que l'on rencontre souvent en mélange. Leur dosage est important en particulier dans le sérum sanguin pour l'étude des maladies telles que le diabète. On propose ici d'étudier un dosage par une méthode cinétique fondée sur le fait que le glucose et le fructose réagissent à des vitesses différentes sur la plupart des réactifs.
Le fructose noté F ainsi que le glucose G réduisent le molybdène (VI) présent sous la forme d'ions molybdate MoO_4^{2-} en ions MoO_2^+ du molybdène (V) (bleu de molybdène) selon les réactions symbolisées :

$$\begin{aligned} \text{Mo(VI)} + \text{F} &\xrightarrow{k_F} \text{Mo(V)} + \text{F}' \\ \text{Mo(VI)} + \text{G} &\xrightarrow{k_G} \text{Mo(V)} + \text{G}' \end{aligned}$$

Les composés sont des solutés et on note $[X]$ la concentration molaire volumique de X.
Les concentration initiales et à l'instant t sont données par le tableau :

	[Mo(VI)]	[F]	[G]	[Mo(V)]	[F′]	[G′]
$t=0$	a	b	c	0	0	0
t	$a-x-y$	$b-x$	$c-y$	$x+y$	x	y

La température et le volume sont maintenus constants, on suppose que les réactions sont d'ordre 1 par rapport à chacun des réactifs.
Les constantes de vitesses permettent de définir les relations

$$\begin{cases} \dfrac{\mathrm{d[F']}}{\mathrm{d}t} = -\dfrac{\mathrm{d[F]}}{\mathrm{d}t} = \dfrac{\mathrm{d}x}{\mathrm{d}t} = k_F[\text{Mo(VI)}][\text{F}] = k_F(a-x-y)(b-x) \\ \dfrac{\mathrm{d[G']}}{\mathrm{d}t} = -\dfrac{\mathrm{d[G]}}{\mathrm{d}t} = \dfrac{\mathrm{d}y}{\mathrm{d}t} = k_G[\text{Mo(VI)}][\text{G}] = k_G(a-x-y)(c-y) \end{cases}$$

Les valeurs numériques, exprimées en $\mathrm{mol}\cdot\mathrm{L}^{-1}$ pour les concentrations et en $\mathrm{min}^{-1}\cdot\mathrm{mol}^{-1}\cdot\mathrm{L}$ pour les constantes de vitesse sont définies comme des variables globales.

```
a  = 12.0e-2
b  =  3.0e-2
c  =  5.0e-2
kF =  7.8
kG  = 2.0
```

On voit que les deux équations différentielles ne peuvent pas être résolues indépendemment : elles sont **couplées**, la dérivée de chacune des deux fonctions $x(t)$ et $y(t)$ dépend des deux fonctions.

2.2 Une solution sans nouvelle fonction

On pourrait, comme ci-dessus, utiliser la méthode d'Euler pour chaque équation en les résolvant en même temps : on devrait avoir pour paramètres les 2 équations et les deux conditions initiales. Pour les systèmes à 3, 4, ...équations on devrait alors écrire une méthode de résolution différente. Pour simplifier les choses on va utiliser la méthode de vectorisation : au lieu de considérer n équations avec chacune une inconnue, on considère une seule équation avec n inconnues. On fera plusieurs calculs en même temps en regroupant les différentes valeurs dans un **vecteur**.

Dans l'exemple ci-dessus on considère donc le vecteur $u(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$.

En dérivant le vecteur terme-à-terme on obtient

$$u'(t) = \begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix} = \begin{pmatrix} k_F\big(a-x(t)-y(t)\big)\big(b-x(t)\big) \\ k_G\big(a-x(t)-y(t)\big)\big(c-y(t)\big) \end{pmatrix} = \Phi\big(u(t),t\big)$$

Comme d'habitude on considère que la variable t peut intervenir dans la dérivée.

Les deux formules obtenues à partir de la méthode d'Euler :

$$x(t_{k+1}) \simeq x(t_k) + (t_{k+1} - t_k)x'(t_k)$$
$$y(t_{k+1}) \simeq y(t_k) + (t_{k+1} - t_k)y'(t_k)$$

deviennent $u(t_{k+1}) \simeq u(t_k) + (t_{k+1} - t_k)u'(t_k) = u(t_k) + (t_{k+1} - t_k)\Phi\big(u(t_k), t_k\big)$.
On est revenu à un schéma d'Euler simple. Il faut simplement pouvoir multiplier les vecteurs par un scalaire et additionner 2 vecteurs, c'est que permettent directementles tableaux `numpy`.
Il reste donc à adapter est fonction de l'équation différentielle qui doit recevoir comme première variable un tableau `numpy` et renvoyer un tableau `numpy` de même taille. L'ensemble des conditions initiales devra aussi être défini par un tableau `numpy`.
On utilisera ensuite la fonction `Euler` (ou `odeint`) du chapitre précédent.

2.3 Traitement de l'exemple

On commence par la fonction : elle déconstruit le vecteur et en construit un nouveau.

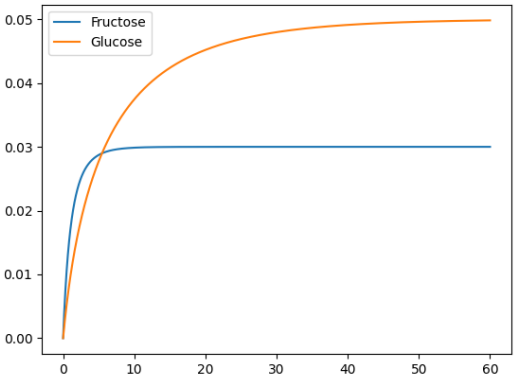
```
def phi(u, t):
    x, y = u
    dx = kF*(a - x - y)*(b - x)
    dy = kG*(a - x - y)*(c - y)
    return np.array([dx, dy])
```

Les conditions initiales sont regroupées dans un vecteur.

```
T = np.linspace(0, 60, 10000)
u0 = np.array([0, 0])
U = Euler(phi, u0, T)
```

Le résultat est une liste de vecteurs, on va définir les listes de composantes.

```
X = [u[0] for u in U]
Y = [u[1] for u in U]
plt.plot(T, X, label = "Fructose")
plt.plot(T, Y, label = "Glucose")
plt.legend()
plt.show()
```



2.4 Retour sur les équations d'ordre 2

La méthode vue dans le paragraphe 1.2 est utile car elle servira de cadre pour des amélioration adaptées aux équations d'ordre deux.
Cependant elle est dispensable ; le retour à une équation d'ordre 1 peut se faire. On a en fait considéré l'équation différentielle d'ordre 2 comme deux équation d'ordre 1 :

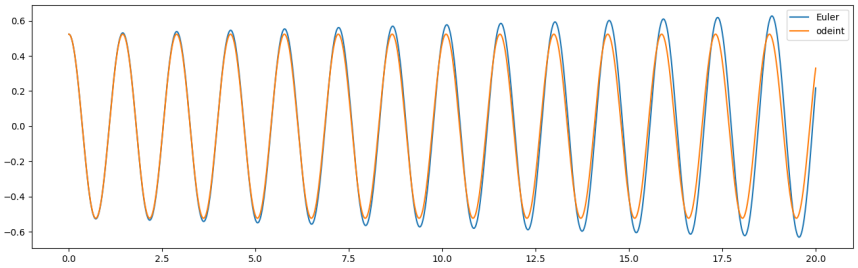
$$\begin{cases} y'(t) = v(t) \\ v'(t) = \psi\big(y(t), y'(t), t\big) \end{cases}$$

On peut donc se ramener au cas précédent en vectorisant.
L'équation considérée ce résout alors avec les instruction

```
def phi(u,t):
    y, v = u
    a = -g/l*np.sin(y)
    return np.array([v, a])

T = np.linspace(0, 20, 20000)
u0 = np.array([np.pi/6, 0])
U = Euler(phi, u0, T)
Y = [u[0] for u in U]
```

L'avantage est que l'on bénéficie de toutes les possibilités d'amélioration, en particulier de la fonction `odeint`.

```
U1 = odeint(phi, u0, T)
Y1 = [u[0] for u in U1]

plt.plot(T, Y, label = "Euler")
plt.plot(T, Y1, label = "odeint")
plt.legend()
plt.show()
```



Chapitre XVII

Bases de données

1 Présentation

Notre société utilise de plus en plus de données et un des enjeux de l'efficacité économique mais aussi de la démocratie est d'en maîtriser l'exploitation.
Nous allons décrire dans ce chapitre le modèle de description d'un ensemble de données, les opérations qu'on veut pouvoir effectuer et le langage universel qui permet ces opérations.

