COURS D'INFORMATIQUE
BCPST 1A

X. MOLIN

Année 2017-2018

Version du : 03/12/2017

TABLE DES MATIÈRES

I Un ordinateur
1 Les origines . . . 3
2 Composition d'un ordinateur . . . 4
3 Programmation et algorithmique . . . 8

II Variables et affectation en Python
1 Pour commencer . . . 12
2 Affectation et modification . . . 13
3 Fonctions . . . 15
4 Import de bibliothèques . . . 17

III Types de données

IV Structures conditionnelles
1 Branchement conditionnel . . . 28
2 Boucle while . . . 31

V Boucle for
1 Boucle for et commande range . . . 34
2 Objets itérables . . . 37
3 Listes en compréhension . . . 38
4 Complément : ruptures de séquences et de boucles . . . 38

VI Algorithmique avec les listes et chaînes de caractères
1 Recherche dans une liste . . . 40
2 Tri d'une liste . . . 43
3 Statistiques d'une liste de nombres . . . 46
4 Recherche dichotomique . . . 47

VII Modélisation d'expériences aléatoires
1 Tirages dans une urne modélisée par une liste . . . 51
2 Approximation des probabilités et de l'espérance . . . 54
3 Modéliser une loi de Bernoulli sans urne . . . 55
4 Simuler une loi discrète non uniforme . . . 56
5 Évaluer des probabilités conditionnelles . . . 56

VIII Le module numpy
1 De la liste au tableau . . . 58
2 Création des tableaux usuels . . . 62
3 Opérations matricielles . . . 63

4 Tests avec les tableaux . . . 65
5 Des statistiques et probabilités avec Numpy . . . 66
6 Quelques commandes pour aller plus loin . . . 67

IX Manipulation des fichiers par Python
1 Motivation . . . 72
2 Ouverture, fermeture et parcours d'un fichier texte . . . 72
3 Où suis-je ? . . . 75
4 Encodage . . . 76

X Liste d'algorithmes à connaître
1 Suites récurrentes . . . 77
2 Parcours de liste . . . 78
3 Tris de liste avec effets de bord . . . 79
4 Fonctions . . . 79
5 Tirages aléatoires . . . 80

I - UN ORDINATEUR

Vous avez tous déjà manipulé des ordinateurs, téléphones mobiles dits *intelligents*... mais savez-vous vraiment comment ils sont structurés et le principe de leur fonctionnement ?

Ce premier chapitre est à caractère essentiellement culturel, cela ne veut pas dire qu'il est dénué d'intérêt pratique. Au contraire.

1 Les origines

En 1642, Blaise Pascal a l'idée de créer une machine d'arithmétique que l'on appellera plus tard pascaline. Cette machine permettait d'additionner, soustraire, multiplier et diviser aisément des entiers en vue de simplifier le travail comptable de son père qui était super-intendant de Haute-Normandie. Elle permit le premier traitement mécanique d'une information[1].
On y trouve déjà les ingrédients essentiels de l'informatique :

- des données d'entrée : les nombres à manipuler,
- un processus (ici programmé mécaniquement avec les engrenages),
- une sortie : le résultat de l'opération.

En 1837, Charles Babbage invente la première machine à calculer programmable. il s'inspire des métiers à tisser Jacquard[2] dont on pouvait "programmer" le motif du tissage grâce à des cartes perforées.

Vers le milieu des années 1930 apparaissent les premiers ordinateurs. D'abord électro-mécaniques, il s'affranchissent peu à peu des composants mécaniques pour intégrer les lampes à vide puis l'électronique à partir de 1947.

> La première erreur informatique est apparue sur un ordinateur à lampe[3]. Un insecte attiré par la chaleur est venu se loger dans les lampes et a créé un court circuit. Il a donné son nom au *bug*.

L'apparition des transistors et des circuits intégrés permet de réduire considérablement la taille des appareils et leur consommation en énergie : on entre dans l'aire des *micro-ordinateurs*. En 1971, le premier micro-ordinateur Kenbach 1 a une mémoire de 256 octets ! Le premier "PC" (*personal computeur*) est commercialisé par IBM en 1981.

Définition 1.1 (Un ordinateur)

Un ordinateur est un dispositif électronique capable de traiter l'information.

[1] L'allemand Wilhelm Schickard avait déjà posé les plans d'une machine semblable quelques années plus tôt, mais ce projet se limita au stade du prototype contrairement à la pascaline qui fut construite en plusieurs exemplaires.

[2] Les premiers métiers Jacquard sont apparus en 1810.

[3] On peut trouver une explication très simple du fonctionnement d'un tube à vide (ou lampe) sur le site http://anjeanje.free.fr/OldHeaven/OldHeavenTubes.htm

Le terme français **ordinateur** a été introduit en 1955 par un responsable du service publicité d'IBM, François Girard. Il s'éloigne de la simple traduction de "computer" (un calculateur) et avance l'idée d'une gestion ordonnée de l'information.

2 Composition d'un ordinateur

Un ordinateur est un composant matériel (électronique) doté de logiciels. La partie matérielle est le **Hardware** et la partie logicielle le **Software**.
La structure de base d'un ordinateur (ou de tout autre appareil informatique : smartphone, appareil photo numérique, tablette…) est basée sur la figure I.1.

Figure I.1: Schéma de base en informatique I/O

Remarque : On pourrait définir une structure un peu plus générale, où l'entrée correspond à un *état* initial, et la sortie à un état final. Le traitement correspond ainsi à un changement d'état et il n'y a donc plus nécessairement de sortie explicite.

A Le Hardware

Pour le Hardware, cette structure peut être résumée par la figure I.2.

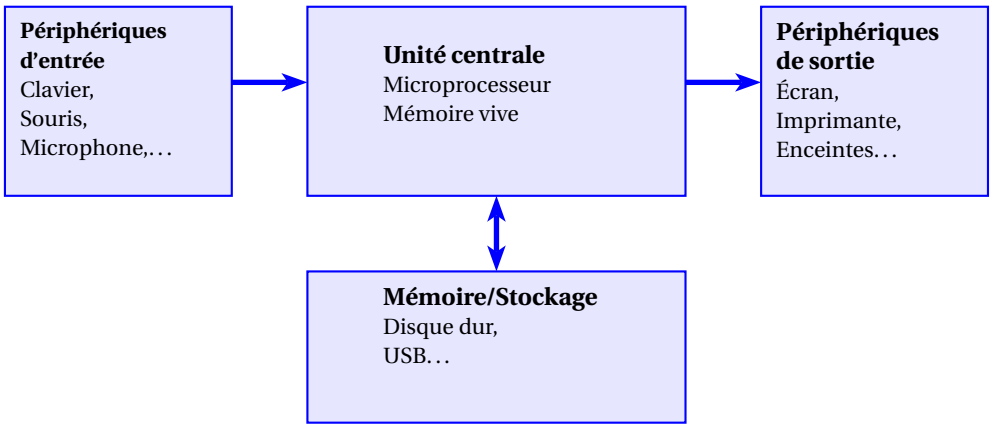


Figure I.2: Structure du Hardware

Les **périphériques** d'entrée et de sortie permettent simplement la communication avec l'unité centrale. Cela s'opère via des **unités d'échange** présentes dans l'unité centrale. L'utilisateur transmet ses instructions à l'ordinateur par l'intermédiaire des périphériques d'entrée, et l'ordinateur lui renvoie le résultat par les périphériques de sortie.
La mémoire assure le stockage des données. On distingue la **mémoire vive** (RAM) qui est une mémoire de transfert accessible très rapidement mais de petite capacité (car très coûteuse), de la **mémoire de masse** ou de stockage qui a vocation à conserver les informations sur le long terme.

Sur les premiers ordinateurs (et jusque vers 1970) les périphériques étaient très rudimentaires et se limitaient à des cartes perforées. En fonction du placement des trous, l'ordinateur comprenait l'information. Il renvoyait ensuite le résultat par carte perforée ou impression. Cette méthode était lourde, fastidieuse et sujette à de très nombreuses erreurs humaines.
La mémoire n'est pas apparue tout de suite non plus. L'intérêt premier de celle-ci est qu'elle permet de "sauvegarder" des programmes en interne pour des utilisations successives.

Le cœur de l'ordinateur est composé de la **carte mère**. Elle comporte certains composants intégrés et d'autres qui lui sont rapportés.

- Le **chipset** est le circuit électronique de la carte qui assure les communications entre les différents organes de

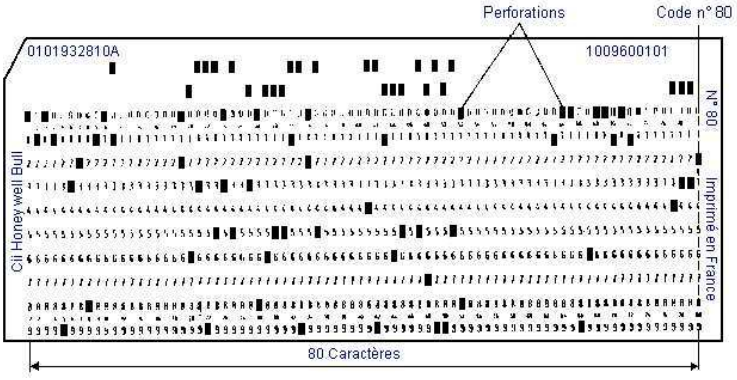


Figure I.3: Carte perforée

l'ordinateur[4]. Le chipset limite les possibilités d'évolution de l'ordinateur en fonction des informations qu'il peut faire transiter (et des connecteurs). Ainsi, il est inutile de brancher un composant trop récent sur une carte mère qui ne le supporte pas car elle ne saura pas l'utiliser.

- Le **CMOS** (*complémentary metal-oxyde semiconductor*) est une mémoire qui contient quelques données essentielles au fonctionnement de l'ordinateur, comme les secteurs disque de démarrage, les configurations matérielles... Cette mémoire est une cible privilégiée des virus car elle reste tout le temps allumée.
  La **pile du CMOS** est une pile plate qui sert à alimenter le CMOS même lorsque l'ordinateur est hors tension. C'est elle qui permet à votre ordinateur d'être à l'heure au démarrage.
  L'**horloge temps réel** (RTC) est chargée de synchroniser les échanges de données. Elle est alimentée par la pile du CMOS.

- Le **BIOS** (*Basic Input/Output System*) est un programme qui communique avec le CMOS (accessible via le BIOS setup) et permet de lancer le système d'exploitation d'un ordinateur. Le BIOS est situé sur la mémoire morte **ROM** (*read-only memory*, originellement, elle ne pouvait pas être modifiée).

- Le **processeur** (ou microprocesseur) exécute les instructions des programmes. Son cadencement définit le nombre d'opérations par seconde qu'il peut exécuter, il est donné par sa fréquence en MHz ou GHz. Un processeur à $1\,GHz$ permet de traiter un milliard d'informations par seconde.

  > L'**overclocking** est une accélération de la vitesse de l'horloge au delà de sa fréquence nominale afin d'augmenter les performances. Mais c'est une technique qui n'est pas sans risque (surchauffe du processeur, erreurs de calcul, instabilité ou destruction de l'OS...)

  La tendance est à la multiplication des processeurs, ou au multi-cœurs. Les architectures parallèles permettent ainsi d'effectuer plusieurs tâches en parallèle et non à la suite les unes des autres.
  Le processeur ne fait pas partie de la carte mère, mais il se branche dessus via le connecteur **SOCKET**. En fonction des gammes de processeurs, les **SOCKET** ne seront pas les mêmes. Le processeur est un élément coûteux de l'ordinateur.

- La **mémoire vive** ou **RAM** (*Random access memory*) est une mémoire d'accès très rapide pour conserver les informations d'usage immédiat. La RAM est entièrement vidée lorsque l'ordinateur est mis hors tension.

  > Si un ordinateur ne possède pas suffisamment de RAM, il utilisera la mémoire de masse (disque dur) pour conserver des résultats intermédiaires. Il fera donc beaucoup de lectures/écritures sur le disque dur qui est beaucoup plus long d'accès que la RAM (environ 100 000 fois plus long). Cela ralentira considérablement les processus.

- Un **dissipateur thermique** est associé à la carte mère pour dissiper la chaleur (produite par effet Joule) et éviter de faire fondre les composants.

  > Les data centers, calculateurs, ou simplement les salles équipées de nombreux ordinateurs sont climatisées pour contrer l'apport thermique dû à l'effet Joule. Les consommations électriques des grands calculateurs et des data centers sont en effet très importantes et représentent un gros enjeu actuel. La consommation des data centers est actuellement estimée à 1% ou 2% de la consommation électrique mondiale.

[4] En général, il contient deux puces, le pont *northbridge* qui gère les communications avec le processeur et les accès mémoire, et le pont *southbridge* qui communique avec les périphériques (entrées/sorties). Aujourd'hui, le chipset intègre souvent une puce graphique et une puce audio qui peuvent remplacer la carte graphique et la carte son (mais de qualité moindre).

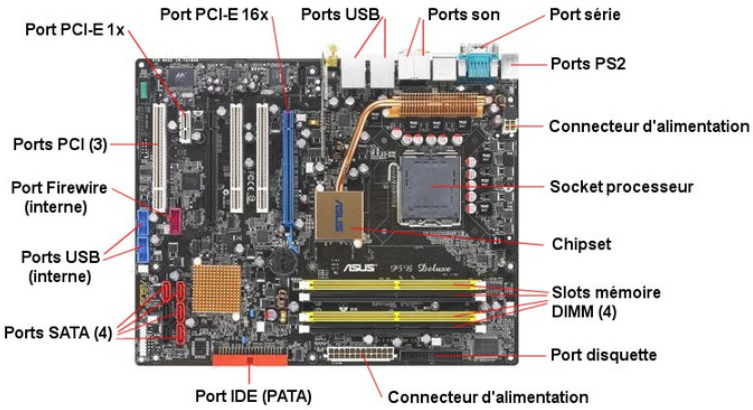


Figure I.4: Description d'une carte mère (*source : choixpc.com*)

La **mémoire de masse** correspond globalement au disque dur. La majorité des disques durs sont composés de disques recouverts d'une couche magnétique dont le champ magnétique local code des données (0 ou 1). La mémoire flash est plus récente, elle est utilisé sur les clés USB. Basée sur la technologie des semi-conducteurs, elle permet un accès plus rapide à toutes les données.

B Le Software

Le **système d'exploitation** (*OS : Operating System*) est un ensemble de programmes qui fait le lien entre les logiciels (ou les programmes que vous rédigerez) et les éléments matériels de votre ordinateur. Il permet l'échange entre le processeur, la mémoire et les systèmes applicatifs (voir la figure I.5).

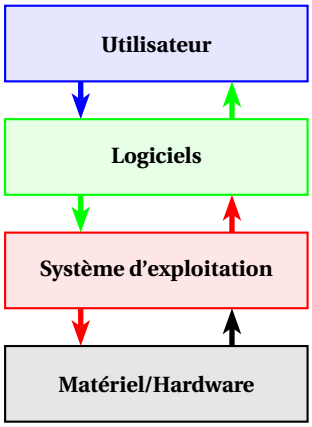


Figure I.5: Rôle du système d'exploitation

Il est démarré directement à partir d'un secteur d'amorçage après le BIOS.

Son rôle est de :

- gérer le processeur, en allouant ses ressources aux différentes applications (il peut gérer les priorités, faire fonctionner des tâches en parallèle...),
- gérer les accès mémoire (RAM et ROM) et le système de fichiers,
- gérer les droits d'accès, lecture, écriture, exécution,
- servir d'interface à la manipulation des périphériques (évite au programmeur de donner des instructions qui seraient propres à chaque périphérique). Pour cela il fait appel aux *pilotes* (pour traduire les instructions en fonction du matériel),

- Détecte et récupère les erreurs et si besoin les signale à l'utilisateur,
- ...

Il est composé

- du **noyau** (kernel) qui contient les fonctions fondamentales nécessaires au bon fonctionnement de l'ordinateur.
- le **shell** (coquille) pour communiquer directement avec le noyau en lignes de commande.
- le **système de fichiers** (*FS : File system*) qui organise les fichiers en arborescence.

Remarque : Un ordinateur peut fonctionner avec différents systèmes d'exploitation. Il faut simplement que l'OS soit programmé pour savoir contrôler l'architecture matérielle et ses composants. Il est par exemple très simple d'installer un système d'exploitation Linux sur une machine Windows.

L'**arborescence fichier** est gérée par le système d'exploitation. Chaque système d'exploitation gère ses arborescences de manière spécifique.
Le système de fichiers permet de situer physiquement sur la mémoire le contenu du fichier. Dans la mesure du possible, l'OS veillera à placer un même fichier sur des zones mémoire contigües, cela évitera d'avoir à aller le chercher en plusieurs endroits du disque et accélèrera les processus. Cette ambition reste néanmoins utopique en raison des modifications de fichiers, suppressions... qui ont vite fait de mettre le bazar dans le disque. Un disque où les fichiers sont en "petits morceaux" éparpillés est dit *fragmenté*. De plus en plus, les systèmes d'exploitation intègrent des processus pour défragmenter le disque au fur et à mesure de son utilisation[5].
Comme chaque système gère les fichiers à sa façon, un disque n'est pas structuré de la même manière selon l'OS (et son utilisation), c'est la raison pour laquelle il existe différents formats[6].

- NTFS pour Windows, et ReFS qui se développe,
- HFS+ pour Mac,
- ext4 et bientôt Btrfs pour Linux,
- FAT32 développé par Microsopft et souvent utilisé pour les échanges (clés USB). C'était le format utilisé de Windows 95 jusqu'à Windows Millénium.
- ...