1.1 Données

Un ensemble de données est avant tout constitué d'information, l'enjeu est d'arriver à trouver une réponse depuis ces ensemble d'information. Quand les données ne sont pas bien structurées cela peut devenir une énigme, comme dans certains jeux.

Le dahut est une espèce très rare de bouquetin qui a la particularité d'avoir les deux pattes d'un coté plus courtes que les autres. Il vit donc dans la montagne en ayant toujours le sommet du coté de ses pattes courtes. Un dahut est appelé de dextrogyre (resp. lévogyre) si ses pattes gauches (resp. ses pattes droites) sont plus courtes, ce qui fait qu'il a le sommet de sa montagne sur sa droite (resp. sur sa gauche) et que donc il tourne toujours vers la droite (resp. vers la gauche).

Les dahuts ont d'autres particularités :

- *tout dahut non lévogyre a des rayures noires,*
- *tout dahut a des oreilles blanches ou n'a pas de rayures noires,*
- *les dahuts qui vivent dans les forêts ne mangent pas de mulots,*
- *un dahut mange des mulots si et seulement s'il est lévogyre,*
- *tout dahut qui a des oreilles blanches est lévogyre et vit dans les forêts,*
- *tout dahut lévogyre a des oreilles blanches.*

Prouver que les dahuts n'existent pas.

Pour organiser les données on va utiliser un modèle simple, celui d'un tableau.
Les données concernent plusieurs objets qui ont des propriétés, on se donne à l'avance un ensemble de caractéristiques et on donne, pour chaque objet, la valeur prise pour chaque caractéristique. On obtient un tableau à deux dimensions avec une ligne par objet et une colonne par caractéristique. Les colonnes sont nommées mais les objets sont définis simplement par leur ensemble de valeurs, s'ils ont un nom alors c'est une de leurs propriétés.

On rencontre souvent ce type de structures :

- répertoire (nom, téléphone, adresse, ...)
- fiche de bibliothèque (auteur, titre, année, pagination, ...)
- livre d'état-civil des naissances (père, mère, nom, lieu, ...)
- ...

Voici un exemple simple dont nous nous servirons. Ce sont les vainqueurs des élections présidentielles de la cinquième république.

nom	**prénom**	**études**	**politique**	**année**	**résultat**
de Gaulle	Charles	Saint-Cyr	D	1958	78,5
de Gaulle	Charles	Saint-Cyr	D	1964	55,2
Pompidou	Georges	ENS Ulm	D	1969	58,2
Giscard d'Estaing	Valery	X	C	1974	50,8
Miterrand	François	Droit	G	1981	51,8
Miterrand	François	Droit	G	1988	54,0
Chirac	Jacques	ENA	D	1995	52,6
Chirac	Jacques	ENA	D	2002	82,2
Sarkozy	Nicolas	Droit	D	2007	53,0
Hollande	François	ENA	G	2012	51,6
Macron	Emmanuel	ENA	C	2017	66,1

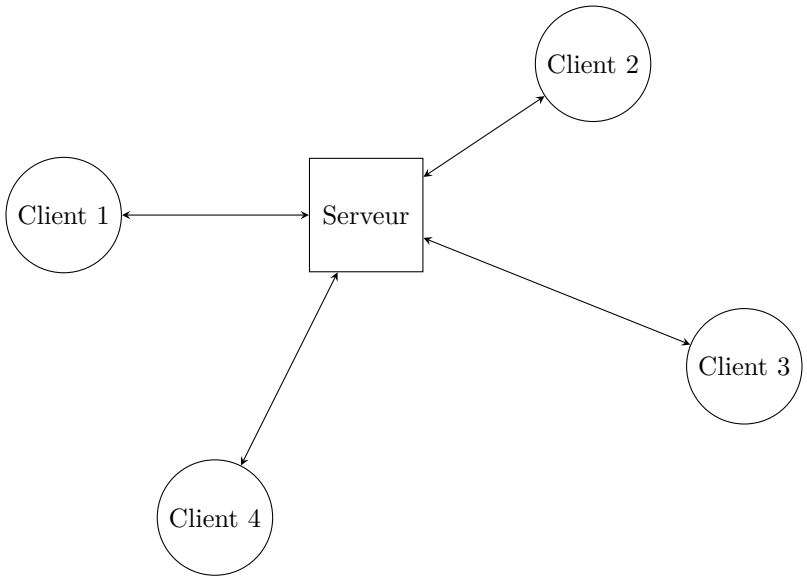
1.2 Travailler sur des données

Travail en direct

La première possibilité pour traiter ces données, et qui reste certainement la plus courante, est d'utiliser un tableur. C'est une solution très simple qui permet de faire des calculs sur des données quand on n'a pas besoin d'en extraire une partie. On peut trier selon une caractéristique, faire des calculs globaux (statistiques), faire des combinaisons de valeurs mais il est difficile de faire des moyennes partielles, par exemple dans le tableau ci-dessus, calculer la moyenne des résultats par couleur politique.
De plus il est difficile de permettre l'utilisation par plusieurs personnes d'une même base.

L'architecture client-serveur

On a choisi d'abstraire la base de données en la plaçant sur un serveur, chaque utilisateur, ou client, envoie ses demandes, on parle de requêtes, au serveur qui lui répond.



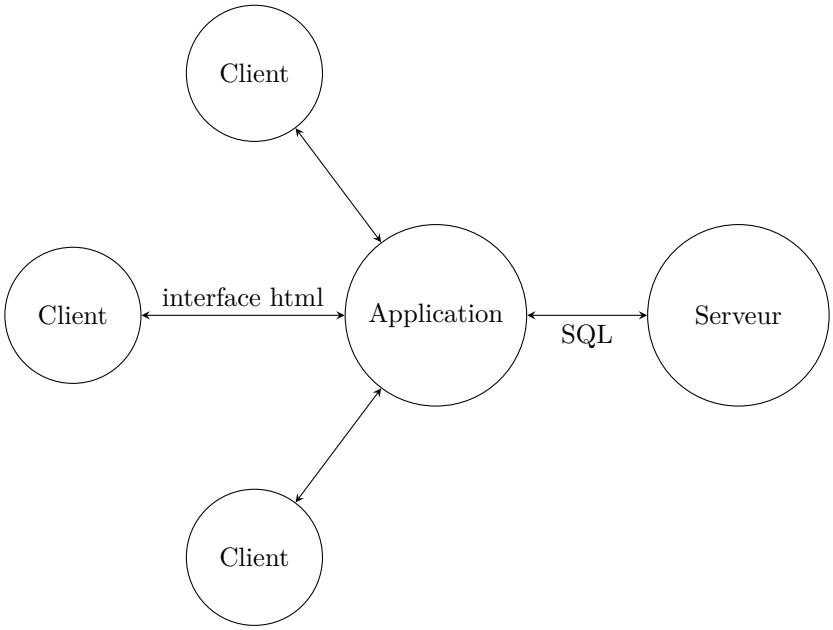
Il ne faut pas se fier à la simplicité apparente de cette structure, les problèmes à gérer sont nombreux et délicats :

- organisation des données,
- rapidité d'accès,
- gestion des autorisations d'accès,
- priorités des accès,
- sauvegarde des modifications,
- gestion des pannes ...

Il faut donc un langage pour établir les requêtes auprès d'un serveur. Ici, un phénomène inhabituel en informatique s'est produit, Cependant, contrairement aux langages de programmation, un standard a été choisi, c'est le langage SQL que nous allons partiellement exposer dans la suite. Ce langage permet de définir, modifier, interroger les différents systèmes du point de vue des utilisateurs. Bien entendu la programmation en interne de ces gros logiciels dépend de l'éditeur (Oracle, SAP, IBM, Microsoft, ...)

Pour les utilisateurs

On peut remarquer que SQL n'est pas visible pour les usagers. En effet l'utilisation des bases de données se fait maintenant à travers une architecture TROIS-TIERS. Entre l'usager et la base de données se place l'application qui travaille au dessus du gestionnaire de bases de données.



Plan

Dans la suite nous allons d'abord présenter le modèle des tableaux et le vocabulaire associé, puis nous définirons les opérations souhaitées. La partie suivante donnera la traduction opératoire de ces concepts dans le langage SQL.
On introduit enfin les outils statistiques.
Le chapitre suivant exposera un autre aspect de ce modèle, la possibilité de croiser plusieurs tables de données.

2 Relations

2.1 Vocabulaire

La composante de base, le tableau, est appelé **relation**.
On a vu qu'il comportait deux parties :

- les différents titres des colonnes qui sont appelés **attributs** ; on appelle **schéma** (**sort** en anglais) l'ensemble des attributs,
- l'ensemble des lignes est l'**extension** de la retaion, une ligne sera nommée **n-uplet** (**tuple** en anglais).

On nommera $A_1, A_2, \ldots, A_p$ les attributs.
L'ensemble des valeurs possibles d'un attribut A est son domaine $\mathrm{Dom}(A)$.
Chaque n-uplet est donc un élément de $\mathrm{Dom}(A_1) \times \mathrm{Dom}(A_2) \times \cdots \times \mathrm{Dom}(A_p)$.

2.2 Contraintes

La définition énoncée ci-dessus implique des contraintes. Certaines de ces conséquences devront être assurées par l'utilisateur d'autres seront contrôlées par le logiciel.

Les attributs forment un ensemble

- Il n'y a pas d'attribut en double : il ne peut être question qu'il y ait deux attributs de même nom, ils risqueraient d'être affectés de valeurs différentes dans des n-uplets.
- L'ordre des attributs n'est pas fixé :
  on ne parle pas du premier attribut mais de l'attribut `nom_attribut` (par exemple).
- En conséquences les différentes données ne sont donc pas des n-uplets au sens informatique du terme. Ce sont plutôt des fonctions depuis l'ensemble des attributs vers l'union des domaines avec la condition que $t[A_i]$ appartient au domaine $\mathrm{Dom}(A_i)$ (remarquer que l'on écrit avec des crochets plutôt que des parenthèses).
- On peut alors restreindre une ligne à un sous-ensemble d'attributs : si S est une partie de $\{A_1, A_2, \ldots, A_p\}$ on notera $t[S]$ l'ensemble des valeurs prises par le n-uplet t pour les attributs appartenant à S, par exemple $t[A_4, A_5, A_2]$.

Les lignes forment un ensemble

- Il n'y a pas de ligne en double : si $t[A_1, \ldots, A_p] = t'[A_1, \ldots, A_p]$ alors $t = t'$; cela correspond à l'aspect fonction des n-uplets.
- L'ordre des lignes n'est pas fixé.

Bien entendu une présentation des données dans un tableau aura un ordre des attributs et un ordre des n-uplets mais ces ordres de présentations sont le résultats des choix de l'utilisateur ou de l'ordre physique des données.

2.3 Clés

La contrainte de non-répétition des n-uplets est très importante ; la répétition de données induirait des problème de rapidité d'accès et fausserait les calculs. Pour cela les bases de données réelles contiennent un concept de **clé** qui doit être pensé dès la conception des bases de données. On verra qu'il a aussi une grande importance lors de l'utilisation de relations multiples.

Définition : Super-clé
Une super-clé d'une relation r est un ensemble S d'attributs tel que

$$\forall t, t' \in R \quad \big(t[S] = t'[S]\big) \Rightarrow (t = t')$$

Les contraintes imposent que l'ensemble de tous les attributs est toujours une super-clé.

Dans l'exemple ci-dessus {`année`}, {`nom`, `résultat`}, {`prénom`, `année`} sont des super-clés.

Définition : Clé candidate
Une clé candidate d'une relation r est une super-clé minimale pour l'inclusion.

K est donc une clé candidate si
1. K est une super-clé
2. Pour tout K' inclus dans K, K' n'est pas une super-clé.

Une super-clé qui ne contient qu'un seul attribut est toujours une clé candidate.
- {`nom`, `résultat`} n'est pas une clé candidate.
- {`année`} est une clé candidate.
- Lorsque vous vous identifiez sur un site sur lequel vous êtes inscrit, l'identifiant + le mot de passe est une clé candidate.

Définition : Clé primaire
Parmi les clés candidates on en choisit une : c'est la clé primaire.

Pour indiquer la clé primaire on souligne les attributs correspondants dans la table.
Indiquer au système une clé primaire pour chaque table permet une indexation des données à l'aide de cette clé, ce qui renforce l'efficacité des procédures d'interrogation de la table.
Nous connaissons déjà ce genre d'attributs :
numéro de sécurité sociale, plaque minéralogique, numéro de série ...
Pour obtenir une clé primaire efficace il est recommandé introduire un attribut, souvent nommé `id`, dans la définition de la relation.
Ce sera un entier qu'on incrémentera à chaque ajout d'un n-uplet.

id	**nom**	**prénom**	**études**	**politique**	**année**	**résultat**
1	de Gaulle	Charles	Saint-Cyr	D	1958	78,5
2	de Gaulle	Charles	Saint-Cyr	D	1964	55,2
3	Pompidou	Georges	ENS Ulm	D	1969	58,2
4	Giscard d'Estaing	Valery	X	C	1974	50,8
5	Miterrand	François	Droit	G	1981	51,8
6	Miterrand	François	Droit	G	1988	54,0
7	Chirac	Jacques	ENA	D	1995	52,6
8	Chirac	Jacques	ENA	D	2002	82,2
9	Sarkozy	Nicolas	Droit	D	2007	53,0
10	Hollande	François	ENA	G	2012	51,6
11	Macron	Emmanuel	ENA	C	2017	66,1

3 Algèbre relationnelle simple

Pour modéliser les requêtes on va considérer qu'une relation n'est qu'une relation parmi toutes les relations possibles et on a va modéliser la construction de relations à partir d'autres relations.
Depuis 1970 deux modèles abstraits de structure des données et d'extraction ont été étudiés : l'algèbre relationnelle et le calcul relationnel.
L'**algèbre relationnelle** formalise les opérations qui doivent être faites, le **calcul relationnel** décrit ce que l'on veut obtenir.
Ces deux modèles sont équivalents et ont servi de base théorique aux logiciels de base de données que les progrès de l'informatique ont permis d'écrire de manière efficace.
Nous allons étudier l'un de ces modèles : l'algèbre relationnelle.