Permissions d'accès aux fichiers et dossiers
Chaque fichier (ou dossier) est affecté de certaines permissions pour définir qui a le droit de le lire, de le modifier ou de l'exécuter. En général, vous n'avez pas besoin de vous préoccuper de la gestion des droits car le système d'exploitation les gère pour vous. En revanche, si vous êtes amené à mettre des ressources en ligne, la gestion des droits d'accès deviendra une préoccupation essentielle pour la sécurité de votre site et de ses données.

Pour un fichier (ou un dossier), les différents droits sont :

- la lecture (**r** *read*) pour lire le contenu du fichier,
- l'écriture (**w** *write*) pour écrire ou modifier le fichier,
- l'exécution (**x**) pour exécuter le fichier (par exemple un script ou un logiciel).

Attribuer des droits à un fichier, c'est associer à chaque catégorie d'utilisateur certains droits (lecture, écriture, exécution).

Les catégories d'utilisateur sont :

- le propriétaire du fichier désigné par la lettre **u** (*user*),
- le groupe du propriétaire désigné par la lettre **g** (*group*).
- les autres désignés par la lettre **o** (*other*),

[5] En raison de ces évolutions, la défragmentation manuelle n'est par exemple plus accessible directement depuis Windows 8, bien qu'elle soit encore possible par le shell.

[6] Ces formats ne sont pas liés au système physique qui est commun à tous les OS. Ainsi un même disque dur peut-être formaté dans différents formats à l'aide d'un logiciel.

On peut indiquer les permissions d'un fichier en mettant à la suite les droits de chaque catégorie. Par exemple si le fichier `file` est affecté des permissions `rwxr-x--x`, cela veut dire que le propriétaire à tous les droits (lecture, écriture et exécution), que le groupe peut lire et exécuter le fichier et que les autres ne peuvent que l'exécuter.
Les permissions peuvent aussi être traduites numériquement. À chaque droit correspond un nombre :

- 4 pour la lecture
- 2 pour l'écriture
- 1 pour exécution

Les différents droits sont additionnés pour donner le code. Avec l'exemple précédent, le propriétaire a tous les droits : $4+2+1=7$, le groupe a la lecture et l'exécution : $4+1=5$, et les autres n'ont que l'exécution : 1.
Les droits du fichier s'écrivent alors 451[7].
Remarque : En général, il existe aussi un super utilisateur, ou administrateur qui a tous les droits. Il est désigné par `sudo` ou `root`. Les systèmes d'exploitation récents évitent de donner les droits du super utilisateur sur une session normale pour limiter les risques d'erreur pouvant endommager le système.

3 Programmation et algorithmique

A Qu'est-ce qu'un langage de programmation

Les langages de programmation sont au fondement de toute l'informatique. Ils servent à communiquer avec la machine pour lui donner une suite d'instructions à réaliser (faire un calcul, afficher quelque chose...)

À la base, un ordinateur ne code que des 0 et des 1 sur son disque dur, sur un CD... C'est un codage qui est facile à réaliser mécaniquement. Par exemple sur une plaque, un zéro est représenté par un trou, et un 1 par une bosse.[8]

Chaque information que nous allons donner à un ordinateur devra être transformée en 0 et en 1 pour pouvoir être comprise par l'ordinateur. Vous imaginez bien que si on devait tout écrire nous-même en 0 et en 1, ce ne serait pas très pratique !

Un langage de programmation sera donc simplement une langue compréhensible par l'humain et qui sera ensuite traduite en langage machine.

> Il existe des langages de plus ou moins haut niveau. Un langage de haut niveau permet de faire abstraction de beaucoup de problématiques purement informatiques (gestion de la mémoire...) pour se concentrer sur les instructions.
> Python fait partie des langages de haut niveau.

Il existe deux[9] types de langages, les langages compilés et les langages interprétés (voir figure I.6).

> En réalité, de plus en plus de langages dits interprétés sont des **langages compilés en langage intermédiaire bytecode**. Cela signifie qu'au moment de l'interprétation, une compilation a lieu en bytecode. Le bytecode est une forme de langage intermédiaire entre le code source et le langage machine.
> C'est le cas pour Python, et le bytcode nécessite Python pour être interprété mais est plus rapide à exécuter que le code source. L'étape de compilation n'est pas visible pour l'utilisateur, elle produit le fichier d'extension `.pyc`.

Lorsque vous avez un logiciel comme Word, Firefox,..., il s'agit d'un programme compilé : si vous ouvrez le fichier du programme (fichier.exe sous Windows), il n'est pas compréhensible : vous pouvez simplement l'exécuter. Ceux qui programment le logiciel possèdent le texte du programme, mais il ne vous vendent que le programme déjà compilé.
Pour vous donner des idées, un système d'exploitation comme Windows, est un programme en plusieurs morceaux qui utilise plusieurs langages de programmation, mais majoritairement du C/C++. Windows XP contenait 40 millions de lignes de code. Pas étonnant qu'il y ait des erreurs et des failles !

Il existe plus de 7000 langages de programmation différents, mais peu sont effectivement utilisés. Le fonctionnement de chacun est spécifique et répond à des objectifs particuliers. Un programmeur connaît toujours plusieurs langages et utilise l'un ou l'autre en fonction de ce qu'il veut faire.

[7] Cela correspond à une écriture en base 8. On peut vérifier simplement que chaque type de droit donne un unique nombre.
[8] Le principe de creux et de bosses est utilisé pour les CD. Pour les disques durs, on utilise plutôt le magnétisme comme expliqué plus haut : de façon simplifié, l'ordinateur regarde si l'aimant est orienté dans un sens ou dans l'autre et en déduit la valeur du *bit* : 0 ou 1.
[9] En fait, cela dépend de l'implémentation du langage plus que du langage lui-même et un langage de type interprété pourrait aussi être implémenté pour fonctionner avec un compilateur. Néanmoins, il existe un usage majoritaire pour chaque langage ce qui explique la dénomination.

HTML qui "code" les pages web n'est pas un langage de programmation. Il ne permet de donner aucune instruction à l'ordinateur. C'est juste une convention pour dire que tel élément est un titre, tel autre est un paragraphe...
Chaque navigateur lira ce "code" et choisira comment l'afficher. Mais il n'est pas possible de faire faire des choses intelligentes à HTML. C'est simplement du texte mis en forme suivant certaines conventions.
Par contre, il existe PHP qui est un langage de programmation pour le web. Lorsque vous allez sur une page web, vous ne voyez pas de PHP parce qu'il a déjà été exécuté par le serveur et vous ne recevez qu'une page HTML toute simple (ou presque).

Nous allons apprendre les langages de programmation un peu comme on apprend des langues étrangères. Nous apprendrons leur syntaxe et leur grammaire pour pouvoir communiquer avec la machine.

Pour écrire un programme, il faut être précis et rigoureux. En effet, à la moindre faute de grammaire, l'interpréteur ne comprend plus rien et renvoie un résultat faux ou une erreur. Il est encore moins tolérant que votre prof.

B Algorithmique

Maintenant que vous savez ce qu'est un langage de programmation, il faut apprendre à l'utiliser de façon intelligente. En effet, il ne suffit pas d'apprendre un langage, il faut aussi savoir ce que l'on veut dire avec ce langage. Que veut-on que fasse l'ordinateur ? C'est l'objet de l'**algorithmique**.

Le langage concerne la forme[10]
L'algorithmique concerne le contenu

Lorsque vous avez un article de journal, vous pouvez avoir le même article en plusieurs langues différentes, mais le contenu sera le même.

L'algorithmique que vous apprendrez, c'est "le contenu du journal". Vous pourrez donc l'utiliser ensuite sur n'importe quel langage de programmation, et pas seulement avec Python.

Vous imaginez bien que lorsque vous réalisez de gros programmes, de plusieurs milliers, voire millions de lignes, il n'est pas possible de mettre un programmeur qui commence à la première ligne et qui avance jusqu'à la dernière ligne comme s'il écrivait un livre. C'est la raison pour laquelle les programmes sont toujours écrits par "**blocs**" que l'on imbrique les uns dans les autres, ou que l'on accole... comme dans un jeu de construction. Chaque programmeur est alors responsable du codage de certains blocs.

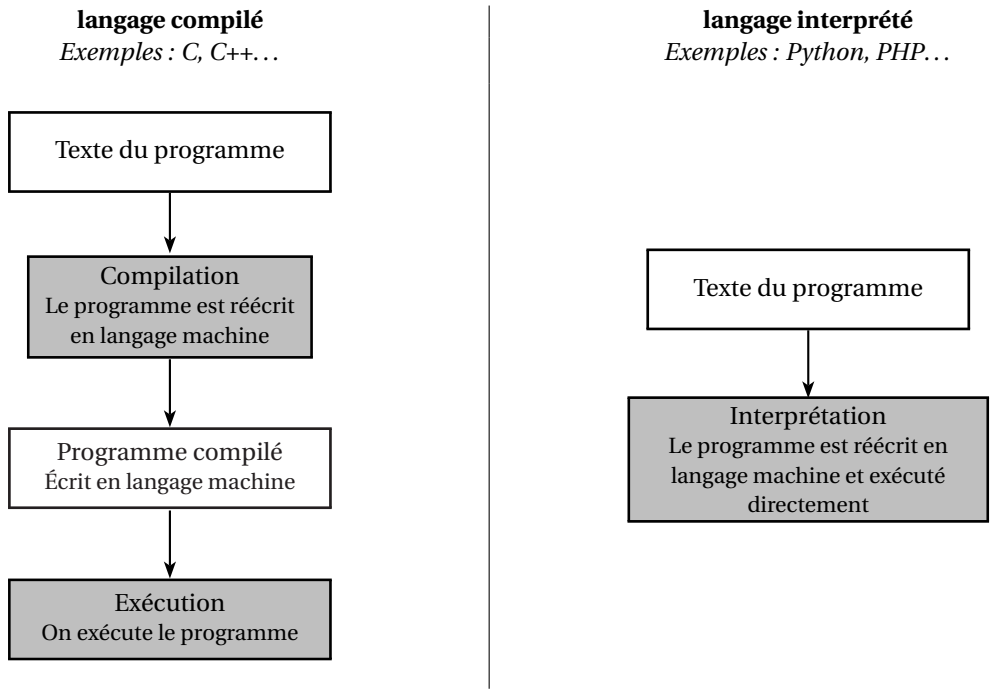
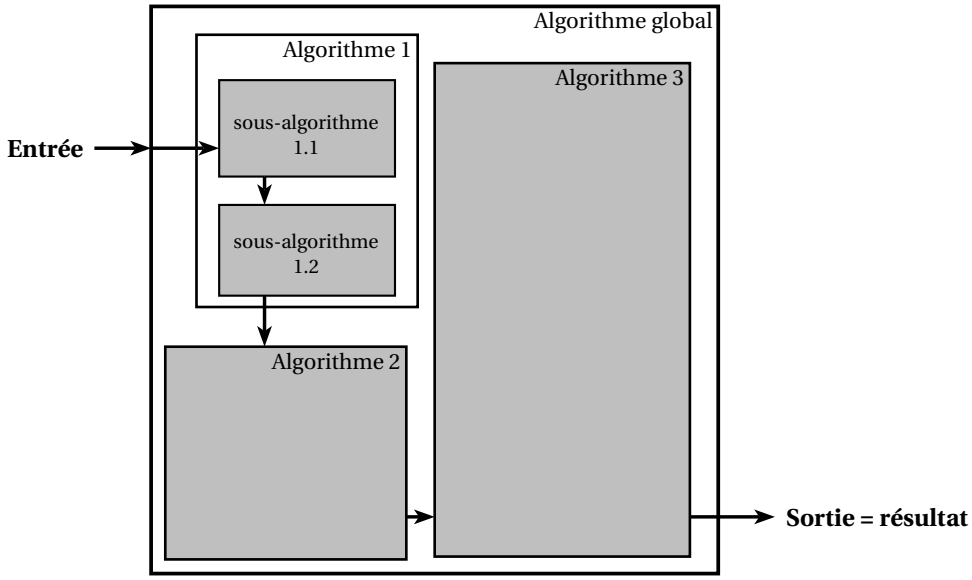


Figure I.6: Langage compilé - langage interprété

[10] En réalité, comme pour une langue, la forme va influencer le contenu. C'est la raison pour laquelle il existe une multitude de langages de programmation différents.

Les algorithmes sont donc des structures naturellement **récursives**. Cela veut dire que chaque algorithme[11] peut être décomposé en plus petits algorithmes (qui peuvent eux-mêmes être décomposés à nouveau, et ainsi de suite). Réciproquement, chaque algorithme que j'ai écrit peut ensuite servir de bloc pour un algorithme "plus gros" qui l'englobe.



Les zones grisées peuvent être décomposées à nouveau

Figure I.7: Schéma "digestif" d'un algorithme

Dans la réalité, ce schéma (figure I.7) peut être implémenté de deux façons selon le type de langage.

- le langage **fonctionnel** crée les sous-algorithmes qu'il réutilise un maximum de fois dans des "méta-algorithmes"... C'est ce que nous avons proposé ici (et sans doute le plus simple) en partant des briques élémentaires pour construire l'édifice.
- le langage orienté **objet** possède une logique plus algébrique. On crée des *objets* à l'intérieur de *classes*. Chaque objet possède des *attributs* et des *méthodes*. On part de l'objet le plus général possible que l'on spécifie ensuite par un système d'*héritage* en fonction de l'utilisation voulue. La logique est donc un peu inversée, puisque l'on part du plus général pour aller vers le particulier. Ce type de programmation est privilégié aujourd'hui, même si nous l'utiliserons peu cette année.

Python permet de programmer suivant les deux méthodes.

Par la suite, chacun de ces blocs fonctionnera un peu comme une boîte qui "transforme" des données d'entrée en données de sortie.
On définira alors un algorithme de cette façon :

Définition 3.1 :
Un algorithme est un processus logique qui comporte une entrée, un traitement, et une sortie.
Un algorithme comporte un nombre fini d'instructions et **se termine en un temps fini**.

Remarque : On pourrait remplacer le terme "logique" par "déterministe" dans la définition. Cela signifie d'une certaine manière que l'algorithme n'a pas de libre arbitre, qu'il n'improvise pas. Quand je lui donne les mêmes données à l'entrée, il me donnera toujours les mêmes données à la sortie : il n'y a pas d'aléatoire[12].

Comme spécifié plus haut, on pourrait parler d'état initial et d'état final au lieu d'entrée et de sortie.

Lorsque l'on veut écrire un programme, le plus important est de définir proprement quelles seront nos "briques", ce qu'elles recevront en entrée et le résultat qu'elles doivent donner en sortie. Lorsque ce travail est fait, il ne suffit plus que d'écrire les algorithmes pour chaque brique.

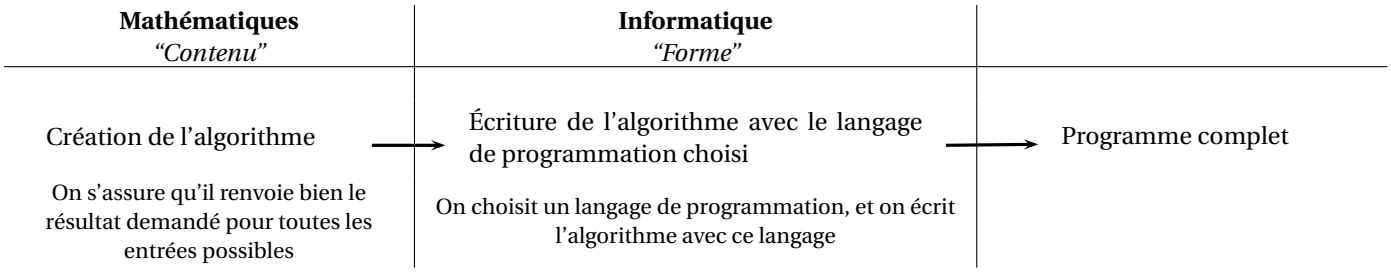
Avant d'écrire un algorithme, posez vous toujours ces questions :
Quelle entrée vais-je donner à mon algorithme ?
Quel résultat doit-il me fournir ?

[11] sauf un algorithme *élémentaire*

[12] Les ordinateurs ne savent pas générer d'aléatoire. Même lorsque vous demandez un nombre aléatoire à un ordinateur ou à une calculatrice, ce nombre n'est pas vraiment aléatoire, mais est calculé de façon suffisamment complexe pour qu'il en donne l'impression. Cela peut parfois poser des problèmes lors de certaines études pointues qui demandent des nombres *vraiment* aléatoires.

C Bonne exécution d'un programme

Pour définir un algorithme, nous avons parlé de processus logique. Écrire un algorithme relève donc essentiellement du domaine mathématique dont la logique est partie intégrante. La partie informatique de la programmation, n'est plus tellement l'algorithme lui-même, mais plutôt son écriture avec le bon langage de programmation.



Cela est le signe d'une exigence : s'assurer qu'un algorithme se termine forcément et qu'il donne le résultat voulu dans tous les cas.

Par exemple, si un algorithme prend en entrée un nombre quelconque, il faut bien vérifier que cela marche pour tous les nombres (positifs, négatifs, nuls, entiers, décimaux...). Ainsi, avant de diviser par ce nombre, je m'assurerai qu'il n'est pas nul.

Normalement, tout algorithme doit pouvoir être prouvé (comme un théorème mathématique). C'est un exercice formel long et un peu compliqué. Dans la réalité, les algorithmes sont rarement prouvés, car cela serait trop long à mettre réaliser. Nous ne le ferons pas non plus.

En pratique, on remplace donc les preuves par des **tests**. Ces tests ne pourront jamais nous prouver que l'algorithme est *juste*, mais ils nous aideront à éviter de nombreuses erreurs. Cela suppose simplement de penser à faire des tests qui recouvrent le maximum de cas possibles (nombres positifs et négatifs, nuls...) et de vérifier à chaque fois que le résultat rendu est le bon.
Il ne suffit pas d'obtenir un résultat, il faut s'assurer que ce soit celui voulu. On commencera donc par des tests dont on peut prévoir l'issue sans utiliser l'ordinateur.