3.1 Définition

L'algèbre relationnelle est un ensemble d'opérateurs que l'on peut appliquer à des relations, et dont le résultat est une relation.
Comme le résultat est toujours une relation on pourra combiner ces opérateurs : on forme ainsi des **requêtes** élaborées.
L'objectif est de pouvoir exprimer toute requête raisonnable par une combinaison d'opérateurs élémentaires.
Exemples de requêtes sur le tableau donné en exemple.

- Quels sont les présidents de droite ?
- Quels sont les présidents qui sortent de l'ENA ?
- Quels présidents ont gouverné plus de 10 ans ?

3.2 Opérateurs ensemblistes

Un premier type d'opérateur combine 2 relations qui ont le même schéma ; on les utilisera surtout pour assembler les résultats d'autres requêtes.
r et r' sont deux relations ayant le même schéma (c'est-à-dire les mêmes attributs).

1. $r \cup r'$ est la relation de même schéma dont les n-uplets appartiennent à r **ou** à r'.
2. $r \cap r'$ est la relation de même schéma dont les n-uplets appartiennent à r **et** à r'.
3. $r \setminus r'$ est la relation de même schéma dont les n-uplets appartiennent à r **mais pas** à r'.

3.3 Opérateurs spécifiques

Les opérateurs étudiés ici sont ceux qui utilisent la structure des relations. Dans cette partie nous n'étudions que les opérateurs unaires, qui s'appliquent à une relation unique.
Par exemple la question "Quels sont les présidents qui sortent de l'ENA ?" se traduit par :
donner les noms (projection) des présidents dont les études se sont passées à l'ENA (sélection).

Le **RENOMMAGE** consiste à renommer un attribut.
Ce sera utile lors de produits de tables lorsque deux tables ont des attributs portant le même nom mais correspondent à des données différentes.
Si $A \in R$ où R est le schéma de r et $B \notin R$ on note $\rho_{A\to B}(r)$ la relation obtenue en changeant le nom de l'attribut A en B.

1. Si R est le schéma de r avec $A \in R$, le schéma de $\rho_{A\to B}(r)$ est $(R \setminus \{A\}) \cup \{B\}$.
2. L'extension de $\rho_{A\to B}(r)$ est $\big\{t \ ; \ \exists s \in r, t[B] = s[A], \forall C \in R \setminus \{A\}, \ t[C] = s[C]\big\}$.

La **SELECTION** (ou restriction) consiste à ne garder que les n-uplets qui vérifie une propriété.
On note $\sigma_F(r)$ la relation extraite de r par la propriété F.

1. $\sigma_F(r)$ a le même schéma que r
2. L'extension de $\sigma_F(r)$ est $\big\{t \in r \ ; \ t \text{ vérifie } F\big\}$

On notera, pas abus d'écriture, $\sigma_F(r) = \big\{t \in r \ ; \ t \text{ vérifie } F\big\}$ sans noter explicitement le schéma.

Les propriétés élémentaires sont de la forme $P \,\theta\, Q$
où θ un opérateur de comparaison ($=$, $<$, $\leqslant$, $>$, $\geqslant$) et P et Q des expression construites à partir des attributs et des constantes avec des fonctions usuelles.

Une propriété composée avec des connecteurs logiques correspond à des unions et intersections et peut être écrite directement.
Par exemple $\sigma_{F\ et\ F'}(r) = \sigma_F(r) \cup \sigma_{F'}(r)$.

Exemple : F est la condition "prénom = François", $\sigma_F(r)$ est la relation

id	**nom**	**prénom**	**études**	**politique**	**année**	**résultat**
5	Miterrand	François	Droit	G	1981	51,80
6	Miterrand	François	Droit	G	1988	54,0
10	Hollande	François	ENA	G	2012	51,6

La **PROJECTION** consiste à ne garder qu'une partie des attributs. On note $\pi_X(r)$ la relation extraite de r avec les restriction des n-uplets à l'ensemble X.

1. $\pi_X(r)$ admet X pour schéma.
2. L'extension de $\pi_X(r)$ est $\{t[X];\ t \in r\}$

On notera, pas abus d'écriture, $\pi_X(r) = \{t[X];\ t \in r\}$ sans noter explicitement le schéma et en assimilant la restriction à X avec l'image $t[X]$.
Exemple : si X est l'ensemble {prénom, nom, résultat}, $\pi_X(r)$ est la relation

nom	**prénom**	**résultat**
de Gaulle	Charles	78,5
de Gaulle	Charles	55,2
Pompidou	Georges	58,2
Giscard d'Estaing	Valery	50,8
Miterrand	François	51,8
Miterrand	François	54,0
Chirac	Jacques	52,6
Chirac	Jacques	82,2
Sarkozy	Nicolas	53,0
Hollande	François	51,6
Macron	Emmanuel	66,1

Comme la projection doit être une relation les lignes ne sont pas répétées :
si X={étude}, $\pi_X(r)$ donne

études
Saint-Cyr
ENS Ulm
Polytechnique
Fac de droit
ENA

4 SQL

4.1 Introduction

Nous venons de décrire les premiers opérateurs de l'algèbre relationnelle.
On peut les comparer aux instructions de bases en programmation.
Pour les langages de requêtes il s'est produit un événement inhabituel : une norme universelle a été établie qui permet d'écrire des requêtes de la même façon quel que soit le logiciel : c'est **SQL**, pour Structured Query Langage.
Le langage SQL reprend la structure de l'algèbre relationnelle en y ajoutant des moyens de calculs et autres améliorations. Il le fait dans une formulation plus proche d'un langage humain (l'anglais) en structurant les requêtes dans un modèle simple.
Bien entendu chaque éditeur optimise les opérateurs pour donner des réponses le plus rapidement possible, ajoute des améliorations supplémentaires mais presque tous contiennent le langage tel qu'il est normalisé.

4.2 Instructions de base

Les représentations des relations se font avec le modèle du tableau.
En SQL on parlera de

- **tables** à la places de relations
- **colonnes** à la places d'attributs
- **lignes** à la places de n-uplets.

La forme de base d'une requête en SQL est une composition d'une sélection et d'une projection.

```
FROM table1
WHERE condition;
```

Les mots-clés de SQL sont usuellement écrits en majuscule mais ce n'est pas obligatoire

SELECT correspond à la projection, les attributs ou colonnes à afficher sont énumérés et séparés par une virgule. Il y a toujours une projection.
Si on ne veut pas effectuer de projection, on doit indiquer que l'on veut toutes les colonnes. On peut le faire par le symbole * qui signifie "tout".

FROM sélectionne le nom de la table à utiliser.
Cela deviendra plus signifiant lorsqu'on utilise plusieurs tables.

WHERE correspond à la sélection (attention : `SELECT` ne désigne pas la sélection).
La condition peut être composée à l'aide des connecteurs logiques `AND`, `OR` ou `NOT`.
Les comparateurs sont `=`, `>`, `<`, `>=`, `<=`, `!=`.
Les chaînes de caractères sont entourées de guillemets simples ou doubles.
La clause `WHERE` est facultative. On peut ne pas faire de sélection.

AS On peut renommer une colonne avec AS suivi du nouveau nom.

UNION, INTERSECT, EXCEPT Si on veut combiner plusieurs requêtes on peut les assembler avec `UNION` (pour $\cup$), `INTERSECT` (pour $\cap$) ou `EXCEPT` (pour $\setminus$).

Exemple : quels sont les noms et prénoms des présidents ayant été élus avant 1970 ou étant passés par l'ENA ? La question en algèbre relationnelle s'écrit

$$\pi_{\text{nom,prénom}}\big(\sigma_{\text{année} < 1970}(r)\big) \cup \pi_{\text{nom,prénom}}\big(\sigma_{\text{études} = \text{ENA}}(r)\big)$$

En SQL cela se traduit directement par

```
  SELECT nom, prénom
  FROM élections
  WHERE année < 1970
UNION
  SELECT nom, prénom
  FROM élections
  WHERE études = 'ENA';
```

ou, plus simplement,

```
SELECT nom, prénom
FROM élections
WHERE année < 1970 OR études = 'ENA';
```

Dans tous les cas on obtient

nom	prénom
de Gaulle	Charles
de Gaulle	Charles
Pompidou	Georges
Chirac	Jacques
Chirac	Jacques
Hollande	François
Macron	Emmanuel

SQL ne respecte pas la contrainte d'unicité des lignes.
Si on veut éviter les redondances il faut ajouter `DISTINCT` à `SELECT` :

```
SELECT DISTINCT nom, prénom
FROM élections
WHERE année < 1970
OR études='ENA'
```

donne

nom	prénom
de Gaulle	Charles
Pompidou	Georges
Chirac	Jacques
Hollande	François

4.3 Fonctions de présentation

SQL permet d'améliorer la présentation des résultats en triant selon un attribut avec ORDER BY.
Le résultat est trié selon l'ordre croissant.
Si l'on veut un ordre décroissant on le spécifie par `ORDER BY colonne DESC`
Cette instruction est placée à la fin de la requête.

On peut aussi limiter le nombre de résultats en faisant suivre la requête par LIMIT n.

Exemple

Si on veut le nom, l'année et le score des 2 meilleurs résultats des élections

```
SELECT nom, année, résultat
FROM élections
ORDER BY résultat DESC
LIMIT 2
```

donne

nom	année	résultat
Chirac	2002	82,2
de Gaulle	1958	78,5

5 Fonctions sur les attributs

5.1 Fonctions mathématiques

Il sera souvent utile d'effectuer des calculs à l'aide des valeurs renvoyées par une requête.
On peut utiliser les 4 fonctions arithmétiques de base : `+, -, *, /`
il est recommandé de renommer le résultat.
Par exemple, si une table contient les attributs `population` et `surface` on peut calculer la densité
`SELECT nom, population/surface AS densité, ...`
Remarque : si les colonnes utilisées ont des valeurs entières, les opérations sont à valeurs entières. En particulier la division est alors la division euclidienne : `7/2` renverra `3`. Pour obtenir la division usuelle il suffit de convertir au moins un des termes en flottant :
`(population + 0.0)/surface AS densité`.

Les gestionnaires de bases de données peuvent contenir d'autres fonctions, **sin**, **exp**, ...(mais ce n'est pas le cas de SQLite).

5.2 Fonctions statistiques

Les fonction ci-dessus permettent de calculer un résultat pour chaque ligne filtrée par la sélection : on crée en fait une nouvelle colonne.
On peut aussi utiliser le résultat d'une requête à des fins statistiques en calculant un résultat à partir de l'ensemble des données filtrées. Le résultat est alors table à une seule valeur calculée (ou plusieurs si on demande plusieurs calculs statistiques). Il n'est pas pertinent de demander l'affichage des attributs : quelle valeur serait donnée ?
Les fonctions disponibles sont

1. le décompte, c'est-à-dire le nombre de lignes : COUNT
2. le maximum des éléments dans une colonne : MAX
3. le minimum des éléments dans une colonne : MIN
4. la somme des éléments d'une colonne : SUM
5. la moyenne des éléments d'une colonne (sum/count) : AVG

Comme la table renvoyée par une fonction statistique n'a qu'une ligne et une colonne, sa valeur peut être utilisée dans les expressions de sélection.

```
SELECT AVG(résultat)
FROM élections;
```

donne 59,09.

```
SELECT nom, année,résultat
FROM élections
WHERE résultat  > (SELECT AVG(résultat)
                   FROM élections);
```

renvoie les résultats

nom	année	résultat
de Gaulle	1958	78,5
Chirac	2002	82,2
Macron	2017	66,1

5.3 Agrégations

On peut affiner les calculs statistiques en les calculant pour les différentes parties d'un découpage de la table.

Partition

Si on se donne un attribut B (ou un ensemble d'attributs) on peut partitionner la table selon les valeurs prises par B.

La table suivante correspond à la partition selon **études**.
On remarque que l'ordre peut être changé.

id	**nom**	**prénom**	**études**	**politique**	**année**	**résultat**
1	de Gaulle	Charles	Saint-Cyr	D	1958	78,5
2	de Gaulle	Charles	Saint-Cyr	D	1964	55,2
3	Pompidou	Georges	ENS Ulm	D	1969	58,2
4	Giscard d'E.	Valery	X	C	1974	50,8
5	Miterrand	François	Droit	G	1981	51,8
6	Miterrand	François	Droit	G	1988	54,0
9	Sarkozy	Nicolas	Droit	D	2007	53,0
11	Macron	Emmanuel	ENA	C	2017	66,1
10	Hollande	François	ENA	G	2012	51,6
8	Chirac	Jacques	ENA	D	2002	82,2
7	Chirac	Jacques	ENA	D	1995	52,6

Statistiques

On peut alors appliquer une fonction statistique à chacune des parties.
On note ${}_B\gamma_{f(A)}(r)$ cette opération si f est la fonction statistique appliquée à l'attribut A..
La table obtenue admet 2 attributs :

1. un attribut B qui prend les valeurs de B dans la table
2. un attribut calculé : pour chaque valeur b prise par B on calcule $f(A)$ pour la table r où l'on a sélectionné $B = b$.

${}_{\text{études}}\gamma_{\text{max(année)}}(r)$ donne la dernière année d'élection selon les écoles

études	**année**
ENA	2017
Saint-Cyr	1964
Droit	2007
ENS Ulm	1969
X	1974

${}_{\text{politique}}\gamma_{\text{moyenne(résultat)}}(r)$ donne la moyenne des résultats selon l'orientation politique

politique	**résultat**
C	58,45
D	63,28
G	52,47

- On peut faire plusieurs calculs statistiques par partie, ce qui donnera plus d'une colonne de calculs.
- On peut partitionner selon plusieurs attributs : chaque partie est alors définie par une même valeur pour chacun des attributs choisis.
- Un calcul statistique global, vu auparavant, correspond à une partition selon un ensemble vide d'attributs : il n'y a alors qu'une seule partie.

5.4 Agrégations en SQL

Dans SQL on regroupe les données par `GROUP BY colonne1, colonne2, ..., colonnep`.
Cette instruction est placée après l'éventuelle clause `WHERE`.
L'instruction `SELECT` ne doit contenir que les colonnes qui servent au regroupement et les fonctions statistiques. On rappelle qu'il est utile de nommer ces dernières.