De même, dès que vous programmerez une boucle, il faudra toujours vous demander si elle se termine dans tous les cas et ne risque pas de tourner indéfiniment.

D Algobox

Algobox est un langage de programmation à usage pédagogique uniquement. La syntaxe (la "grammaire") de ce langage est beaucoup trop lourde pour que l'on puisse rédiger de gros programmes avec.

L'avantage de ce langage est qu'il met bien en exergue l'idée d'un algorithme construit à partir de boîtes (box en anglais).

Dès que vous avez bien compris cette structuration des algorithmes comme des boîtes ou des briques que l'on assemble, il est temps de passer à un langage un peu plus sérieux. C'est ce que nous faisons avec Python.

E Python

Python est un langage qui a été conçu vers 1990 par Guido van Rossum. Son nom vient de la série télévisée "Monty Python" dont le créateur était un passionné.

C'est un langage assez simple, bien documenté et donc adapté pour débuter. Mais il est également très utilisé par des programmeurs expérimentés. Si vous êtes amenés à travailler avec un langage de programmation en entreprise (dans un avenir pas trop lointain...), il y a de fortes chances que ce soit celui-ci.

Son succès, fait qu'il est enrichi de nombreuses bibliothèques (des collections d'algorithmes pour réaliser des opérations qui ne sont pas *natives* à Python : travail sur les tableaux, tracer des graphes, réaliser des dessins dynamiques...).

Revenir au sommaire

II - VARIABLES ET AFFECTATION EN PYTHON

1 Pour commencer

Pyzo

Python existe sous de nombreuses présentations. Dans le cadre du cours, nous utiliserons la distribution Pyzo que l'on peut télécharger gratuitement sur le site http://www.pyzo.org

C'est cette distribution qui est actuellement utilisée aux concours.

Deux modes de saisie

Les éléments qui composent la fenêtre Pyzo peuvent être réarrangés en faisant glisser les bandeaux correspondants avec la souris. Le choix des éléments à afficher peut être déterminé à partir du menu.

Pour commencer, nous conseillons d'adopter la présentation suivante avec une séparation verticale :

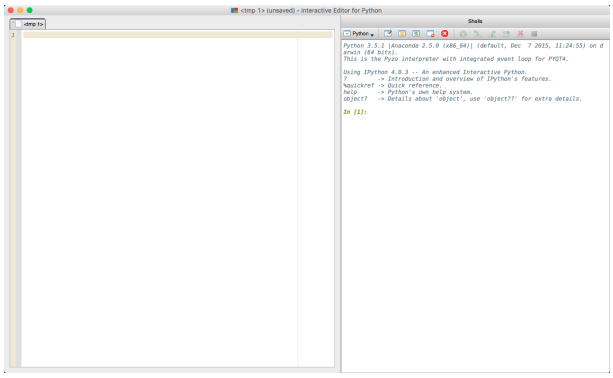


Figure II.8: Fenêtre Pyzo

- **Partie de gauche :** le fichier ".py" dans lequel on écrit les instructions. Ce ficher peut être enregistré pour une utilisation ultérieure.
  Pour exécuter les instructions qui y sont écrites on peut taper sur F5 (tout est exécuté) ou F9 (seulement la sélection est exécutée).
- **Partie de droite :** le shell dans lequel apparaissent les réponses de Python lors de l'exécution du fichier. C'est aussi le mode interactif. Python réagit "en direct" à ce que l'on écrit. Il interprète chaque ligne dès que l'on appuie sur la touche "Entrée".

Instructions Python

Les instructions Python s'écrivent les unes à la suite des autres en passant à la ligne entre deux.

Si on veut placer plusieurs instructions sur une même ligne, on peut les séparer d'un point-virgule ";".

Nota : Contrairement à beaucoup d'autres langages, il n'y a pas de point virgule à mettre en fin de ligne.

Commentaires

On peut placer des commentaires dans notre programme Python pour le rendre plus lisible et expliquer ce que l'on code.

Les commentaires sont précédés du signe "#". Dès qu'il voit ce signe, Python passe directement à la ligne suivante sans s'occuper de ce qui est écrit après.

Attention : Dans la suite du cours, il est conseillé de tester tous les exemples sur une machine, pour assimiler la syntaxe.

2 Affectation et modification

A Affection d'une variable

Affectation

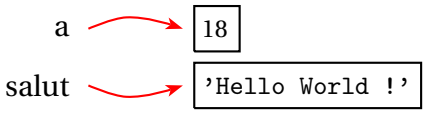
En Python, une variable est une façon de nommer un objet, ou plus précisément, de faire référence à un objet présent dans la mémoire.

Une variable ne *"contient"* pas l'objet, mais pointe vers l'objet. On parle de **référence objet**[13].

Lorsque l'on fait pointer la variable vers un objet, on dit que l'on **affecte** cette variable. En Python, l'affectation se fait avec le signe égal =.

```
# Ceci est un commentaire (précédé par dièse)
a = 18                    # la variable a pointe vers la valeur 18
salut = 'Hello World !' # la variable salut pointe vers le texte 'Hello World !'
```

Python II.1: Affectation



Affichage

La commande `print` affiche la valeur vers laquelle pointe une variable[14].

Pour écrire plusieurs éléments à la suite (séparés d'un espace), on les sépare par des virgules en argument de `print`.

```
salut = 'Hello World !'
print(salut)
print('Python vous dit :', salut)
```

Python II.2: Affichage

Attention : Python n'est pas un logiciel de calcul formel : il faut affecter une variable (la définir) avant de l'utiliser. C'est comme en maths.

Remarques pour les pros :

- Les variables sont des mots composés des caractères alphanumériques habituels et du signe souligné : "_". Un nom de variable ne peut pas commencer par un chiffre. N'hésitez pas à donner des noms de variable explicite avec suffisamment de lettres.
  Python est *sensible à la casse*. Cela veut dire qu'il distingue les majuscules des minuscules. Ainsi, les variables `var` et `Var` sont différentes.
- Lorsque l'on affecte une variable pour la première fois, elle est créée automatiquement par Python, il n'est pas nécessaire de la déclarer avant comme dans certains langages.
- Une fois que l'on a affecté une valeur à une variable, on peut la modifier en *écrasant* sa "valeur" par une autre. On pointe simplement la flèche vers une autre case mémoire.

[13] Ceci sera important lors des copies de listes que nous verrons plus tard.
[14] On peut aussi simplement écrire le nom de cette variable dans le shell et exécuter.

Modification d'une variable

```
a = 'glop'
b = 'pas glop'
a = b                  # on écrase la variable a pour lui donner la valeur 'pas glop'
print(a); print(b)
maVariable = 1
print(maVariable)
print(mavariable)      # Renvoie une erreur car la casse est différente
```

Python II.3: Modification d'une variable

Attention : Le signe égal n'est pas symétrique pour Python. Dites vous que = se traduit par "pointe vers" (voir la figure II.9).

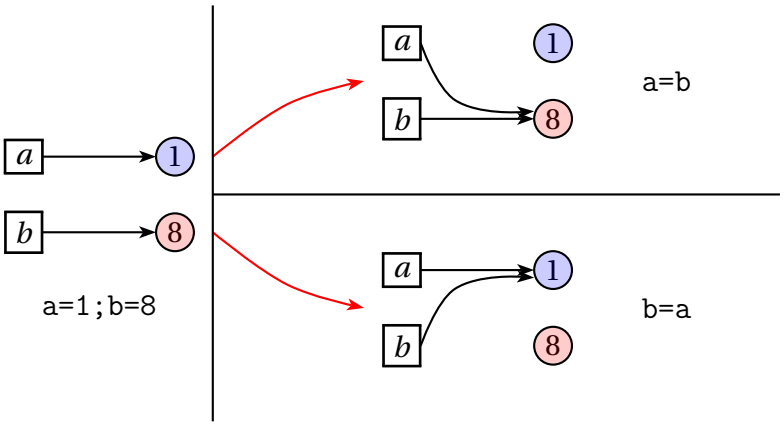


Figure II.9: Non symétrie de l'affectation

Explication de la figure II.9 :

Au début, on affecte la valeur 1 à la variable `a` et la valeur 8 à la variable `b`. Python crée donc ces deux variables et les fait pointer vers des cases mémoires dont l'une contient la valeur 1 et l'autre, la valeur 8.

Si on leur donne l'instruction `a=b` (illustration d'en haut), alors on modifie la variable `a` pour lui donner la valeur de `b`. Le pointeur de `b` ne change pas, par contre, celui de `a` va pointer vers la même case mémoire que celui de `b`.

À l'inverse, si on place l'instruction `b=a` (illustrée en bas), cette fois ci, c'est la variable `b` qui est modifiée et va faire référence à la même case mémoire que `a`.

Remarques pour les pros :

- Python optimise la mémoire et dans l'instruction `a=b`, on fait pointer `a` vers la même case mémoire que `b`. Cela évite d'avoir deux fois la même information en deux endroits du disque différents. Nous verrons que ceci n'est pas sans influence sur le comportement de Python lorsque nous manipulerons des objets plus complexes.
- Dans l'exemple, plus aucune variable ne pointe vers la case mémoire qui contient le 8. On peut donc libérer cet emplacement mémoire pour y placer d'autres informations plus tard. Python s'en occupera pour nous à travers le GC :*garbage collector* ou ramasse-miettes. Le rôle de cet outil intégré à Python est de détecter les cases mémoires vers lesquelles plus aucune variable ne pointe pour les remettre à disposition.
  Tous les langages ne possèdent pas de GC (qui n'a pas que des qualités), ainsi en est-il du langage C pour lequel c'est au programmeur de faire ce travail.

B Affections simultanées

On peut affecter plusieurs variables d'un coup, en les séparant par des virgules (*tuple*). On peut mettre ou non des parenthèses autour du uplet. De manière général, les parenthèses faciliteront la lecture.

```
a,b,c = 1,'deux',3
print(a); print(b); print(c)
(a,b,c) = ('un',2,'trois')
```

```
print(a); print(b); print(c)
print('je compte',a,b,c)
```

Python II.4: Affectations simultanées

C Échange de deux variables

L'affectation simultanée permet d'échanger facilement les valeurs de deux variables. C'est une spécificité de Python qui est très pratique[15].

```
a = 1
b = 5
(a,b) = (b,a)
print(a); print(b)
```

Python II.5: Échange de variables

3 Fonctions

À partir de maintenant, nous aurons besoin d'exécuter plusieurs lignes à la fois. Nous les écrirons donc d'abord dans la partie fichier avant de les compiler (par F5 ou F9).
Pour tester les fonctions ainsi écrites, on peut taper directement dans le shell.

Définition 3.1 :

Une **fonction** est une suite finie d'instructions qui reçoit un argument et renvoie un résultat.

Remarque : On utilise le terme de **procédure** pour désigner une fonction en informatique lorsqu'elle ne renvoie pas de résultats. Dans ce cours, nous assimilerons les deux termes[16].

La syntaxe est la suivante

```
def maFonction(arg):
  suite d'instructions
  suite d'instructions
  ...
  return resultat
```

Python II.6: Syntaxe d'une fonction

La fonction est donc composée :

- d'une ligne d'entête commencée par le mot clef **def** et terminée par deux-points,
- d'un bloc d'instructions indenté par rapport à la ligne d'entête
- d'une instruction **return** qui renvoie le résultat (aussi indentée)

Remarques :

- Pour éviter les problèmes d'indentation, il faut demander à Pyzo de transformer automatiquement les tabulations en espaces (depuis le menu).
  En général, on choisit une indentation de 2 ou 4 espaces. C'est à vous de choisir.

```
def␣maFonction(arg):
␣␣suite␣d'instructions
␣␣suite␣d'instructions
␣␣...
␣␣return␣resultat
```

Python II.7: Syntaxe d'une fonction avec les espaces

[15] En général, dans les autres langages il faut faire appel à une variable auxiliaire
[16] En Python, tout est fonction : si on ne demande pas de renvoyer une valeur, alors Python de lui-même renvoie la valeur `none`.

- Après **return**, la fonction ne lit plus rien dans le bloc. Toutes les instructions indentées qui suivent seront ignorées.

```
def plus1(nombre):
  nombre += 1           # ajoute 1 à la variable nombre
  return nombre
  print('blablabla')

# instructions pour tester à écrire dans le shell
plus1(4)
plus1(-7)
plus1(3.75)

b = 9
plus1(b)
print(b)
```

Python II.8: Fonction incrémentielle

Variables globales - variables locales (*plus subtile, peut être lu dans un second temps*)

On remarque que lorsque l'on applique une fonction à une (ou des) variable, Python crée une **nouvelle variable locale** pour ne pas modifier la précédente. Cette variable *locale* n'a d'existence qu'au sein du bloc d'instructions.

Remarque pour les pros : La méthode d'affectation pour les listes fait que la variable locale et la variable globale, bien de différentes, pointent souvent vers le même objet mémoire contenant la liste. Ainsi, la modification de la variable locale affecte aussi la variable globale.
Nous verrons plus loin comment contourner cette difficulté ou au contraire l'utiliser à notre profit.

```
def empile(liste,elt):
  liste += [elt]
  return liste

empile([4],2)
empile([4],[1,2])
liste=[]
empile(liste,4)
liste               #la liste a été modifiée
```

Dans le corps de la fonction, Python, peut utiliser sans problème toutes les variables globales qui ont été définies.

Par contre, dès qu'il définit une nouvelle variable à l'intérieur de la fonction, cette variable est **locale** : cette variable n'a d'existence qu'à l'intérieur de la fonction.

Il est possible de forcer Python à définir une variable **globale** avec le mot clef `global`.

```
def createALocal():
  A = 5
def createA():
  global A
  A = 5

# dans le shell
A = 4

createALocal()
print(A)      # affiche 4 : la variable globale n'est pas modifiée.

createA()
print(A)      # affiche 5 : la variable globale est modifiée.
```

Python II.9: Définition d'une variable globale au sein d'une procédure.

4 Import de bibliothèques

Au gré des besoins, des programmeurs ont crée de très nombreuses fonctions que l'on peut rajouter à Python. Ils ont placé ces fonctions au sein de **bibliothèques** ou librairies.
Pour utiliser ces fonctions, il faut charger la librairie concernée.

```
1
2 pi                      # pi n'est pas défini dans Python par défaut -> erreur
3
4 # importer une bibliothèque, mais sans 'sortir' ses objets
5
6 import math              # on importe la bibliothèque math
7 pi                      # pi n'existe toujours pas de Python -> erreur
8 math.pi                 # pi existe dans la bibliothèque math-> donne valeur approché
      e.
9
10 # importer un objet d'une bibliothèque et le 'sortir'
11 from math import pi     # pour importer la commande pi de la bibliothèque math
12 pi                      # -> donne valeur approchée
13 cos(pi)                 # commande cos non définie -> erreur
14
15 # importer tous les objets d'une bibliothèque et les 'sortir'
16 from math import *       # pour tout importer de la bibliothèque math
17 cos(pi)
```

Python II.10: Import d'une bibliothèque

Parfois, lorsque l'on importe plusieurs bibliothèques, on peut avoir des conflits entre des fonctions qui sont définies différemment selon la bibliothèque. il est alors préférable d'importer la bibliothèque suivant la première procédure : `import <bibliothèque>`.
Mais à chaque fois que l'on utilise une fonction de cette bibliothèque, il faut alors la faire précéder du nom de cette bibliothèque et d'un point. Comme c'est un peu long, on donne alors un *alias*, un surnom à cette bibliothèque.

```
1 randint(1,10)   #nombre entier aléatoire entre 1 et 10 (inclus)
2 import random
3 random.randint(1,10)
4
5 import random as rnd
6 rnd.randint(1,10)
```

Python II.11: Import d'une bibliothèque avec un alias

Bibliothèque	Description
math	Outils de base en mathématiques
random	Nombres (pseudo-)aléatoires
numpy	Tableaux
matplotlib	Outils graphiques
pylab	*méta*-bibliothèque qui en contient beaucoup d'autres

Table II.1: Bibliothèques usuelles

Revenir au sommaire

III - TYPES DE DONNÉES

Avertissement : *Ce chapitre peut être sauté en première lecture pour être lu de façon fractionnée en fonction des besoins.*

Sous Python, chaque objet possède un **type** qui désigne la nature de cette objet. Une des premières choses que vérifie Python lorsque vous lui soumettez un code est que vous ne mélangez pas les torchons et les serviettes. N'essayez donc pas d'additionner des entiers avec des chaînes de caractères : il refusera.

En Python, une variable est **typée**. Son **type** correspond à sa nature : un entier, un flottant, une chaîne de caractère. Lors de l'exécution du programme, le langage vérifie d'abord les types de données avant d'effectuer les opérations.
Python possède un *typage dynamique*, cela signifie que le type de chaque variable est deviné par la machine au moment de l'exécution et vous n'avez pas besoin de le déclarer vous même. C'est plus rapide et moins lourd à écrire, mais demande de faire plus attention, d'être rigoureux et de toujours **se poser la question de la nature des objets que l'on manipule** (comme en mathématiques, vous le verrez plus tard).
Le type d'une variable peut être modifié au cours du programme comme on l'a montré dans l'exemple plus haut.

A contrario, un langage comme C++ utilise un *typage statique* qui est déclaré par l'utilisateur dans le code source. Cela permet au compilateur de détecter les erreurs de typage plus tôt et d'optimiser l'allocation mémoire. En revanche cela donne plus de rigidité au code et alourdit la programmation. Le type d'un objet s'obtient par la commande `type(`*`objet`*`)`.

Nous allons faire ici la revue des principaux types proposés par Python.