La requête ci dessus s'écrit

```
SELECT politique, AVG(résultat) AS résultatMoyen
FROM élections
GROUP BY politique;
```

Autre exemple : calcul des moyennes des résultats des mandats par président.

```
SELECT nom, AVG(résultat) AS moyenne
FROM élections
GROUP BY nom;
```

nom	moyenne
de Gaulle	66,9
Pompidou	58,2
Giscard d'Estaing	50,8
Miterrand	52,9
Chirac	67,4
Sarkozy	53,0
Hollande	51,6
Macron	66,1

On peut toujours filtrer avant les regroupements (en amont des calculs) :

```
SELECT nom, AVG(résultat) AS moyenne
FROM élections
WHERE politique ='D'
GROUP BY nom;
```

nom	moyenne
de Gaulle	66,9
Pompidou	58,2
Chirac	67,4
Sarkozy	53,0

On peut aussi filtrer en utilisant les attributs calculés. Il faut donc imbriquer les requêtes.

```
SELECT nom,  moyenne
FROM (SELECT nom, AVG(résultat) AS moy
      FROM élections
      WHERE politique ='D'
      GROUP BY nom)
WHERE  moyenne < 60;
```

nom	moy
Pompidou	58,2
Sarkozy	53,0

SQL permet de simplifier cette construction en autorisant une sélection après calcul à l'aide de l'instruction `HAVING` qui se place après le `GROUP BY`.

```
SELECT nom, AVG(résultat) AS moy
FROM élections
WHERE politique ='D'
GROUP BY nom
HAVING moyenne < 60;
```

Chapitre XVIII

Bases de données composées

1 Tables multiples

1.1 Utilité

Dans le chapitre précédent on a décrit ce qui peut se faire avec un formulaire unique qui doit contenir tout ce qui est pertinent.
La plupart du temps une base de données sera composées de plusieurs tables. Pour illustrer l'utilité de cette structure et la construction d'une base, nous allons utiliser l'exemple d'une base de données concernant les notes de colles d'un lycée. Le cadre sera celui d'une décomposition selon le modèle **entités-associations**, parfois appelé entités-relations.

Si on veut pouvoir gérer les notes de colles on voit tout de suite un ensemble d'attributs nécessaires : colleur, matière, étudiant, note, date, heure.
Cependant, pour pouvoir identifier les étudiants, il faudra certainement plusieurs attributs : le nom, le prénom, la classe et d'autres caractéristiques. Ces informations sont alors inutilement répétées pour chaque colle passée par l'étudiant ; il en est de même pour les colleurs. Pour éviter cette redondance on organise les informations dans des tables séparées.

1.2 Entités

On commence donc par définir deux tables, c pour les colleurs et e pour les étudiants.
Ces tables décrivent des **entités**, c'est-à-dire des objets identifiables qui ont des caractéristiques communes, ici les colleurs ou les étudiants.
Un des gros intérêts de cette séparation est que l'on peut modifier les caractéristiques d'un étudiant (ou d'un colleur) en ne faisant qu'une seule modification de table, les caractéristiques ne sont pas répétées.
Le plus souvent chaque n-uplet sera repéré par un identifiant qui sert de clé primaire, dans nos tables c'est la valeur de l'attribut **id**.

Table XVIII.1 – Table des colleurs : c

id	**nom**	**prénom**	**établissement**	...
1	Martin	Edgar	Jean Bart	...
2	Durand	Valentin	César Baggio	...
3	Van Oyden	Lucille	Faidherbe	...
...	...	...	...	...

Table XVIII.2 – Table des étudiants : e

id	**nom**	**prénom**	**classe**	...
1	Aernout	Ludovic	PCSI1	...
2	Camblor	Patricia	MPSI3	...
3	Courbois	Éloïse	MP1	...
...	...	...	...	...

1.3 Associations

Une note de colle est le résultat d'une rencontre d'un colleur et d'un étudiant ; on va définir la tables des notes qui va contenir les références aux colleur et à l'étudiant ainsi que les caractéristiques de la colle. C'est une table d'**association**, elle établit une liaison entre entités.
Dans cette table, deux attributs, **clr** et **etu** doivent prendre pour valeur un indice existant respectivement dans la table c et dans la table e comme valeurs de l'attribut **id** qui est la clé primaire des tables. Une telle dépendance définit une clé **étrangère**. Une valeur d'une clé étrangère **doit** exister dans la table à laquelle elle réfère.

Table XVIII.3 – Notes des colles : la table n

id	**clr**	**etu**	**matière**	**date**	**heure**	**note**
1	2	1	Math	24/11/19	17h00	13
2	2	3	Math	24/11/19	13h00	17
3	3	1	Phys	24/11/19	17h00	14
...	...	...	...	...	...	

1.4 Représentation des schémas de la base

L'égalité, structurelle, entre une clé étrangère et l'attribut auquel elle réfère sera souvent indiquée lors de la représentation des schémas des tables par une liaison, doublement fléchée, entre les attributs qui doivent être égaux.

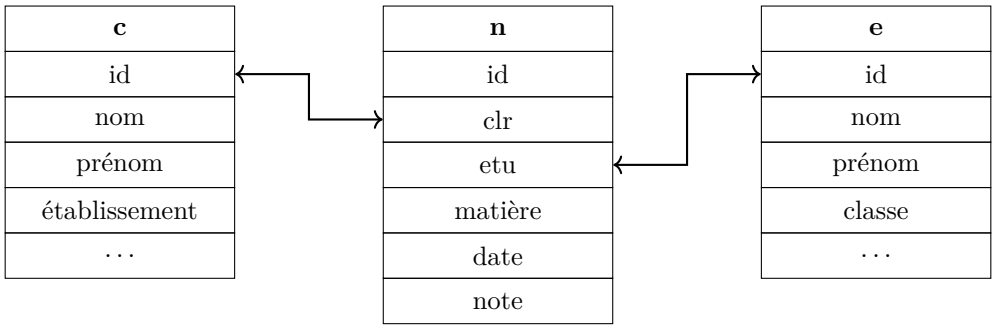


Figure XVIII.1 – Structure de la base de données

2 Utilisation de plusieurs tables

2.1 Produit

Le premier moyen d'assembler deux relations est d'en faire le produit cartésien.
On crée ainsi une nouvelle table en général beaucoup plus importante.

Définition : Produit de deux relations
Si r est une relation de schéma R et r' est une relation de schéma R' avec R et R' disjoints alors le produit de r et r' est la relation $r \times r'$ de schéma $R \cup R'$
et dont l'extension est l'ensemble des fonctions définies sur $R \cup R'$ dont les restrictions à R et à R' appartiennent respectivement à l'extension de r et à l'extension de r'.
L'abus usuel d'écriture donne $r \times r' = \{u \ ; \ u[R] \in r, \ u[R'] \in r'\}$

La condition $R \cap R' = \emptyset$ n'est pas très contraignante, il suffit de renommer les noms des attributs ; en général on notera **r.xxx** tous les attributs de r et **r'.xxx** tous les attributs de r' pour éviter les homonymies. En SQL, le produit est simplement engendré en listant les tables à multiplier dans la clause FROM séparées par des virgules.
Par exemple le produit de n et de e revient à associer les notes de tout le monde avec chaque étudiant

Table XVIII.4 – Produit des tables : $n \times e$

n.id	clr	etu	matière	date	heure	note	e.id	nom	prénom	classe	...
1	2	1	Math	24/11/19	17h00	13	1	Aernout	Ludovic	PCSI1	...
1	2	1	Math	24/11/19	17h00	13	2	Camblor	Patricia	MPSI3	...
1	2	1	Math	24/11/19	17h00	13	3	Courbois	Éloïse	MP1	...
...	...	...	...	...	...	...	...	...	...		
2	2	3	Math	24/11/19	13h00	17	1	Aernout	Ludovic	PCSI1	...
2	2	3	Math	24/11/19	13h00	17	2	Camblor	Patricia	MPSI3	...
...	...	...	...	...	...	...	...	...	...		