A Les types numériques

Les entiers relatifs : `int`
Depuis Python 3, il n'y a pas de limite de taille pour l'entier[17] : l'entier peut être aussi grand que l'on veut dès lors que cela tient dans la mémoire.

Les nombres décimaux : `float`
La virgule est représentée par un point (notation anglo-saxonne). Les flottants possèdent un nombre limité de chiffres après la virgule. N'imaginez donc pas que vous faites des calculs exacts[18].

Les nombres complexes : `complex`
Les nombres complexes sont codés sous forme cartésienne (couple de flottants). Le nombre imaginaire i est désigné par la lettre `j`. On les écrit sous la forme `a+bj`.
N'oubliez pas le `b` *même s'il vaut* 1.

[17] Ce n'est pas le cas dans beaucoup d'autres langages.
[18] En fait, Python fait une approximation du flottant en fraction binaire avant de réaliser les calculs. Ceci peut conduire a des résultats surprenants même pour des valeurs avec peu de décimales.

```
type(5)
type(-3)
type(0)
type(5.)
type(5.09934)
type(4,34)       #renvoie une erreur
type(5+3j)
type(2j)
type(3+j)        #renvoie une erreur
type(2-1j)
```

Python III.1: Types numériques

Méthodes sur les complexes
Python est un langage orienté objet. Chaque objet possède donc des méthodes qu'il hérite de sa classe.
Nous n'allons pas rentrer dans les détails, mais vous reconnaîtrez les méthodes à la syntaxe utilisée : un point suivi du nom de la méthode.

```
a=3+2j
a.real
a.imag
a.conjugate()
```

Python III.2: Méthodes sur les complexes

Remarque : Lorsque dans Pyzo,vous tapez le nom de la variable suivi du point, il vous propose les méthodes disponibles (beaucoup plus nombreuses que celles présentées ici).

Opérations usuelles
Les opérations usuelles sont proposées par Python :

Opérateur Python	Opération mathématique
`x+y`	Somme de x et y
`x-y`	Différence de x et y
`x*y`	Produit de x et y
`x/y`	Quotient de x par y
`x**y`	x à la puissance y
`abs(x)`	Valeur absolue de x
`x//y`	Division *entière* de x par y
`x%y`	Reste de la division *entière* de x par y

Table III.2: Opérations numériques usuelles

Remarque : Python ne fait pas de calculs exacts, il introduit donc des erreurs d'approximation dans certains calculs. Il faut en tenir compte.
Dans le dernier exemple du listing qui suit, on demande à Python d'écrire 0,1 avec 20 décimales dernière la virgule. Si vous essayez de taper ce code, vous verrez que Python a introduit des décimales non nulles plus loin dans le développement décimal. Cela vient du fait qu'il enregistre les nombres en format binaire. Lorsque Python convertit 0,1 en binaire, comme l'écriture ne "tombe pas juste", cela provoque une petite erreur d'approximation.

- Pour la puissance, n'utilisez pas le caractère "^". Par convention, $0^0 = 1$.

- Les *puissances de* 10 sont codées avec la lettre `e` en minuscule. Ainsi 5000 peut s'écrire $5e3$.
  La puissance est entière.

- Python accepte de faire des opérations entre des éléments numériques qui n'ont pas le même type. Il prendra alors le type le plus général pour le résultat.

```
1e-20      #renvoie 1e-20
1e-20+1-1 #renvoie 0, cela vient des erreurs d'arrondis
1e-20+(1-1) #renvoie 1e-20, l'addition n'est pas associative chez les serpents
5+5        #int
5+3.       #float
3.*(5-2j) #complex
5/3        #float
13//4      #int
13.0//4.0 #float
13.2%4.0 #float

#Le code suivant n'est pas à connaître.
#écrit l'approximation Python de 0.1 avec 20 décimales
"{0:.20f}".format(0.1)
```

Python III.3: Opérations sur les nombres

Incrémenter une variable
Supposons à présent que dans votre programme, vous savez que la variable "a" contient un nombre. Vous cherchez simplement à augmenter sa valeur de k, on dit incrémenter. Par exemple, pour incrémenter de 1 :

- si la variable "a" pointe vers 5 au début, vous voulez qu'elle pointe $5+1=6$ ensuite,
- si elle pointe vers -12 au début, vous voulez qu'elle pointe vers $-12+1=-11$ ensuite.
- ...

Il existe plusieurs manières de faire la manipulation. La plus naturelle consiste à remplacer la valeur de a par a + 1.

```
a=1
a=a+1
print(a)
```

Python III.4: Incrément simple

On peut aussi utiliser une écriture plus ramassée

```
a=1
a+=1
print(a)
a+=5
print(a)
a-=4
print(a)
a=+10          #Attention
print(a)
```

Python III.5: Incrément compact

B Les booléens

Le type `bool` correspond aux booléens : `True` (Vrai) et `False` (Faux). `True` est synonyme de 1 et `False` est synonyme de 0.
Nota : Python fait la distinction entre les majuscules et les minuscules. `true` n'est pas la même chose que `True`.

Les comparaisons entre éléments renvoient un booléen.

⚠ Le test d'égalité se fait avec un double égal ==, le simple égal est réservé à l'affectation (le test d'égalité ne modifie pas les variables, contrairement à l'affectation).
Nota : Il existe aussi les opérateurs `is` et `is not` pour déterminer s'il s'agit du même objet. C'est à dire si les deux variables pointent vers le même objet en mémoire.

On peut construire des expressions logiques à partir des liens logiques :

Un python, c'est fainéant (et nous aussi) :
Les opérateurs `or` et `and` ne lisent que le minimum d'information et s'arrêtent dès qu'ils peuvent conclure.

Opérateur	Signification
==	Est égal
!=	Est différent
<	Est strictement inférieur
>	Est strictement supérieur
>=	Est supérieur ou égal
<=	Est inférieur ou égal

Table III.3: Opérateurs de comparaison

Opérateur	Signification
`or`	OU logique
`and`	ET logique
`not`	NON logique

Table III.4: Opérateurs logiques

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

Table III.5: Tables de vérité

Ainsi, dans l'expression ''`a or b`'', Python ne lit l'expression `b` que si `a` vaut `False`. Sinon, il renvoie directement `True`.
De même, pour "`a and b`", Python ne lit l'expression `b` que si `a` vaut `True`. Sinon, il renvoie directement `False`.
Cette subtilité est à garder en mémoire, car cela permet parfois de simplifier des écritures. Par exemple "`a==0 or abs(b/a)>100`" est une expression valide car la division n'est effectuée que si `a==0` renvoie `False`.

Remarque : L'opérateur `not` n'est pas prioritaire par rapport aux opérateurs de comparaisons. N'hésitez pas à utiliser des parenthèses pour écrire les expressions logiques et éviter des erreurs dans les priorités.

On peut mettre des comparaisons les unes à la suite des autres, le mot clef `and` est alors implicite.
Par exemple `a<b<c!=d` veut dire `a<b and b<c and c!=d`.

```
1==0       #False
1==1       #True


(1==1)+1   #convertit True en 1
1==1+1

5.3>2      #convertit tout en float
2.0==2

3<4<6>4
3<4<6>(5/0)
3<4>6>(5/0)

#Le not n'est pas prioritaire et calculé à la fin
not (1==1)*(2==5)

#ici le résultat diffère car not est dans la parenthèse
(not 1==1)*(2==5)

```

```
#Seul le 0 vaut False
bool(0)   #convertit en booléen
bool(1)
bool(-2.5)
```

Python III.6: Tests logiques

C Les chaînes de caractères

Les chaînes de caractères servent au texte, elles sont désignées par le type `str`.

Guillemets
Elles sont encadrés par des guillemets simples ou doubles. Ils ont le même rôle, vous devez simplement ouvrir et fermer la chaîne avec les mêmes types de guillemets.

```
type("Hello World !")
type('Hello World !')
"trop"=='trop'   #trop, c'est trop
```

Python III.7: Chaînes de caractères

Si on choisit des guillemets doubles, on peut mettre des apostrophes (guillemets simples) dans la chaîne de caractères sans que cela ne pose de problèmes. Par contre, si on veut mettre des doubles guillemets, il faut les *échapper* avec l'antislash : \".
De même si on encadre par des guillemets simples.

```
"J'aime Python"
'J'aime Python'   #erreur
"J'ai dit \"J'aime Python\""
'J\'ai dit "J'aime Python"'
```

Python III.8: Guillemets et échappement

Opérations sur les chaînes de caractères

Opération Python	Manipulation de la chaîne
`a+b`	Concaténation des chaînes `a` `b` (Colle deux chaînes bout à bout)
`len(s)`	Longueur de la chaîne `s` (nombre de caractères)
`n*s`	Répète n fois la chaîne `s`
`a in b`	appartenance de `a` à la chaîne `b`
`a not in b`	le contraire

```
serpent="Python"
poisson='thon'
len(serpent)     #Pas si long, finalement
poisson in serpent
'Boum'=='boum'    #attention à la casse majuscule/minuscule !
5*"idiot"         #restera différent de ''intelligent''
serpent=="Py"+poisson
```

Python III.9: Opérations sur les chaînes de caractères

Indexation et slicing (tranchage)
Dans une chaîne de caractères, les caractères sont numérotés **en commençant par** 0. On peut ainsi faire référence à certains caractères en particulier.
À la suite de la session précédente ,

```
serpent[0]
lg=ln(serpent)
serpent[lg-1]
serpent[lg]   #erreur, on est sorti de la chaîne car l'index commence par 0
```

On peut aussi faire référence à une partie de la chaîne entre deux indices `i` et `j` (`j` est exclu) : `[i,j]`.

```
serpent[0:2]     #caractères 0 et 1

#''l'oubli'' du premier indice le met automatiquement à 0
serpent[:2]      #caractères 0 et 1

serpent[2:5]     #caractères 2,3 et 4

serpent[2:lg]==poisson
#''l'oubli'' du deuxième indice le met automatiquement au maximum
serpent[2:]

serpent[:]

planete=serpent[:1]+'lut'+serpent[4:]
```

Python III.10: Slicing des chaines de caractères

Pour prendre qu'une lettre sur deux, on rajoute un troisième argument qui vaut 2, pour une lettre sur trois, ce sera 3.

```
serpent[0:lg:2]
serpent[::2]
serpent[1:lg:2]
serpent[1::3]
```

Python III.11: Slicing des chaines de caractères - pas

Enfin, on peut aussi compter à partir de la fin avec les nombres négatifs.

```
serpent[-1]      #dernier élément
serpent[-2]      #avant-dernier élément
serpent[::-1]    #lit à l'envers
```

Python III.12: Slicing des chaines de caractères - indices négatifs

Syntaxe	Signification
`chaine[i]`	$i^{\text{ème}}$ élément, on compte à partir de 0 Si i est négatif, compte à partir de la fin.
`chaine[i:j]`	Sous liste des éléments i (inclus) à j (exclu) Si i est vide, alors il est remplacé par 0 (début de la chaîne) Si j est vide, alors il est remplacé par `len(chaine) (fin de la chaîne)`
`chaine[i:j:k]`	Sous chaîne des éléments i (inclus) à j (exclu) avec un pas de k k peut être négatif pour compter à rebours.

Table III.6: Parcours de la chaîne et extraction

D Les listes

Les listes sont du type `list`. Dans le principe, elles sont très semblables à des chaînes de caractères, ou chaque caractère est indexé par sa position, mais possède un type quelconque (`int`, `float`, `str`,..., ou même `str`). Les éléments d'une liste n'ont pas tous forcément le même type.

Une liste s'écrit entre crochets avec des éléments séparés par des virgules.
Les éléments sont indexés et on peut y avoir accès comme pour les chaînes de caractères.

```
liste=[1,'deux']
liste
type(liste)
liste[0]       #élément d'indice 0
```

```
liste+=3       #erreur : on ne peut ajouter un entier à une liste
liste+=[3]     #c'est mieux !
liste[1:]      #sous liste
```

Python III.13: Listes

Attention : Il faut faire attention à distinguer deux notations :

- `liste[2]` donne l'élément d'indice 2 de la liste,
- `liste[2:3]` donne la sous liste qui contient l'élément d'indice 2.
- `liste[2:2]` donne une sous-liste vide (placée en indice 2). On peut utiliser cette syntaxe pour insérer une sous-liste en position 2).

```
liste=[1,'deux',3]
liste
type(liste)
liste[2]         #renvoie l'entier 3
liste[2:3]       #renvoie la sous liste qui contient 3
type(liste[2])     #entier
type(liste[2:3])   #liste
liste=[]           #liste vide
liste
```

Python III.14: Élément d'une liste versus sous liste

Modification d'une liste
Contrairement à une chaîne de caractères, une liste peut être modifiée de l'intérieure sans être recopiée.

```
liste=[1,'deux',3]
liste[2]=2
liste
liste[1:3]=5       #erreur, ce doit être une sous-liste
liste[1:3]=[5]
liste
liste[1:3]=[2,3,4,5,6]
liste
liste[0]=['do','re','mi'] #liste dans la liste
```

Python III.15: Modification d'une liste

Pour montrer la proximité entre les listes et les chaînes de caractères, on peut voir qu'une chaîne peut être interprétée comme une liste. Comparez les deux instructions :

```
liste=[1,2]
liste[0]='chaine'
liste
liste[0:1]='chaine'
liste
```

Python III.16: Les chaînes et les listes sont *iterables*

Dans le premier cas, on change l'élément 0 de la liste pour le remplacer par la chaîne de caractères `chaine`.
Dans le deuxième cas, on change une sous liste, il faut donc la remplacer par une sous-liste. Python interprète alors la chaîne de caractères comme une liste de caractères qu'il rentre les uns à la suite des autres.

Méthodes sur les listes
Ces méthodes **modifient** la liste.

```
liste=[1,2]
liste.append(3)
liste
liste.pop()
```

Méthode	Action correspondante
`liste.append(a)`	Ajoute l'élément "a" en fin de liste
`liste.pop()`	Enlève le dernier élément et le renvoie en résultat
`liste.pop(i)`	Enlève l'élément i de la liste et le renvoie en résultat
`liste.index(x)`	Renvoie l'indice de la première occurrence de x dans la liste. Renvoie une erreur s'il n'y est pas.
`liste.count(x)`	Compte le nombre d'occurences de x dans la liste.
`liste.reverse()`	Retourne la liste.
`liste.sort()`	Trie la liste (utiliser pour les types numériques).

Table III.7: Méthodes sur les listes

```
liste
liste.append(3)
liste.reverse()
liste
liste.append(7)
liste.sort()
liste
```

Python III.17: Méthodes sur les listes

Difficultés avec l'affectation

```
liste=[1,2,3]
liste2=liste
liste[0]='modif'
liste
liste2
liste2 is liste
```

Python III.18: Listes et affectations

On constate que l'élément 0 de **liste2** a aussi été modifié !
Ces subtilités nous amènent doucement vers les difficultés de l'affectation.
Dans l'exemple, lorsque l'on effectue l'affectation `liste2=liste`, Python fait pointer les deux éléments vers la même liste en mémoire (comme nous n'avons vu plus haut pour les types plus simples). C'est ce qui est confirmé par le test logique : `liste2 is liste`.
Lorsque `liste[0]` est modifiée, ce n'est pas tant l'adresse mémoire de la liste qui est modifiée, qu'un élément à l'intérieur de cette liste. On modifie ainsi le pointeur d'indice 0 dans la liste, mais la liste elle-même garde le même emplacement mémoire. `liste2` qui pointe vers le même emplacement est donc aussi modifiée.
Cela reste vrai si on ajoute ou supprime des éléments de la liste.

Si on veut imposer à Python de faire une copie de la liste sur un autre emplacement mémoire, on utilise la méthode `copy`. (On peut aussi le faire avec une boucle **for**).

```
liste=[1,2,3]
liste2=liste.copy()
liste2 is liste
liste[0]='modif'
liste
liste2
```

Python III.19: Copie d'une liste

E Les dictionnaires

Dans une liste, les éléments sont indexés par des entiers de 0 à n.
Un dictionnaire correspond à une liste dont on choisit les indexes (qui ne sont pas nécessairement entier) que l'on appelle clef (key). Un dictionnaire est de type `dict`.
Les dictionnaires sont entrés entre accolades. Chaque élément est représenté par un couple de la forme `clef:valeur`.

```
dico={"un":1,"deux":2,"trois":3,55:[5,'cinq']}
dico[1]          #erreur, ce n'est pas un indice.
dico[55]
dico[55][1]
dico['un']=0       #modification
dico
del(dico['un'])    #suppression
dico
dico['dix']=10     #ajout
dico
len(dico)          #nombre d'éléments
dico.keys()        #liste des clefs
dico.values()      #liste des valeurs
```

Python III.20: Dictionnaires

Exemple d'utilisation concrète d'un dictionnaire :

```
coord={'abscisse':3.2,'ordonnee':-4.3}
coord[abscisse]
coord[ordonnee]
```

Python III.21: Dictionnaire : exemple concret

F Les tuples

Les tuples sont désignés par le type `tuple`. Ils fonctionnent comme des listes figées (que l'on ne peut pas modifier une fois construites : les tuples sont **non mutables**). On les distingue des listes par l'utilisation de parenthèses à la place des crochets.

```
t1=()                    #tuple vide
type(t1)
t2=(1,2,'trois')
type(t2)
t2[0]
t2[2]
len(t2)
t2[1:]                   #slicing
t2[2]=3                  #erreur : tuple non mutable
t2+t2                    #fonctionne car le tuple n'est pas modifié
t2
t2.append(4)             #erreur : tuple non mutable
t2.reverse()             #erreur : tuple non mutable
t2[::-1]                 #lit à l'envers mais ne modifie pas
t2
```

Python III.22: Tuples

Attention, les parenthèses servent aussi à séparer les expressions, à modifier les ordres de priorité...comme en maths. Ainsi, il n'existe pas de tuples avec un seul élément.

```
t3=(17)
type(t3)                 #entier
t3
```

Python III.23: Tuples à un élément ?

G Conversion et lecture au clavier

Lorsque les expressions sont compatibles, on peut les convertir d'un type à un autre.

Type	Nature
`int`	Entier *de taille arbitraire*
`float`	Nombre décimal *dit nombre à virgule flottante*
`complex`	Nombre complexe
`str`	Chaine de caractères (texte)
`bool`	Booléen *(Valeur logique)*
`tuple`	Liste de longueur fixe
`list`	Liste de longueur variable

Table III.8: Quelques types de données en Python

```
lettre='1'
type(lettre)
entier=int(lettre)
type(entier)
flottant=float(entier)
type(flottant)
list(lettre)

#Attention
print(str(5))
print(str(float(5))
```

Python III.24: Conversion du type

Pour lire une saisie au clavier, on utilise la commande `input`. Par défaut, `input` renvoie une chaîne de caractères.

```
entree=input('entrez un entier')
entree+1                          #erreur car...
type(entree)
int(entree)+1
entree=float(input('entrez un nombre'))
type(float)
```

Python III.25: Lecture au clavier

Revenir au sommaire

IV - STRUCTURES CONDITIONNELLES

1 Branchement conditionnel

A Test simple

La structure avec `if` est la traduction (littérale) de la structure conditionnelle mathématique :

Si *condition,*	**if** `condition`:
Alors *instructions si la condition est vraie,*	`bloc d'instructions si la condition est vraie,`
Sinon *instructions si la condition est fausse.*	**else**:
	`bloc d'instructions si la condition est fausse.`
suite du programme en dehors de la boucle	`suite du programme en dehors de la boucle`

*Le "**alors**" (**then**) est sous-entendu. Il ne faut pas l'écrire.*

```
1 age = int(input('Entrez votre âge '))
2 if age<18:
3   print('Vous être mineur')
4   print('Dans moins de '+str(18-age)+' ans, vous serez majeur')
5 else:
6   print('Vous êtes majeur')
```

Python IV.1: Exemple test if

Les **indentations**[19] déterminent les blocs. La partie **else** est optionnelle. Lorsque Python a fini les instructions du bloc qu'il doit exécuter, il passe à la suite du programme.

B Imbrication des blocs

Dans le programme, chaque bloc déterminé par un niveau d'indentation constitue une entité autonome.

Ainsi, dans un bloc **if**, on peut intégrer un autre branchement conditionnel sans problème, et ainsi de suite.

[19] L'indentation est le fait de décaler la ligne par rapport à la marge. Pour un bloc donné, on s'assurera que toutes les lignes ont la même indentation : le même décalage.

if <*condition 1*> :
Bloc d'instructions si la condition 1 est **vraie**
if <*condition 2*> :
Bloc d'instructions si la condition 2 est **vraie**
else:
Bloc d'instructions si la condition 2 est **fausse**
else:
Bloc d'instructions si la condition 1 est **fausse**
`Suite du programme...`

Si toute la structure conditionnelle précédente doit être intégrée au sein d'une fonction, alors, il suffit d'indenter tout ce bloc et de le faire précéder de la commande : **def**.

def <*nom fonction*>(<*arguments*>):
if <*condition 1*> :
Bloc d'instructions si la condition 1 est **vraie**
if <*condition 2*> :
Bloc d'instructions si la condition 2 est **vraie**
else:
Bloc d'instructions si la condition 2 est **fausse**
else:
Bloc d'instructions si la condition 1 est **fausse**
`Suite du programme...`
`Suite du programme...`

Ce fonctionnement par blocs est très explicite dans le langage AlgoBox.

Astuce Pyzo : chaque "deux points" ":" annonce une indentation. Lorsque l'on tape les deux points dans Pyzo, celui-ci indente automatiquement la ligne suivante.
Pour indenter tout un paquet de lignes à la fois, on les sélection et on appuie sur `Tab`, la touche tabulation (touche à gauche de la lettre *A* sur un clavier). Pour supprimer une indentation, on sélectionne et on appuie sur `Maj-Tab`.

C Utilisation de elif

Il est possible de spécifier plusieurs alternatives dans un branchement conditionnel. Cela peut éviter d'imbriquer trop de branchements les unes dans les autres. On utilise pour cela la commande **elif** autant de fois que nécessaire. Cette commande se traduit par "sinon, si...".

```
if expressionBooleenne:         #si..., faire
  suite d'instructions
  suite d'instructions
  ...
elif expressionBooleenne2:     #sinon, si..., faire
  suite d'instructions
  suite d'instructions
  ...
elif expressionBooleenne3:     #sinon, si..., faire
  suite d'instructions
  suite d'instructions
  ...
else:                           #sinon,
  suite d'instructions
  suite d'instructions
  ...
suite du programme en dehors du branchement
```

Python IV.2: syntaxe avec elif

Dès qu'une expression booléenne renvoie `True`, le bloc d'instructions correspondant est exécuté puis le programme sort du test (sans vérifier les conditions suivantes).

Si aucune condition n'est vérifiée, alors le programme exécute le bloc d'instruction de **else**.

```
def parite(nb):
  if type(nb) != int:
    print("Un nombre entier ! espèce de #&!@*#!")
  elif nb%2 == 0:
    print("Le nombre choisi est pair")
  else:
    print("Le nombre choisi est impair")
```

Python IV.3: Utilisation de elif

Remarque : cette fonction ne renvoie rien (ou `none`), elle se contente d'afficher un texte à l'écran.

Exercice : quelle est l'erreur dans le programme qui suit ?

```
def parite(nb):
  if type(nb) != int:
    print("Un nombre entier ! espèce de #&!@*#!")
  elif nb%2 == 0:
    print("Le nombre choisi est pair")
  elif nb == 0:
    print("Le nombre choisi est nul")
  else:
    print("Le nombre choisi est impair")
```

Python IV.4: Utilisation de elif, erreur dans l'ordre des instructions

Solution :
La condition `nb==0` ne sera jamais testée, car si `nb` vaut zéro, alors la condition `nb%2==0` renverra `True`, le programme affichera que le nombre est pair puis sautera les instructions suivantes. Si on veut que le cas `nb==0` soit testé à part, il faut mettre cette condition plus tôt.

```
def parite(nb):
  if type(nb) != int:
    print("Un nombre entier ! espèce de #&!@*#!")
  elif nb == 0:
    print("Le nombre choisi est nul")
```

```
  elif nb%2 == 0:
    print("Le nombre choisi est pair")
  else:
    print("Le nombre choisi est impair")
```

Python IV.5: Utilisation de elif, correction de l'erreur dans l'ordre des instructions

D Comportement particulier de la commande return

Exemple (Maximum d'un couple)

Voici une fonction qui donne le maximum de deux nombres (sans utiliser la fonction max).

```
def maximum(a,b):
  if a>b:
    return a
  else:
    return b
```

Python IV.6: Maximum d'un couple

Essayez de comprendre pourquoi la deuxième fonction `maximum2` marche aussi.

```
def maximum2(a,b):
  if a>b:
    return a
  return b
```

Python IV.7: Maximum d'un couple

Nous avions vu que lorsque Python rencontre "**return**", il renvoie le résultat et sort de la fonction, quelque soit ce qui est écrit après.
Dans notre cas,

- si $a > b$, il réalise l'instruction "**return** `a`" puis sort de la fonction.
- sinon, il sort de la structure conditionnelle et suit l'instruction suivante : `return b`.

Cette fonction renvoie donc bien le maximum.

Exercice

Faites la même chose pour le maximum de trois nombres.

Remarques pour les pros : Lorsque le branchement conditionnel est simple, il est possible d'utiliser une écriture compacte qui correspond à ce que l'on dirait en français.

```
def maximum(a,b):
  return a if a>b else b
```

2 Boucle while

Syntaxe et utilisation

La boucle `while` consiste à réaliser des instructions tant qu'une assertion est vraie.

Tant que *condition est vraie,*	**while** `condition:`
Faire *instructions,*	`    bloc d'instructions si condition est vraie,`
Fin	
suite du programme	`suite du programme`

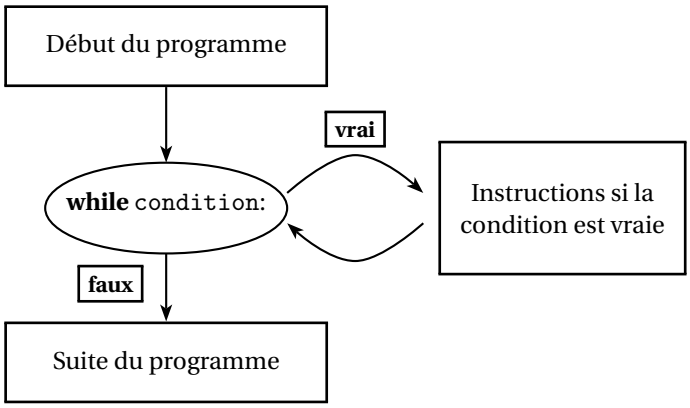


Figure IV.10: Fonctionnement de la boucle while

Lorsque l'assertion est fausse, le programme saute le bloc indenté.

Un premier exemple : voici un programme qui compte jusqu'à 10.

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

Python IV.8: Boucle while : compter jusqu'à 10

Voici le déroulement du programme :

- `i` prend la valeur 1
- `i<=10` est vrai, donc les instructions de la boucle sont effectuées
  - Affiche `i` : 1
  - Incrémente `i` de 1 : `i=1+1=2`
- Les instructions de la boucle sont terminées, on teste à nouveau la condition `i<=10`. Comme `i=2`, l'assertion est vraie et les instructions de la boucle sont effectuées.
  - Affiche `i` : 2
  - Incrémente `i` de 1 : `i=2+1=3`
- L'assertion est à nouveau testée, elle vaut `True`...
- Et ainsi de suite, jusqu'à ce que `i` vaille 11, alors l'assertion `i<=10` vaut `False` et le programme saute les instructions de la boucle.
- Le programme affiche "fini !"

Pour complexifier, on peut créer une fonction pour compter jusqu'à n (pas forcément 10). On mettra donc la boucle `while` dans une fonction, ce qui revient à indenter tout le bloc.

```
def compter(n):
  i = 1
  while i <= n:
    print i
    i += 1
  print('fini !')
```

Python IV.9: Boucle while : fonction compter jusqu'à n

Attention aux erreurs

Pour que le programme termine (que l'on sorte un jour de la boucle **while**, il faut que la condition évolue à chaque tour par l'intermédiaire d'une variable que l'on modifie[20]. Dans l'exemple précédent, c'est la variable `i` que l'on incrémente. Si on oublie la ligne correspondante, alors la boucle ne termine jamais

Astuce Pyzo : Lorsque vous testez vos fonctions, il est possible de forcer Python à s'arrêter en cas de boucle infinie (bouton en forme d'éclair au dessus du shell sur Pyzo).

Remarque pour les pros : Pour démontrer mathématiquement qu'une boucle termine, on définit un *variant de boucle*. À chaque occurrence de la boucle, ce variant doit augmenter d'au moins 1 et on doit prouver que la boucle s'arrête lorsque l'incrément prend une valeur supérieure à un certain nombre fixé.
Dans l'exemple (celui qui fonctionne), l'incrément est simplement i.
Mauvaise nouvelle : le théorème de Rice énonce qu'il n'existe pas d'algorithme qui puisse prouver la terminaison de n'importe quel programme.

Exemple

Indiquez ce que fait l'algorithme suivant.
La terminaison de cet algorithme n'a pas été prouvé. Elle est connue sous le nom de conjecture de Syracuse.

```
def syracuse(nb):
  while nb != 1:
    if nb%2 == 0:
      nb = nb/2
    else:
      nb = 3*nb+1
```

Python IV.10: Conjecture de Syracuse

Astuce de pros : Lorsque l'on met une boucle **while** dans une fonction, on peut arrêter la boucle par la commande **return**.
Par exemple (programme très maladroit, mais permet de comprendre) :

```
# renvoie le nombre de coups pour trouver a
def trouver(a):
  b = 1
  n = 0
  while True:
    n += 1
    if b == a:
      return n
    if b < a:
      b = b+1
    else:
      b = b-1
```

Revenir au sommaire

[20] Il existe une exception avec l'utilisation de **return**.

V - BOUCLE FOR

1 Boucle for et commande range

A Un premier exemple

La boucle **for** est comme une boucle `while` mais dans laquelle l'incrément de boucle est intégré : on n'a pas à s'en occuper. Python se charge lui-même d'initialiser et d'incrémenter le variant de boucle. En particulier, on n'aura pas de boucles infinies avec **for**.

Si on veut écrire les nombres de 1 à 5, on peut programmer la boucle **while** :

```
i=1
while i <= 5:
  print(i)
  i += 1
print('fini')
```

Python V.1: Syntaxe avec une boucle while

Pour le traduire avec une boucle **for** :

```
for i in range(1,6):
    print(i)
print('fini')
```

Python V.2: Syntaxe avec une boucle for

range(n,p) est l'écriture en Python de l'intervalle $[\![n,p[\![= [\![n,p-1]\!]$.
⟶ la première borne n est comprise (incluse), mais pas la deuxième (exclue).

Ainsi on peut traduire en français la première ligne :

`for i in range(1,6)` **pour** i variant dans l'intervalle $[\![1,6[\![$.

Une telle écriture sous-entend que i commence par la valeur 1 et termine à $6-1=5$. À chaque boucle, la valeur de i est incrémentée de 1.

B Compléments sur la commande range

Commencer à 0 :

Lorsque l'on commence à compter à partir de 0, ce n'est pas la peine de le préciser et **range** n'a alors qu'un seul argument : la deuxième borne (qui est exclue)

```
#compte de 0 à 10
for i in range(11):      # idem que : for i in range(0,11):
    print(i)
print('fini')
```

Python V.3: range à un seul argment

Remarque : on comprend pourquoi la deuxième borne est exclue. Ainsi, dans `range(11)`, le compteur va de 0 à 10, la boucle est parcourue 11 fois.
Il arrive que la valeur exacte du compteur n'ai pas d'importance sur le résultat, mais que seul compte le nombre de fois que la boucle est effectuée. Dans ce cas, il suffit d'écrire `range(n)` pour que la boucle soit effectuée n fois.

Exemple

Programmer une suite de commande qui écrit 1000 fois à l'écran : "J'aime Python".

Solution :

```
for i in range(1000):
    print("J'aime Python")
```

Exemple (Corriger l'erreur)

Pour calculer $n!$ on propose le programme suivant :

```
# trouver l'erreur :
def factoriel(n):
    fact = 1
    for i in range(n+1):
        fact *= i
    return fact
```

Python V.4: Calcul de la factorielle n, énoncé faux

Quelle est l'erreur ? Corriger.

Solution :

i varie entre 0 et n et à la première boucle, on multiplie `fact` par 0. `fact` vaudra alors 0, et sa valeur ne changera plus jusqu'à la fin de la boucle (0 multpliplié par n'importe quel nombre donne toujours 0).
Il faut donc commencer par `i=1`, voire même `i=2` pour ne pas perdre de temps avec une boucle inutile.

```
# version corrigée
def factoriel(n):
    fact = 1
    for i in range(2,n+1):
        fact *= i
  return fact
```

Python V.5: Calcul de la factorielle n, corrigé

Avancer par sauts :

Il est possible de compter avec un pas de k, en spécifiant la valeur de k en troisième argument.

```
# compte de 0 à 10 de 2 en 2
for i in range(0,11,2):
    print(i)
print('fini')
```

Python V.6: range avec un pas fixe

Résumé sur la commande range :

Syntaxe	Signification
`range(`n`)`	entiers de 0 à $n-1$
`range(`p`,`n`)`	entiers de p à $n-1$ (1)
`range(`p`,`n`,`k`)`	entiers de p à $n-1$, avec un pas de k (2)

Notes :

- (1) Si $p \geq n$, alors `range` est vide (mais ne renvoie pas d'erreurs).
  p et n peuvent être négatifs.
- (2) k peut être négatif, pour un parcours décroissant.

Remarques pour les pros :
La variable d'itération est une variable globale : lorsque l'on sort de la boucle, elle garde la dernière valeur prise.

```
for i in range(11):
    pass                    #instruction vide : ne fait rien
print('Hors de la boucle, i vaut ',i)
```

Python V.7: Variable d'itération hors de la boucle

Nota : le bloc d'instruction doit contenir au moins une instruction. Ici, comme on ne veut rien faire, on lui donne l'instruction vide (ou instruction nulle) avec la commande Python `pass` (ne rien faire).

C Quel type de boucle choisir ?

Vaut-il mieux programmer avec une boucle **for** ou une boucle **while** ? Ça dépend.

De manière générale, si on sait programmer avec une boucle **for**, alors il est facile d'écrire avec une boucle **while**, le contraire est faux.

La boucle **while** est plus souple et plus générale, mais aussi plus lourde à programmer.

On cherchera toujours à privilégier la boucle **for**.

Plus simplement :

- J'utilise une boucle **for** lorsque je sais à l'avance combien de fois devra tourner la boucle.
  (en général cela correspond au nombre de calculs je dois réaliser pour arriver au résultat.)
- Dans les autres cas, j'utilise une boucle **while**.

boucle while	BOUCLE FOR
Avantages : Très souple et très général → on peut avoir une condition d'arrêt très compliquée qui dépend du calcul réalisé dans la boucle, → on n'est pas obligé de savoir à l'avance combien de fois sera réalisée la boucle.	**Avantages : simple et robuste** → pas de risque de boucle infinie, → programmation souvent plus succincte et *plus élégante*, → davantage dans *l'esprit Python*.
Inconvénients → gérer le compteur à la main, → risque de boucles infinies, → souvent un peu *lourd* à programmer.	**Inconvénients** → moins général.
⚠ il faut **initialiser** et **incrémenter** le compteur	**for** le fait tout seul.