Pour des raisons d'encombrement on n'a écrit le préfixe que pour les colonnes de même nom. Dans la pratique il est recommandé de **toujours** écrire le préfixe.

2.2 Produit amélioré

Le produit ci-dessus ne représente rien sauf des paires note-étudiant. Si on veut vraiment les notes de colles pour chaque étudiant, on doit, dans le produit des tables n et e, imposer la condition d'égalité des identifiants des étudiants.

```
SELECT e.nom, e.classe, n.matiere, n.date, n.note
FROM e, n
WHERE  e.id = n.id_e
```

Si on veut obtenir de plus le nom du colleur on doit utiliser les trois tables.

```
SELECT e.nom, e.classe, c.nom, n.matiere, n.date, n.note
FROM c, e, n
WHERE  c.id = n.id_c AND e.id = n.id_e
```

Bien entendu on peut ajouter des sélections. Les notes, avec le colleur, des étudiants de PCSI3 en chimie sont obtenues par

```
SELECT e.nom, e.classe, c.nom, n.matiere, n.date, n.note
FROM c, e, n
WHERE  c.id = n.id_c AND e.id = n.id_e
                     AND e.classe = "PCSI3"
                     AND n.matière = "Chim"
```

2.3 Jointure

La réponse donnée ci-dessus n'est pas totalement satisfaisante : elle donne certes le bon résultat mais on écrit la requête en écrivant une sélection qui mélange une condition qui n'est pas facultative, celle provenant de la structure de la base, avec des critères de sélection qui dépendent de la question posée (clase, matière, ...).
La construction de la table produit ne contenant que les n-uplets respectant les conditions structurelles est donc fondamentale, c'est la **jointure** (en fait equi-jointure).

Définition : Jointure
Si r et r' sont des relations et si F est une condition sur $r \times r'$ alors la jointure de r et r' selon F est la relation $r \bowtie_F r' = \sigma_F(r \times r')$.

La jointure n'apporte donc rien de nouveau quant aux résultats produits mais elle permet de mettre à des niveaux différents les critères de sélection selon qu'ils sont donnés par la structure ou simplement par la question posée.
La traduction de la jointure en SQL est

```
table1 AS t1 JOIN table2 AS t2 ON t1.attribut1 = t2.attribut2
```

On remarquera qu'on peut renommer les tables jointes afin de simplifier le nom des attributs quand on leur ajoute le nom de la table. L'exemple ci-dessus devient

```
SELECT e.nom, e.classe, n.matiere, n.date, n.note
FROM  e JOIN n ON e.id = n.id_e
```

Dans le cas de plusieurs jointures on peut penser qu'on joint une nouvelle table à une jointure

```
table1 AS t1 JOIN table2 AS t2 ON t1.attribut1 = t2.attribut2
             JOIN table3 AS t3 ON tx.attributa = t3.attribut3
```

ou joindre 3 tables avec 2 conditions associées par AND.

```
table1 AS t1 JOIN table2 AS t2 JOIN table3 AS t3
             ON t1.attribut1 = t2.attribut2 AND tx.attributa =
                 t3.attribut3
```

3 Exercices

Le type d'exercices donnés ici est celui que l'on retrouve dans les épreuves de concours : on doit écrire des requêtes sans pouvoir contrôler les résultats. En général, comme ici, on donne les schémas des tables et les relations qui servent aux jointures et quelques lignes d'exemples.
Les tables sont donc celles de la base de colle. On demande d'écrire les requêtes SQL qui permettent de donner les réponses aux questions.

3.1 Requêtes simples

Exercice XVIII.1
Quels sont les noms et prénoms des colleurs qui viennent du lycée Faidherbe ?

Exercice XVIII.2
Quels sont les noms et prénoms des étudiants de la classe de MPSI4 ?

Exercice XVIII.3
Quelles sont les matières qui ont eu au moins une colle le "07/12/2019" ?

3.2 Agrégations

Exercice XVIII.4
Combien y-a-t-il d'élèves par classe ?

Exercice XVIII.5
Donner les note maximale, la note minimale et la moyenne des notes par matière.

Exercice XVIII.6
Déterminer le nombre d'occurrences de chaque note.

Exercice XVIII.7
Quels sont les lycées d'où proviennent plus de 30 colleurs ?

Exercice XVIII.8
Quelles sont les matières qui ont donné une moyenne des notes supérieure à 15 ?

3.3 Requêtes imbriquées

Exercice XVIII.9
Quelle est le nombre maximal d'élèves par classe .

Exercice XVIII.10
Quelle est la note de colle la plus attribuée ?

3.4 Jointures

Exercice XVIII.11
Donner les notes attribuées par le colleur dont le nom est "de Boisseson".

Exercice XVIII.12
Quels sont les étudiants (nom, prénom) qui ont eu au moins une fois 20 en colle ?

Exercice XVIII.13
Quelles sont les matières évaluées en colle en MPSI1 ?

Exercice XVIII.14
Quels sont les colleurs qui collent en PCSI2 ?

Exercice XVIII.15
Quels sont les étudiants ayant eu une colle avec le colleur Stéphane Hoguet en MPSI3 ?
On donnera aussi la date et la note.

3.5 Jointures avec agrégation

Exercice XVIII.16
Donner la moyenne des notes des colles de mathématiques (`Math`) des étudiants de la classe de MPSI2.

Exercice XVIII.17
Donner la note maximale donnée par les colleurs en physique (`Phys`).

Exercice XVIII.18
Quels étudiants n'ont eu que des notes supérieure à 15 en chimie (`Chim`) en PCSI1 ?

Exercice XVIII.19
Quels sont les étudiants de MPSI1 qui ont une moyenne de notes de colles supérieure à la moyenne de la classe en mathématique et en physique ?

4 Compléments

4.1 Division relationnelle

Il existe une fonction réciproque du produit, la division, dans l'algèbre relationnelle.
C'est l'ensemble des tuples sur $R \setminus R'$ que l'on peut prolonger sur R avec tous les tuples de r' en obtenant toujours un tuple de r.
On a $(r \times r') \div r' = r$ et $(r \div r') \times r' \subset r$.

Définition : Division de deux relations
Si r est une relation de schéma R et r' est une relation de schéma R' avec $R' \subset R$ alors la division de r par r' est la relation $r \div r'$ de schéma $R \setminus R'$ dont l'extension est l'ensemble des n-uplets l sur $R \setminus R'$ telles que, pour tout n-uplet l' appartenant à l'extension de r', l'union de l et l' appartient à l'extension de r..
L'abus usuel d'écriture donne $r \div r' = \{u \ ; \ \forall t' \in r', \ (u, t') \in r\}$.

Il n'existe pas en SQL d'équivalent direct à la division.

Un exemple

On crée la table **classe** des classes

```
SELECT DISTINCT classe
FROM e
```

On crée aussi la table **recap** des colleurs et des classes qui les concernent

```
SELECT DISTINCT c.nom, e.classe
FROM c JOIN n JOIN e
       ON c.id = n.id_c AND n.id_e = e.id
```

Le quotient **recap ÷ classe** est la table des colleurs qui collent dans **toutes** les classes.
Voici une requête permettant d'obtenir le résultat.

```
SELECT c.nom
FROM c JOIN n JOIN e
       ON c.id = n.id_c AND n.id_e = e.id
GROUP BY c.id
HAVING COUNT(DISTINCT e.classe) = (SELECT  COUNT()
                                   FROM (SELECT DISTINCT
                                      classe FROM e)
                                  )
```

Remarques
- L'agrégation porte sur `c.id` mais on n'affiche que `c.nom` qui est unique pour un identifiant donné.
- Le sélection avec `HAVING` utilise un calcul statistique qui n'est pas affiché.
- On a ajouté le modificateur `DISTINCT` qui permet de ne compter que les éléments distincts.
- On l'a utilisé aussi dans la projection `SELECT DISTINCT classe` car une table peut avoir des doublons en SQL, ce n'est pas ce qui est attendu en algèbre relationnelle.