Exemple

- Pour calculer le 100ème terme d'une suite définie par récurrence, j'utilise une boucle **for**.
  En effet, je sais à l'avance que je devrai faire tourner ma boucle exactement 100 fois.

- Pour calculer le premier terme d'une suite qui est plus grand que 100, j'utilise une boucle **while**.
  En effet, je ne sais pas à l'avance à quel terme cela correspond : u_5, u_{100}, u_{845} ? et je ne sais donc pas à l'avance combien de calculs je dois réaliser.

Avertissement : la suite de ce chapitre fait intervenir les listes et les chaînes de caractères.
Si vous n'avez pas encore lu les sections correspondantes dans le chapitre sur les types de données... alors c'est le moment.

2 Objets itérables

range est ce que l'on appelle un objet *itérable*. Cela veut dire qu'on peut le parcourir et c'est justement ce que fait la commande **for** : elle parcours l'intervalle défini pas la commande **range**.

Ceci peut être généraliser à tout autre objet que l'on peut parcourir de la même façon : on parle d'objet *itérable*.

La commande for parcourt un objet itérable.

Il existe d'autres objets itérables que l'on peut parcourir avec une boucle **for**. En particulier les listes et les chaînes de caractères.

```
def parcoursObjet(objet):
    for i in objet:
        print(i)

# utilisation
texte = 'abcdef'
parcoursObjet(texte)  # écrit les lettres les unes à la suite des autres

liste = [1,2,3,4]
parcoursObjet(liste)  # écrit les éléments de la liste les uns à la suite des autres
```

Python V.8: Parcours d'objets iterables

Ici, le compteur i prend directement les valeurs successives de l'objet (soit les lettres, soit les éléments de la liste).

On peut aussi faire les choses *à la main* avec la commande **range**, mais c'est beaucoup moins élégant et à éviter si on peut faire autrement.

```
# Parcours avec range
def parcoursObjet(objet):
    lg = len(objet)
    for i in range(lg):
        print(objet[i])
```

Python V.9: Parcours d'objets iterables, alternative avec range

Parcourir l'objet ou ses indices ?

On cherchera toujours à privilégier la boucle **for** sur l'objet.

itération sur les indices	ITÉRATION SUR L'OBJET
Avantages : Très général → Contient davantage d'informations : → on n'est pas obligé de savoir à l'avance combien de fois sera réalisée la boucle.	**Avantages : simple et robuste** → pas de risque d'erreur sur les indices, → plus simple à programmer, → plus succinct et plus élégant, → davantage dans *l'esprit Python*.
Inconvénients → risque de se tromper sur le `range`, oublier des indices ou sortir de la liste. → syntaxe plus lourde.	**Inconvénients** → on ne sait pas où on est dans la liste : l'indice n'est pas accessible.
⚠ si on utilise **range**, on risque toujours de se tromper dans les indices (d'aller trop loin, pas assez...).	

3 Listes en compréhension

Pour construire des listes, on est souvent amené à utiliser des boucles, ce qui amène à des syntaxes un peu lourdes. La compréhension de liste est une manière très compacte de construire de telles listes. Elle consiste à écrire la boucle dans la liste elle-même [21].

Exemple (Liste de carrés)

On cherche à faire la liste des carrés de x pour x variant de 1 à 10.

```
# méthode basique
carres = []
for x in range(1,11):
    carres.append(x**2)
print(carres)

#avec compréhension de liste
[x**2 for x in range(1,11)]
```

Python V.10: Listes en compréhenion

L'écriture en compréhension correspond exactement à ce que l'on dit en français " les carrés de x pour x variant de 1 à 10".

Exercice

À partir d'une liste de nombres `liste`, construire la liste des mêmes éléments incrémentés de 1.

Remarques pour les pros :
Lorsque la variable d'incrémentation n'est pas utilisée pour construire le résultat, on peut la remplacer par un caractère souligné "`_`".
Par exemple `[1 for _ in range(10)]`.

4 Complément : ruptures de séquences et de boucles

L'existence de ces possibilités est intéressante à connaître, mais à utiliser avec la plus grande réserve. Leur utilisation traduit souvent une programmation maladroite.

Ce passage peut être sauté sans préjudice majeur.

[21] C'est une notation : in fine, la boucle est exécutée et la liste ne contient pas cette boucle à proprement parler.

A Commande break - else

La commande **break** permet de sortir d'une boucle avant la fin.
La commande **else** au niveau d'une boucle **for** permet de donner un code à exécuter après la boucle, sauf si on est sorti par un **break**.

Voici un exemple tiré du tutoriel Python sur le site officiel :

```
for n in range(2, 10):
  for x in range(2, n):
    if n % x == 0:
      print(n, 'equals', x, '*', n/x)
      break
  else:
    # loop fell through without finding a factor
    print(n, 'is a prime number')
```

Python V.11: Commandes break-else - exemple tutorial Python

Cette boucle recherche les nombres premiers compris entre 2 et 9.
Pour chaque nombre n compris entre 2 et 9, elle teste tous les diviseurs possibles $x \in [\![2, n-1]\!]$.
Si x divise n alors le programme donne le produit en question et sort de la boucle **for** du x.
Sinon, à la fin de la boucle, Python effectue le **else** (qui est au niveau du **for** et pas du **if**).

Résumé : le **else** est exécuté en fin de boucle **for**, sauf dans le cas où une commande **break** fait sortir *artificiellement* de la boucle.

Remarque : dans le cas d'une boucle à l'intérieur d'une fonction, la commande **return** permet également d'arrêter le parcours de la boucle pour renvoyer directement la valeur. Contrairement à **break**, cette méthode *spécifique à Python* permet parfois des formulations élégantes. Il nous arrivera de la proposer.

B Commande continue

La commande **continue** permet d'ignorer la suite des instructions dans la boucle et de passer directement à l'incrément suivant.
Voici à nouveau un exemple issu du tutoriel.

```
for num in range(2, 10):
  if num % 2 == 0:
    print("Found an even number", num)
    continue
  print("Found a number", num)
```

Python V.12: Commande continue - exemple issu du tutoriel Python

Revenir au sommaire

VI - ALGORITHMIQUE AVEC LES LISTES ET CHAÎNES DE CARACTÈRES

Le but de ce chapitre est d'apprendre à programmer quelques algorithmes élémentaires qui font intervenir les listes et les chaînes de caractères.
Beaucoup de ces algorithmes sont déjà implémentés dans Python. Leur programmation est donc un *exercice* qui n'a d'utilité que pédagogique (et qui vous aidera à programmer d'autres fonctions plus compliquées). Si vous aviez besoin d'utiliser une telle fonction dans un projet, il faudrait utiliser celle de Python si elle existe et ne pas la reprogrammer vous-même.

Ces algorithmes sont des attendus du programme : À l'exception du tri à bulles, vous devez être en mesure de les programmer rapidement un jour d'oral.

1 Recherche dans une liste

A Recherche d'un élément quelconque dans une liste

Le but de cette question est de programmer manuellement certaines fonctions de recherche dans les listes.

Fonction "in"
`a in liste` teste l'appartenance de `a`, à la liste `liste`.

```
def appartient(a,liste):
    for i in liste:
        if i == a:
            return True
    return False
```

Python VI.1: Appartenance à une liste

On parcourt la liste avec une boucle **for**. Dès que l'on rencontre `a`, on arrête pour renvoyer `True`.
Si à la fin de la boucle **for**, on n'a toujours pas trouvé `a`, alors on renvoie `False`.

Remarque : Même programmée ainsi, nous utilisons l'opération **in**, on pourrait s'en passer en utilisant une boucle **while**.

Méthode "index"
Ce serait cet algorithme de recherche que vous devriez savoir refaire s'il n'y en avait qu'un.
On modifie un peu la fonction précédente pour que la nouvelle fonction :

- lève une erreur si `a` n'est pas dans la liste,
- sinon, renvoie le premier indice où se trouve `a`.

```
def indice(a,liste):
    for i in range(len(liste)):
        if liste[i] == a:
            return i
    raise ValueError(str(a)+" n'est pas dans la liste")
```

Python VI.2: Indice d'un élément dans une liste

Remarques :

- Cette fois-ci la boucle **for** sur l'objet itérable `liste` n'est pas suffisante car on perd la valeur de l'indice. Il faut donc itérer sur les indices avec `range(len(liste))`.
- On aurait pu écrire simplement `raise Exception(...)`, mais on a utilisé `raise ValueError(...)` pour avoir le même type de comportement que la méthode Python.

Méthode "count"
On compte le nombre de fois que l'on trouve `a` dans la liste. Contrairement aux fonctions précédentes, on ne doit pas s'arrêter dès que l'on trouve la valeur, mais parcourir la liste jusqu'au bout pour voir si elle n'y est pas d'autres fois.

```
def compter(a,liste):
    count = 0
    for i in liste:
        if i == a:
            count += 1
    return count
```

Python VI.3: Nombre d'occurences d'un élément dans une liste

Exercice

Programmez la fonction `indices` qui renvoie la liste de tous les indices où se trouve `a`. Si `a` n'est pas dans `liste`, alors la fonction renvoie la liste vide.

B Recherche du maximum d'une liste

On veut implémenter la fonction `max` pour les listes de nombres. On supposera la liste non vide.

```
def maximum(liste):
    maxi = liste[0]
    for m in liste:
        if m > maxi:
          maxi = i
    return maxi

#Version alternative pour ne pas lire deux fois le premier élément
def maximum(liste):
    maxi = liste[0]
    for m in liste[1:]: #modifié
        if m > maxi:
            maxi = i
    return maxi
```

Python VI.4: Nombre d'occurences d'un élément dans une liste

Au fur et à mesure que l'on avance dans la liste, `maxi` correspond au maximum de la sous-liste déjà parcourue.

Lorsque l'on est arrivé au bout, `maxi` correspond donc au maximum de la liste complète.

Ce type de raisonnement qui consiste à *"grignoter"* la liste est souvent utilisé. Nous le retrouverons dans le tri des listes.

Cet argument nous permet de prouver formellement par récurrence que notre algorithme se termine et renvoie le bon résultat.

Preuve :

On fait une récurrence sur la longueur de la liste pour montrer que la boucle se termine au bout de n étapes et qu'à l'issue de la boucle, `maxi` vaut bien le maximum de la liste.

- **Initialisation :** pour une liste de longueur 1.
elle n'a qu'un seul élément `liste=[a]`.
Au début `maxi` prend la valeur `liste[0]`.
On rentre dans la boucle : m prend la valeur `a`.
alors "`m>maxi`" est faux, et on passe à l'incrément suivant.
Le parcours de la liste se termine ici car elle ne contenait qu'un seul élément, donc on sort de la boucle `for`.
La boucle s'est donc terminée au bout d'une étape, et `maxi` vaut `a` et qui est bien le maximum.

- **Hérédité :** Soit $n \in \mathbb{N}^*$ quelconque fixé.
On suppose que pour toute liste de longueur n l'algorithme se termine en n étapes et qu'à l'issue de celles-ci, `maxi` vaut bien le maximum de la liste.

  Supposons que `liste` soit une liste de longueur $n+1$, alors on peut l'écrire sous la forme `liste=liste0.append(a)` avec `a` le dernier élément et `liste0` une liste de longueur n.

  La boucle parcourt d'abord `liste0`, et par hypothèse de récurrence,

  - la boucle a parcouru `liste0` en n étapes,
  - lorsque m est à la dernière valeur de `liste0`, `maxi` vaut le maximum de `liste0` par hypothèse de récurrence.

  À l'incrément suivant, m vaut a.
  Si `a>maxi`, alors `maxi` prend la valeur `a`, sinon `maxi` conserve la même valeur : celle du maximum de `liste0`.
  Or le maximum de `liste` est égal au maximum entre `a` et le maximum de `liste0`. C'est bien la valeur de `maxi`.
  Et la boucle se termine au bout de $n+1$ étapes.

- **Conclusion :** Par principe de récurrence, pour n'importe quelle longueur de liste, la fonction renvoie bien le maximum de la liste après un calcul ayant autant d'étapes que la longueur de la liste.

■

Remarque : Vous voyez que prouver un algorithme est assez laborieux, c'est la raison pour laquelle on ne le fait pas à chaque fois.

Complexité :

Ce raisonnement nous permet aussi de donner la **complexité** de l'algorithme : c'est-à-dire le nombre d'opérations à réaliser en fonction de la taille de la liste. Pour une complexité, on ne s'intéresse qu'à l'ordre de grandeur.

Ici, il faut n tests, on dira donc que l'on a une complexité en n : *la complexité est **linéaire** par rapport à la taille de la liste.*

Lorsque l'on conçoit un algorithme, le but est d'avoir la complexité la plus faible possible pour que l'algorithme soit rapide.

C Application aux chaînes de caractères

On veut tester si une chaîne de caractère `mot` est incluse dans une autre : `texte`.
Cela revient à programmer la commande **in** pour les chaînes de caractères. La différence avec les listes est qu'il ne suffit pas de tester caractère par caractère, mais avec une chaîne de plusieurs caractères.

```
def appartientStr(mot,texte):
    for i in range(len(texte)-len(mot)+1):
        if texte[i:i+len(mot)] == chaine:
            return True
    return False
```

Python VI.5: Recherche d'une sous-chaîne

On pourrait améliorer sensiblement cette programmation qui n'est pas du tout efficace dès que `mot` devient un peu long.

Exercice

Programmez une fonction indice et une fonction de comptage comme cela a été fait pour les listes.

2 Tri d'une liste

Nous allons programmer des algorithmes de tri *en place*. Le terme "*en place*", veut dire que l'on ne va pas créer de nouvelles listes, mais modifier la liste existante peu à peu jusqu'à ce qu'elle soit triée.

L'avantage d'une telle méthode est qu'elle est économe en espace mémoire (on n'a pas besoin de recopier la liste).

A Tri par sélection

C'est le tri le plus simple à programmer. Vous prenez le plus petit élément de la liste et vous le placez au début, puis le deuxième plus petit que vous mettez en deuxième position... jusqu'à ce que tout soit trié.
À l'étape p,

- la sous-liste des p premiers éléments est triée,
- tous les éléments de cette sous-liste sont inférieurs aux restants.

```
def triSelection(liste):
    # modifie en place
    # va chercher le plus petit et le met au début
    for p in range(len(liste)):
        indiceMin = p
        for i in range(p+1,len(liste)):
            if liste[i] < liste[indiceMin]:
                indiceMin = i
        (liste[p],liste[indiceMin]) = (liste[indiceMin],liste[p])
```

Python VI.6: Tri par sélection

Preuve :

par récurrence en montrant la double propriété : À l'étape p, la sous-liste des p premiers éléments est triée et tous ses éléments sont inférieurs à ceux restants.
Donc à l'étape n, toute la liste est triée.

■

Complexité :

Il faut n étapes, comme montré dans la preuve. À l'étape p on parcourt la liste restante de longueur $n-p$ pour trouver le minimum.
Le nombre d'opérations sera donc $\sum_{p=0}^{n}(n-p) = \sum_{p=0}^{n} p$ par inversion de l'ordre de sommation.
Donc la complexité est en $\frac{n(n+1)}{2}$, c'est-à-dire en n^2 pour l'ordre de grandeur.
On dit que le tri par sélection est de **complexité quadratique.**
Si je prend une liste deux fois plus longue, il me faudra 4 fois plus de temps pour la trier. Si je prends une liste 10 fois plus longue, il me faudra 100 fois plus de temps. Vous voyez que cela peut devenir problématique lorsque la taille de la liste à trier est très grande (des centaines de milliers, des millions... de données).

B Tri par insertion

Le tri par insertion est le tri naturel du jeu de carte. On parcourt la liste et dès que l'on trouve un élément qui ne suit pas l'ordre correct, on l'insère où il se doit dans la partie déjà triée.

Ainsi, comme dans le tri précédent, à l'étape p, la liste des p premiers éléments est triée. Par contre, la deuxième condition du tri par sélection n'est plus vérifiée. En effet, dans les éléments de la fin de la liste, il faudra en prendre pour les insérer au milieu de la liste déjà triée.
On peut programmer ce tri de plusieurs façon différentes. Voici trois propositions :

- **le naïf :** il est naïf, il est gentil et on l'aime bien. Il a compris ce qu'il fallait faire, mais on pourra pousser la réflexion un peu plus loin pour être plus efficace.
  Pour insérer un élément au milieu de la liste triée, on le fait remonter en échangeant les éléments 2 à 2 jusqu'à trouver la bonne place.

```
def triInsertionNaif(liste):
    # À chaque étape la sous-liste des p premiers éléments est triée.
    for p in range(1,len(liste)):
        # on remonte la liste
        i = p
        while i > 0 and liste[i-1] > liste[i]:
            # dans le mauvais ordre, on fait encore remonter.
            (liste[i],liste[i-1]) = (liste[i-1],liste[i])
            i -= 1
```

Python VI.7: Le tri par insertion : méthode naïve

- **le pragmatique :** La version naïve est bien, mais cela fait beaucoup d'affectations à chaque fois : dans la réalité, lorsque vous triez votre jeu de carte, vous insérez directement la carte au bon endroit en repoussant toutes les autres d'un cran. Vous n'avez pas besoin de l'insérer dans chaque position intermédiaire. Ici, on se contente donc de garder la carte p en main (on la nomme `element`) et on décale les autres à chaque étape, jusqu'à arriver à la bonne position.

```
def triInsertionPrag(liste):
    for p in range(1,len(liste)):
        #on remonte la liste
        element = liste[p]
        i = p
        while i > 0 and liste[i-1] > element:
            #élément se trouve avant, on repousse d'un cran
            liste[i] = liste[i-1]
            i -= 1
        #on insère entre i-1 et i, c'est-à-dire à la nouvelle position i
        liste[i] = element
```

Python VI.8: Le tri par insertion : méthode pragmatique

- **le persévérent :** autant suivre la logique du pragmatique jusqu'au bout. Plutôt que de décaler les cartes une par une, on les décale en bloc au moment de l'insertion de la carte p.

```
def triInsertionSlice(liste):
    for p in range(1,len(liste)):
        # on remonte la liste
        i = p
        while i > 0 and liste[i-1] > liste[p]:
            i -= 1
        # on insère entre i-1 et i, c'est-- dire à la nouvelle position i
        liste.insert(i,liste[p])
        liste.pop(p+1)          # attention, il a été décalé par l'insertion
```

Python VI.9: Le tri par insertion : méthode avec slicing

C Tri à bulles

On continue sur la même lancée. Cette fois-ci on fait remonter les éléments trop gros comme des bulles. Les bulles les plus grosses remontent le plus haut, jusqu'à rencontrer une bulle plus grosse qu'elle.
Concrètement, on fait des parcours de liste en agrippant les bulles une à une. Lorsque l'on fait un parcours sans faire remonter une seule bulle, c'est terminé.

Programmation : On avance dans la liste et on compare les éléments par couple :

- s'ils sont dans le bon ordre, on ne fait rien

- sinon, on les intervertit.

On fait autant de parcours de la liste que nécessaire.

Cet algorithme est un peu plus difficile, vous devez l'avoir compris et être capable d'expliquer, mais on ne peut pas exiger que vous sachiez le programmer en question de cours.

- **Première version :**

```
def triBulles(liste):
    chaos = True                                    #la liste n'est pas triée
    while chaos:
        chaos = False                              # on suppose la liste triée
        for p in range(len(liste)-1):
            if liste[p] > liste[p+1]:
                #on fait remonter la bulle
                (liste[p+1],liste[p]) = (liste[p],liste[p+1])
                chaos = True    # on n'est pas trié puisqu'on a bougé une bulle
```

Python VI.10: Le tri à bulles naïf

- **Version améliorée :**
  On s'aperçoit qu'à mesure que l'on réalise des parcours de liste, la fin de la liste est triée. Si la dernière bulle est remontée jusqu'à la position $p+1$ et que l'on a rien modifié derrière, c'est que toute la fin de la liste est triée.
  En plus, la dernière bulle que l'on a fait remonter est plus grosse que toutes celles qui précèdent (sinon, ce serait la bulle plus grosse qui aurait été traitée d'abord). Et cette bulle s'arrête en position $p+1$. Cela veut dire que la sous-liste à partir de $p+1$ est non seulement triée, mais que tous ses éléments sont plus grands que ceux qui précèdent.
  On n'a donc besoin de n'effectuer le tri à bulle que jusqu'à p.
  La condition d'arrêt du tri devient alors lorsque $p = 1$. (s'il ne reste que l'élément 0, il est forcément trié puisqu'il est seul).

```
def triBulles2(liste):
    pmax = len(liste)#la sous liste liste[pmax:] est triée
    while pmax > 1:
        last = 0
        for p in range(pmax-1):
            if liste[p] > liste[p+1]:
                #on fait remonter la bulle
                (liste[p+1],liste[p]) = (liste[p],liste[p+1])
                last = p+1 #où remonte la dernière bulle
        pmax = last
```

Python VI.11: Le tri à bulles amélioré

Complexité :
Avec la version améliorée, si la liste est déjà triée, alors, il n'y aura qu'un seul parcours de liste.
Par contre, si la liste est triée à l'envers, alors ce sera la situation la plus longue : chaque bulle devra être remontée.
À chaque étape, la longueur de la liste diminue de 1 et on fait un parcours complet.
La complexité est donc $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$: c'est une **complexité quadratique**.

D La complexité

Pour se rendre compte de l'intérêt des questions de complexité, les algorithmes précédents ont été utilisés sur une même liste. À chaque fois, la durée en seconde nécessaire au tri de la liste a été mesuré (grace à la fonction `clock()` de la bibliothèque `time`).
Pour obtenir un élément de comparaison, la liste a également été triée avec le tri par défaut de Python.
Voici les résultats obtenus pour une liste de 1000 éléments (de 0 à 999) rangés aléatoirement, puis pour une liste avec dix fois plus d'éléments.
Une étude expérimentale complète demanderait de faire des statistiques sur un grand nombre de listes, d'essayer avec des listes déjà triées, ou triées en sens inverse... Mais ce n'est pas le but ici.

```
#Pour 1000 éléments
Tri sélection, temps = 0.09130899999991016
Tri insertion naif, temps = 0.12908400000014808
Tri insertion pragmatique, temps = 0.08068600000001425
Tri insertion optimisé, temps = 0.04642300000000432
Tri insertion bulle, temps = 0.43186900000000605
Tri insertion bulle optimisé, temps = 0.23266100000000733
Tri de Python, temps = 0.0002649999998993735

#Pour 10000 éléments
Tri sélection, temps = 6.991663999999901
Tri insertion naif, temps = 12.147036000000071
Tri insertion pragmatique, temps = 7.282917999999881
Tri insertion optimisé, temps = 4.433541000000105
Tri insertion bulle, temps = 43.287115000000085
Tri insertion bulle optimisé, temps = 23.52394400000003
Tri de Python, temps = 0.0033519999999498395
```

Python VI.12: Temps d'execution des différents tri sur des listes aléatoires

On observe que les écarts de temps sont considérables ! Pour la liste à 10000 éléments (pas très gros), le tri à bulle naïf a mis 43 secondes alors que le tri de Python a mis moins d'un centième de seconde.
Vous voyez donc qu'un algorithme maladroit (même s'il est juste) devient rapidement inutilisable dans la pratique (à moins que vous ne soyez pas pressé...).

Il nous reste beaucoup de travail avant d'obtenir les mêmes performances que ceux qui ont programmé la méthode `sort()` en Python. Mais gardons surtout en mémoire qu'avec un tout petit peu de réflexion, on peut déjà gagner beaucoup de temps. Ainsi entre le tri à bulle naïf et celui optimisé, on a divisé le temps par 2.

On remarque aussi que lorsque l'on a multiplié la taille de la liste par 10, le temps a été globalement multiplié par $10^2 = 100$. CE qui permet de vérifier que la complexité de nos algorithmes est de type **quadratique**[22]. Imaginez donc l'écart de temps pour un tableau de taille 100000. Avec le tri à bulle naïf, on peut imaginer un temps d'environ $4300s \approx 1h10min$ alors que cela demande moins d'une seconde pour la méthode `sort()`.

3 Statistiques d'une liste de nombres

A Somme et produit

Sommer ou faire le produit de tous les éléments d'une liste de nombres

```
def somme(liste):
    s = 0
    for i in liste:
        s += i
    return s

def produit(liste):
    p = 1
    for i in liste:
        p *= i
    return p
```

Python VI.13: Somme des éléments d'une liste

Attention, une somme vide vaut 0, mais un produit vide vaut 1.

B Moyenne

Moyenne des éléments d'une liste (supposée non vide)

[22] En fait la méthode Python est en complexité $n \ln n$, mais on ne voit pas bien la différence ici. L'algorithme utilisé est le `Timsort`. C'est une version hybride du tri fusion (qui n'est pas au programme de première année) et du tri par insertion.

```python
def moyenne(liste):
    m = 0
    for i in liste:
        m += i
    return m/len(liste) #on suppose len(liste)!= 0
```

Python VI.14: Moyenne des éléments d'une liste

C Médiane

Médiane des éléments d'une liste (supposée triée). Faites des exemples sur papier pour voir la bonne formule.

```python
def mediane(liste):
    # si la liste n'est pas supposée triée, commencer par la trier
    lg = len(liste)
    if lg%2 == 0:                              #nombre pair d'éléments
        # moyenne des éléments centraux
        return (liste[lg//2-1]+liste[lg//2])/2
    # sinon renvoie l'élément central
    return liste[lg//2]

#Une autre méthode qui évite la disjonction des cas :
def mediane(liste0):
    lg = len(liste)
    return (liste[lg//2]+liste[(lg-1)//2])/2
```

Python VI.15: Médiane des éléments d'une liste

4 Recherche dichotomique

Ces algorithmes ne sont pas au programme pour Python, mais ils sont tellement souvent utilisés dans les sujets d'oraux que c'est "comme si".

L'idée est de couper à chaque fois en deux la zone de recherche.

A ⋆ Enjeux du problème : la complexité d'une recherche

Lorsque l'on cherche un élément dans une liste, on parcourt la liste jusqu'à ce qu'on l'ai trouvé.
Au pire des cas, il se trouve à la toute fin et on utilise autant d'opérations que la longueur de la liste : n. On dit que la **complexité au pire** est en $\Theta(n)$.
Mais la plupart du temps, on trouvera l'objet avant. On veut donc savoir, en moyenne, combien de temps il faudra pour trouver l'objet.
Bien sûr cela dépend de la probabilité qu'il a de se trouver à l'une ou l'autre position dans la liste. Ici, on supposera qu'on est sûr qu'il se trouve dans la liste et que la probabilité qu'il se trouve à l'une ou l'autre position dans a liste est la même (répartition équiprobable).
Si on veut connaître la **complexité en moyenne**, on fait donc la moyenne des complexités en fonction de la position de l'objet dans la liste pondérée par la probabilité qu'il s'y trouve.
On note n la longueur de la liste.
Pour chaque indice $k \in [\![0, n-1]\!]$, on note

- $t(k)$ le temps pour trouver l'élément s'il se trouve en position k,
- $P(k)$ la probabilité que l'élément se trouve en position k.

Alors, la complexité en moyenne est donnée par

$$C = \sum_{k=0}^{n-1} t(k)P(k)$$

Or, $\forall k \in [\![0, n-1]\!]$, $P(k) = \frac{1}{n}$ (situation équiprobable)
et $\forall k \in [\![0, n-1]\!]$, $t(k) = k+1$ (temps[23] pour aller à l'indice k).
Ainsi

$$C = \sum_{k=0}^{n-1} t(k)P(k) = \frac{1}{n}\sum_{k=0}^{n-1}(k+1) = \frac{1}{n}\sum_{k=1}^{n} k = \frac{n+1}{2} \approx \frac{n}{2} = \Theta(n)$$

En moyenne, on trouve l'objet avec $\frac{n}{2}$ itération. Cela reste une complexité linéaire comme pour la complexité au pire (si on multiplie la longueur de la liste par 10, alors le temps pour trouver l'objet est aussi multiplié par 10 en moyenne).

En général on ne peut pas faire mieux, car tant que la liste n'a pas été entièrement parcourue, nous n'avons aucune information sur les éléments restants.
Par contre, lorsque la liste est triée, la lecture de chaque élément nous donne des informations sur le reste de la liste. On peut donc utiliser cette information pour réduire le temps de recherche dans la liste.

Nous allons mettre en œuvre cette idée dans deux situations

B Recherche dans une liste triée

La recherche **dichotomique** est la recherche naturelle dans une liste triée. Lorsque l'on vous demande de trouver un nombre de 1 à 100 et qu'à chaque essai on vous indique s'il est plus grand ou plus petit, alors la procédure naturelle est de couper l'intervalle en deux à chaque fois : on commence par 50, puis si c'est trop grand, on essaie avec 25…
C'est ce que l'on vous demande de programmer ici.

Principe de l'algorithme :
Pour restreindre l'intervalle de recherche (on divise sa taille par deux environ à chaque itération), on définit les bornes entre lesquelles on cherche l'élément : `mini` (inclus) et `maxi` (exclu[24]).
Au début `mini` vaut le premier indice : 0 et `maxi` vaut le dernier indice +1 : `len(liste)`.
À chaque étape, on prend l'objet du milieu (on note `i` le milieu).

- S'il est égal à `a`, c'est terminé et on renvoie `i`,
- S'il est plus petit, on cherche alors dans [`i+1`,`maxi`[,
- S'il est plus grand, on cherche alors dans [`mini`,`i`[.

Et on continue tant que l'intervalle contient plusieurs éléments. Lorsque l'intervalle est vide, si rien n'a été trouvé, alors c'est que l'objet `a` n'est pas dans la liste et on renvoie une erreur.

Programmation effective :
Dans cette fonction, on restreint peu à peu l'intervalle de recherche . L'intervalle est défini par les bornes.
À chaque itération :

- On prend le milieu de l'intervalle : `i=(maxi+mini)//2`. Par exemple,
  - pour $[1,2,3,4,5,6,7]$, le milieu est 4, et c'est l'élément que l'on prend avec la division entière

$$(\texttt{maxi}+\texttt{mini})//2 = \left\lfloor \frac{\texttt{maxi}+\texttt{mini}}{2} \right\rfloor = \left\lfloor \frac{8+1}{2} \right\rfloor = 4$$

  - pour $[1,2,3,4,5,6,7,8]$ le milieu est à 4,5, on choisit l'élément 5.

$$(\texttt{maxi}+\texttt{mini})//2 = \left\lfloor \frac{\texttt{maxi}+\texttt{mini}}{2} \right\rfloor = \left\lfloor \frac{9+1}{2} \right\rfloor = 5$$

- On compare `a` avec l'élément d'indice `i` :
  - si `a` est égal à l'élément d'indice `i`, c'est terminé : on renvoie `i`,
  - si `a` est strictement supérieur à l'élément d'indice `i`, alors on cherche dans l'intervalle *au delà* de `i`,

[23] En fait, comme on raisonne en ordre de grandeur, le $k+1$ pourrait être remplacé par k.
[24] On conserve ainsi la logique de Python : borne inférieur incluse et borne supérieure exclue. Mais ce n'est pas obligatoire.

– sinon, a est inférieur et on cherche en deça.

On fait les itérations jusqu'à ce que l'on ai trouvé l'élément ou que l'intervalle soit vide (`mini=maxi`).

```
# on suppose la liste de nombres classée par ordre croissant.
def dichotomie(a,liste):
    #intervalle de recherche : [mini,maxi[
    mini = 0
    maxi = len(liste)
    while maxi>mini:
        i = (maxi+mini)//2
        if liste[i] == a:
            return i
        elif liste[i] < a:
            mini = i+1
        else:
            maxi=i #intervalle ouvert à droite
    raise ValueError(str(a)+" n'est pas dans la liste")
```

Python VI.16: Recherche dichotomique dans une liste classée par ordre croissant

⋆ **Complexité de la recherche dichotomique :**La complexité au mieux est bien sûr égale à 1 : on tombe sur la bonne valeur au premier essai.
Intéressons-nous à la complexité au pire.
À chaque étape, l'intervalle de recherche est divisé par 2. Au bout de k étapes, la taille de l'intervalle est donc divisée par 2^k.
Pour connaître le nombre d'étapes maximum avant de trouver le bon nombre, il faut chercher le plus petit $k \in \mathbb{N}$, tel que $2^k \geq n$.
On prend donc $k = \log_2(n) = \frac{\ln n}{\ln 2}$
La complexité est en $\Theta(\ln n)$.
C'est une complexité logarithmique. C'est beaucoup mieux qu'une complexité linéaire. Pour s'en convaincre, il suffit de comparer les courbes de $x \mapsto x$ et $x \mapsto \ln x$, lorsque x devient très grand.

C Recherche d'un zéro d'une fonction continue

Remarque : On peut démontrer le théorème des valeurs intermédiaires en utilisant cette démarche.

Le principe est le même. Soit f une application **continue** sur $[a,b]$ avec $f(a)\cdot f(b) < 0$ ($f(a)$ et $f(b)$ de signes contraires et non nuls)

Alors d'après le théorème des valeurs intermédiaires, $\exists c \in]a,b[$ tel que $f(c) = 0$.

Le but est d'obtenir c à ε près.

Pour cela, on découpe à chaque fois l'intervalle en 2 en posant $x = \frac{a+b}{2}$:

- si $f(x) > 0$, c'est que la fonction s'annule après x et on recommence sur l'intervalle $[x,b]$,
- si $f(x) < 0$, c'est que la fonction s'annule avant x et on recommence sur l'intervalle $[a,x]$.

On s'arrête lorsque la taille de l'intervalle est inférieure à $\varepsilon > 0$. On renvoie alors le milieu de l'intervalle pour minimiser l'erreur.

```
def dichotomie2(f,a,b,epsilon):
    while abs(b-a) > epsilon:
        x = (a+b)/2
        if f(c) == 0:
            return c
        elif f(a)f(c) < 0:
            b = c
        else:
            a = c
    return (a+b)/2

# essai pour obtenir une valeur approchée de racine carrée de 2
def test(x):
```

```
14      return -(x**2-2)
15
16  dichotomie(test,1,2,1e-5)
17  # renvoie : 1.4142112731933594
18
19  test(dichotomie(test,1,2,1e-5))
20  # renvoie : 6.474772817455232e-06
```

Python VI.17: Recherche dichotomique du zéro d'une fonction continue

revenir au sommaire

VII - MODÉLISATION D'EXPÉRIENCES ALÉATOIRES

Nous proposons ici de modéliser des expériences aléatoires simples avec des listes et le module `random`. Ce type de modélisation se trouve au cœur des oraux de mathématiques de type agro-veto (même si les oraux ne s'y résument pas !)

Dans le chapitre sur `numpy`, nous diposerons d'autres outils pour le faire.

Avertissement : il existe de multiples façons de modéliser les expériences. Nous proposons souvent plusieurs méthodes possibles. Le but n'est pas de faire un catalogue, mais de s'approprier celles qui nous parlent le plus.