4.2 Création de bases avec SQL

En complément nous allons présenter, sur l'exemple du chapitre, la création d'une base.
Il ne s'agit que d'une introduction, la création d'une base est un problème qui demande beaucoup d'attention et de savoir-faire.

Création du schéma

Pour créer une table il faut en énumérer les attributs en leur donnant un type.
Les types les plus courants sont

- INTEGER, un entier
- REAL, un réel
- CHAR(n), une chaîne de n caractères exactement
- VARCHAR(n), une chaîne de n caractères au plus

Il faut aussi spécifier les composantes de la clé primaire.
Cela se fait en écrivant les attributs de la clé primaire après les définitions des attributs.

```
CREATE TABLE xxx(
    ....
    PRIMARY KEY(attribut1, attribut2, ...));
```

Si la clé primaire ne contient qu'un attribut on peut le signaler lors de la définition.

```
CREATE TABLE colleurs(
    id INTEGER PRIMARY KEY,
    nom VARCHAR(20),
    prénom VARCHAR(20),
    établissement VARCHAR(20);
```

Clé étrangère

On peut ajouter une contrainte de clé étrangère par
`FOREIGN KEY (attribut) REFERENCES AutreTable (AutreTable.attribut)`
Lors d'usage d'une table avec clé étrangère, le logiciel de base de données se chargera de la vérification de l'existence de la clé référencée

```
CREATE TABLE n(
    id INTEGER PRIMARY KEY,
    id_c INTEGER,
    id_e INTEGER,
    matière VARCHAR(20),
    date VARCHAR(20),
    note INTEGER,
    FOREIGN KEY (id_c) REFERENCES c (id),
    FOREIGN KEY (id_e) REFERENCES e (id));
```

Extension

On introduit les différents n-uplets par l'instruction `INSERT INTO nomTable`
suivie de `VALUE (valeur1, ..., valeurp);`

On peut rentrer plusieurs n-uplets à la fois en les séparant par des virgules.
Si l'ordre d'entrée des valeurs n'est pas celui de la définition des tables on doit préciser les colonnes après INTO nomTable.

```
INSERT INTO e
VALUE (1, "Aernout", "Ludovic", "PCSI1"),
      (2, "Camblor", "Patricia", "MPSI3");
```

5 Solutions

Solution de l'exercice XVIII.1 -

```
SELECT nom, prénom
FROM c
WHERE établissement = "Faidherbe"
```

Solution de l'exercice XVIII.2 -

```
SELECT nom, prénom
FROM e
WHERE classe = "MPSI4"
```

Solution de l'exercice XVIII.3 -

```
SELECT DISTINCT matière
FROM n
WHERE date = "07/12/2019"
```

Solution de l'exercice XVIII.4 -

```
SELECT classe, COUNT() AS nombre
FROM e
GROUP BY classe
```

Solution de l'exercice XVIII.5 -

```
SELECT matière, MAX(note) AS note_max, MIN(note) AS note_min,
   AVG(note) AS note_moyenne
FROM n
GROUP BY matière
```

Solution de l'exercice XVIII.6 -

```
SELECT note, COUNT() as nb_occ
FROM n
GROUP BY note
```

Solution de l'exercice XVIII.7 -

```
SELECT établissement, COUNT() as nb_colleurs
FROM c
GROUP BY établissement
HAVING nb_colleurs >= 30
```

Solution de l'exercice XVIII.8 -

```
SELECT matière, AVG(note) as note_moy
FROM n
GROUP BY matière
HAVING note_moy > 15
```

Solution de l'exercice XVIII.9 -

```
SELECT MAX(nombre)
FROM (SELECT classe, COUNT() AS nombre
      FROM e
      GROUP BY classe)
```

Solution de l'exercice XVIII.10 -

```
SELECT note
FROM (SELECT note, COUNT() as nb_occ
      FROM n
      GROUP BY note)
WHERE note = (SELECT MAX(note)
              FROM (SELECT note, COUNT() as nb_occ
                    FROM n
                    GROUP BY note)
             )
```

Solution de l'exercice XVIII.11 -

```
SELECT n.note
FROM n JOIN c ON n.id_c = c.id
WHERE c.nom = "de Boisseson"
```

Solution de l'exercice XVIII.12 -

```
SELECT e.nom, e.prenom
FROM n JOIN e ON n.id_e = e.id
WHERE n.note = 20
```

Solution de l'exercice XVIII.13 -

```
SELECT DISTINCT n.matière
FROM n JOIN e ON n.id_e = e.id
WHERE e.classe  = "MPSI1"
```

Solution de l'exercice XVIII.14 -

```
SELECT c.nom, c.prenom
FROM n JOIN e ON n.id_e = e.id
       JOIN c ON n.id_c = c.id
WHERE c.classe = "PCSI2"
```

Solution de l'exercice XVIII.15 -

```
SELECT e.nom, e.prenom, n.date, n.note
FROM n JOIN e ON n.id_e = e.id
       JOIN c ON n.id_c = c.id
WHERE c.classe = "MPSI3" AND c.nom = "HOGUET"
                         AND c.prénom = "Stéphane"
```

Solution de l'exercice XVIII.16 -

```
SELECT e.nom, e.prenom, AVG(n.note)
FROM n JOIN e ON n.id_e = e.id
group by e.id
WHERE e.classe = "MPSI2" AND n.matière = "Math"
```

On notera qu'on n'a pas affiché l'identifiant des étudiants.

Solution de l'exercice XVIII.17 -

```
SELECT c.nom, c.prenom, MAX(n.note)
FROM n JOIN c ON n.id_c = c.id
WHERE n.matière = "Phys"
```

Solution de l'exercice XVIII.18 -

```
SELECT e.nom, e.prenom
FROM n JOIN e ON n.id_e = e.id
WHERE e.classe = "PCSI1" AND n.matière = "Chim"
HAVING MIN(n.note) >= 15
```

Solution de l'exercice XVIII.19 -

```
SELECT c.nom, c.prenom
FROM n JOIN e ON n.id_e = e.id
WHERE e.classe = "MPSI1" AND n.matière = "Math"
GROUP BY e.id
HAVING AVG(note) > (SELECT AVG(moy)
                    FROM (SELECT e.id, AVG(note) AS moy
                          FROM n JOIN e ON n.id_e = e.id
                          WHERE e.classe = "MPSI1" AND n.matiè
                             re = "Math"
                          GROUP BY e.id)
                   )

INTERSECT

SELECT c.nom, c.prenom
FROM n JOIN e ON n.id_e = e.id
WHERE e.classe = "MPSI1" AND n.matière = "Phys"
GROUP BY e.id
HAVING AVG(note) > (SELECT AVG(moy)
                    FROM (SELECT e.id, AVG(note) AS moy
                          FROM n JOIN e ON n.id_e = e.id
                          WHERE e.classe = "MPSI1" AND n.matiè
                             re = "Phys"
                          GROUP BY e.id)
                   )
```