1 Tirages dans une urne modélisée par une liste

A Modélisation d'une urne

Un moyen simple de modéliser une urne est d'utiliser une liste que l'on remplit comme l'urne. Par exemple, pour une urne qui contient p boules blanches et q boules noires, on peut créer une liste avec p chiffres 1, et q chiffres 0.

On adaptera sa modélisation en fonction du contexte. Si on veut compter le nombre de boules blanches pour différents tirages, ce seront elles que l'on codera par 1. Si, au contraire, on compte plutôt les boules noires, alors on échangera le codage. Le plus important ici est de *commenter* son code pour expliquer à l'utilisateur la convention choisie.

```
def modelUrne(p,q):
    urne = p*[1]+q*[0]
    return urne
```

Bien sûr, on peut généraliser à davantage de couleurs ou de valeurs.

B Tirage avec remise

Méthode de base : Pour tirer un objet avec remise, il suffit de le désigner dans l'urne (sans y toucher). On pourra utiliser pour cela la commande **choice**.
On peut ensuite réiterer l'expérience aléatoire de nombreuses fois de façon indépendantes. Dans certains cas, on voudra conserver la liste des résultats successifs, ou simplement le nombre de succès au cours des n tirages (par exemple le nombre de fois que l'on tire le chiffre 1).

```
# tirage avec remise
def tirage(urne):
    return choice(urne)

# tirages successifs avec remise.
# on enregistre les résultats dans une liste.
def tirageSuccessif(urne, nb):
```

```
    res = []
    for i in range(nb):
        res.append(choice(urne))
    return res

# tirages successifs avec remise.
# on compte le nombre de fois que l'on a obtenu la valeur k.
def tirageNbSucces(urne, k, nb):
    succes = 0
    for i in range(nb):
        succes += (choice(urne) == k)
    return succes
```

Méthode alternative : On peut également modéliser ce tirage en choisissant une position aléatoire dans l'urne. On utilise alors la commande **randrange**(`i,j,k`) qui tire un nombre aléatoire dans l'intervalle donné par **range**(`i,j,k`). Comme pour la commande **range**, on peut diminuer le nombre d'arguments.

```
# tirage avec remise
# utilisation de randrange
def tirage(urne):
    indice = randrange(len(urne))
    return (urne[indice])

# tirages successifs avec remise.
# on enregistre les résultats dans une liste.
def tirageSuccessif(urne, nb):
    res = []
    for i in range(nb):
        res.append(tirage(urne))
    return res

# tirages successifs avec remise.
# on compte le nombre de fois que l'on a obtenu la valeur k.
def tirageNbSucces(urne, k, nb):
    succes = 0
    for i in range(nb):
        succes += (tirage(urne) == k)
    return succes
```

C Tirage sans remise :

Cette fois-ci, il faut modifier l'urne à chaque tirage. Pour cela nous proposons de fonctionner avec effets de bords. Cela signifie que la liste (variable supposée globale) qui représente l'urne est modifiée à chaque tirage.

Attention :
Si on veut effectuer plusieurs fois l'expérience de façon indépendante, alors il faudra recréer une nouvelle urne à chaque fois.
(ou travailler avec des copies d'une urne *originale*, en veillant à ce que les copies ne pointent pas toutes vers le même objet mémoire)

Méthode de base : La méthode la plus simple est sans doute de mélanger l'urne avec **shuffle** puis de tirer le dernier élément.

Lorsque l'on veut réaliser plusieurs tirages, il suffit d'un seul `shuffle`, puis on tire les éléments les uns à la suite des autres avec `pop()`.

```
# tirage sans remise
def tirage(urne):
    shuffle(urne)
    return (urne.pop())

```

```
# n tirages sans remise
# mélange l'urne, mais ne la modifie pas.
def tirageSuccessif(urne, nb):
    shuffle(urne)
    return (urne[:nb])

# tirage sans remise,
# on compte le nombre de fois que l'on obtient la valeur k au cours de nb tirages
# l'urne est modifiée à la fin
def tirageNbSucces(urne, nb):
    succes = 0
    shuffle(urne)
    for i in range(nb):
        succes += (urne.pop() == 1)
    return succes

# pour ne pas modifier l'urne initiale
def tirageNbSucces(urne, nb):
    succes = 0
    urneLocale = urne[:]
    shuffle(urneLocale)
    for i in range(nb):
        succes += (urneLocale.pop() == 1)
    return succes
```

Remarque : Ici, dans l'algorithme du tirage successif, on prend les premiers éléments de la liste (sans les enlever), alors que dans les autres fonctions, on prend les derniers éléments (et on les enlève avec `pop()`). Cela n'a pas d'influence sur l'aléatoire, mais nous avons simplement écrit le plus facile à programmer dans chaque situation.

Lorsqu'il s'agit d'un **tirage sans remise exhaustif** : on tire tous les éléments de l'urne. Cela revient simplement à choisir un ordre aléatoire pour notre liste (c'est une permutation).

Il suffit alors de la seule commande

```
shuffle(urne)
```

Méthode alternative : Comme pour le tirage avec remise, on peut aussi pointer directement vers un indice avec **randrange** et le supprimer avec **pop**. Dans les algorithmes suivants, l'urne est modifiée à chaque fois. Comme précédemment, on peut empêcher ce résultat en créant une urne locale.

```
def tirage(urne):
    indice = randrange(len(urne))
    return(urne.pop(indice))

# tirages successifs sans remise.
# on enregistre les résultats dans une liste.
def tirageSuccessif(urne, nb):
    res = []
    for i in range(nb):
        res.append(tirage(urne))
    return res

# tirages successifs sans remise.
# on compte le nombre de fois que l'on a obtenu la valeur k.
def tirageNbSucces(urne, k, nb):
    succes = 0
    for i in range(nb):
        succes += (tirage(urne) == k)
    return succes
```

2 Approximation des probabilités et de l'espérance

A Approximation numérique de la probabilité

On admet que lorsque l'on réalise une expérience de nombreuses fois et de façon indépendante, la proportion de chaque événement tend vers sa probabilité. Dans le cas d'une variable aléatoire réelle, la moyenne des résultats tend vers l'espérance.

Ainsi, il suffit de calculer la proportion ou la moyenne obtenues lors de la répétition de l'expérience pour avoir une valeur approchée de la probabilité et de l'espérance. Ceci est très simple à mettre en œuvre.

Par exemple, on dispose d'une urne qui contient p boules blanches et q boules noires. On note X le nombre de boules blanches obtenus lors de `nb` tirages avec remise. On cherche une approximation numérique de $p(X = k)$.

```
urne = p*[1]+q*[0]

# nombre de boules blanches pour nb tirages avec remise dans l'urne
def tirage(urne, nb):
    res = 0
    for i in range(nb):
        res += choice(urne)           # 1 si blanc, 0 si noir
    return res

#p(X=k)
def proba(urne, nb, k, n = 1000):
    res = 0
    for i in range(n):
        res += (tirage(urne,nb) == k) # on ajoute 1 si le tirage donne k
    return res/n                      # proportion
```

Mais 1000 répétitions de l'expérience sont-elles suffisantes. Faut-il n'en faire que 50, ou au contraire 10 000 ?
Pour établir une conjecture, on peut appliquer cette fonction pour différentes valeurs de n, et lorsqu'il semble que cela converge, on s'arrête. Mais il y a beaucoup mieux en utilisant les outils graphiques.

B Visualisation graphique de la convergence

On peut calculer successivement les proportions en fonction du nombre de fois que l'on réalise l'expérience et les afficher sur un graphique. Ainsi, on verra graphiquement lorsque la courbe semble proche de sa limite.

```
#p(X=k)
def proba(urne, nb, k, n = 1000):
    res = 0
    for i in range(n):
        res += (tirage(urne,nb) == k) # on ajoute 1 si le tirage donne k
        plot(i+1,res/(i+1),'r,')
    return res/n                      # proportion
```

Remarques importantes :

- Lorsque l'on trace les valeurs successives, il est souvent préférable de tracer point par point, plutôt que d'enregistrer toute la liste. Cela évite d'utiliser trop de place en mémoire inutilement.
- Pour chaque point, il faut diviser par $i+1$. En effet, au premier "tour", on est à $i = 0$: il y a donc un décalage de 1, entre le nombre d'expériences que l'on a réalisées et la valeur du compteur i. C'est cohérent avec la valeur retournée en fin d'algorithme, car à la fin de la boucle, $i = n-1$.
- Ne pas oublier de renvoyer une valeur numérique avec `return` à la fin. La visualisation graphique n'est pas toujours suffisante (et n'est pas réutilisable par d'autres programmes).

C Estimation de l'espérance

On fait de même pour l'espérance, avec le calcul de la moyenne :

```
def esperance(urne, nb, n = 1000):
    res = 0
    for i in range(n):
        res += tirage(urne,nb)
        plot(i+1,res/(i+1),'r,')
    return res/n                        # moyenne
```

D Cas des tirages sans remise

Comme nous l'avons spécifié plus haut, lorsque le tirage est sans remise, il faut veiller à "remettre l'urne à zéro" entre chaque expérience. Par exemple, on peut utiliser des urnes "locales" pour cela (par contre, il faut faire un nouveau **shuffle** pour chaque expérience).

```
    urne = p*[1]+ q*[0]

# nombre de boules blanches pour nb tirages avec remise dans l'urne
# nb < p + q
def tirage(urne, nb):
    urneLocale = urne[:]
    shuffle(urneLocale)
    res = 0
    for i in range(nb):
        res += urneLocale.pop()           # 1 si blanc, 0 si noir
    return res

#p(X=k)
def proba(urne, nb, k, n = 1000):
    res = 0
    for i in range(n):
        res += (tirage(urne,nb) == k) # on ajoute 1 si le tirage donne k
        plot(i+1,res/(i+1),'r,')
    return res/n                        # proportion

def esperance(urne, nb, n = 1000):
    res = 0
    for i in range(n):
        res += tirage(urne,nb)
        plot(i+1,res/(i+1),'r,')
    return res/n                        # moyenne
```

3 Modéliser une loi de Bernoulli sans urne

Exemple type : on lance n fois de suite une pièce qui a la probabilité p de tomber sur pile et $1-p$ de tomber sur face.

On veut compter le nombre de fois que la pièce est tombée sur pile.

Programmation :

On tire un nombre aléatoire $x \in [0,1[$: si $x < p$, on renvoie 1 (pile), sinon, on renvoie 0 (face) :

```
def bernoulli(p):
    if random() < p:
        return 1
    return 0

def binomiale(p,n):
    res = 0
    for i in range(n):
        res += (random()<p)
    return res
```

4 Simuler une loi discrète non uniforme

On utilise pour cela la même idée que pour la loi de Bernoulli sans urne. Prenons un exemple :

On dispose d'un dé pipé dont la probabilité d'obtenir 6 est $\frac{1}{2}$, et la probabilité d'obtenir chacune des autres valeurs est $\frac{1}{10}$.

On modélise alors l'expérience aléatoire ainsi :

```
valeurs = [1,2,3,4,5,6]                    # valeur de la variables
p = 5*[1/10]+[1/2]          # probabilités associées

def tirage(valeurs, p):
    essai = random()
    f = 0                                  # fonction de répartition
    for i in range(len(p)):
        f += p[i]
        if essai < f:                      # p[i-1] < essai < p[i]
            return valeurs[i]
    return valeurs[-1]                     # inutile en théorie (en cas d'erreur d'
        approximation)
```

Ce programme se comprend mieux avec la fonction de répatition :

k	1	2	3	4	5	6
$p(X=k)$	0,1	0,1	0,1	0,1	0,1	0,5
$p(X\leq k)$	0,1	0,2	0,3	0,4	0,5	1

Par exemple, si `essai=0.25`, alors cela revient à obtenir un 3.

Pour tester dans quel intervalle de la fonction de répartition se trouve la variable `essai`, on la compare avec les valeurs successives de la fonction de répartition que l'on calculer tour à tour.

5 Évaluer des probabilités conditionnelles

Prenons à nouveau un exemple pour montrer comment on peut évaluer des probabilité conditionnelles avec Python.

On rappelle qu'évaluer des probabilités conditionnelles, revient simplement à changer de loi de probabilité. Il sera donc logique que la fonction ressemble à l'évaluation d'une probabilité *simple*.

On dispose de deux dés : un dé normal et un dé truqué (comme dans la section précédente). On tire un dé au hasard et on obtient un six. Quelle est la probabilité que le dé soit truqué.

On cherche donc la probabilité d'avoir un dé truqué, sachant que l'on a obtenu un six.

La modélisation est très simple. Comme nous l'avons vu en mathématiques, cela revient à restreindre notre univers. Ainsi, sur tous les essais, on ne retiendra que ceux qui donnent un six. Et parmi ceux-ci, on compte ceux pour lesquels le dé était truqué.

```

def tirage(listep):
    essai = random()
    for i in range(len(listep)):
        if essai < listep[i]:
            return i+1
        essai -= listep[i]
    return len(listp)

def probaCond(nb = 1000):
    listeT = 5*[1/10]+[1/2]
    listeN = 6*[1/6]
    n = 0                                  # nombre de tirages avec un six
    res = 0                                # nb dés truqués sachant six
    while n<nb:
        de = random()
        if de < .5:
            lancer = tirage(listeT)
        else:
```

```
            lancer = tirage(listeN)
        if lancer == 6:                          # sachant six
            n += 1
            if de < .5:                          # truqué sachant 6
                res += 1
            plot(n, res/n, 'r,')
    return res/n
```

revenir au sommaire

VIII - LE MODULE NUMPY

Il n'est pas question de connaître par cœur toutes les commandes présentées ici. Vous devez surtout retenir le fonctionnement de `numpy` et connaître quelques commandes essentielles. Pour les autres, soit vous les retrouverez dans ce cours ou dans toute autre documentation (utilisez l'autocomplétion de Pyzo), soit vous les programmerez à nouveau.

La documentation officielle est disponible sur le site https://docs.scipy.org/doc/numpy/ et elle est extrêmement claire !

1 De la liste au tableau

A Chargement du module

Nous chargerons l'ensemble du module numpy avec la commande

```
1 import numpy as np
```

Python VIII.1: Numpy : Import du module

Tous les noms de fonction de ce module seront préfixées par "`np.`". Cela permet de distinguer les fonctions qui viennent de ce module par rapport aux autres et d'éviter les risques liés aux homonymies[25].

Pas de préfixe pour les méthodes : On remarquera que le préfixe "`np.`" n'est pas utile lorsqu'il s'agit d'une méthode que l'on applique à un objet numpy. En effet, une méthode est une fonction qui se trouve programmée *dans* l'objet lui-même et Python ne la cherche donc pas dans une bibliothèque, mais directement dans l'objet qui est de type tableau.

B Spécificité des tableaux

Un tableau ressemble beaucoup à une liste, mais en plus *rigide* :

- Un tableau a une taille fixée lors de sa création.
  On ne peut donc pas lui rajouter ou enlever des éléments comme avec une liste. Ici, pas de méthode `append`, `pop`...
- Tous les éléments d'un tableau doivent être du même type[26] (lui aussi fixé à la création).

Ces contraintes permettent d'optimiser la gestion mémoire et les opérations sur les tableaux. Ainsi, à la création, la taille et le type des données du tableau permettent à Python de savoir exactement quelle espace mémoire prendra le tableau et cela ne variera plus.

[25] Deux fonctions issues de deux bibliothèques différentes peuvent porter le même nom et avoir un comportement différent. C'est le cas de `randint(a,b)` selon qu'il est issu de `random` ou de `math`. L'utilisation d'un alias lors du chargement des bibliothèques permet alors de distinguer ces deux fonctions homonymes.

[26] Il existe une méthode pour contourner cette contrainte, mais ce n'est pas l'*objet* ici.

Les accès aux éléments du tableau, les opérations pourront être optimisés et plus rapides qu'avec des listes.

C Création de tableaux à partir de listes

On peut créer un tableau à partir d'une liste. Ce tableau peut avoir une ou plusieurs dimensions (par exemple pour faire des matrices de taille $n \times p$).

```
liste = [1,2,3,4,5]
tableau = np.array(liste)                          # crée un tableau à partir de la liste
type(tableau)                                      # numpy.ndarray
tableau.dtype                                      # int64 : type des éléments du tableau
matrice = np.array([[ 1,0,0],[0,1,0]])             # matrice de taille 2x3
tab = np.array([1,'a',2])                          # erreur : des types différents
```

Python VIII.2: Numpy : Création d'un tableau

La commande `arange` fonctionne comme la commande `range` mais pour créer un tableau. On en profite pour retrouver une autre commande que l'on connaît déjà `linspace`.

```
tableau = np.array(range(100))
tableau = np.arange(100)              # même résultat
tableau = np.arange(-5,6)
tableau = np.arange(-5,6.4,.2)        # la commande accepte des arguments non entier
tableau = np.linspace(-5,6,100)       # le troisième argument est le nombre de points.
```

Python VIII.3: Numpy : Commandes arange et linspace

Réciproquement, on peut transformer un tableau en liste avec la méthode `tolist()` (cela crée un nouvel objet).

```
matrice.tolist()              # crée une liste à partir du tableau
type(matrice)                 # renvoie numpy.ndarray car matrice n'est pas modifiée
matrice = matrice.tolist()    # remplace le tableau par une liste.
matrice = array(matrice)      # revient au tableau
```

Python VIII.4: Numpy : passage des tableaux aux listes

Exercice

Quelle est la différence entre la fonction `list` et la méthode `tolist()` ?

D Attributs d'un tableau

```
matrice.dtype                 # type des éléments du tableau
matrice.shape                 # dimension sous la forme d'un tuple
matrice.shape[0]              # nombre de lignes d'une matrice
matrice.shape[1]              # nombre de colonnes d'une matrice
matrice.size                  # nombre total de coefficients
matrice.ndim                  # nombre de dimensions du tableau (2 pour matrice)
```

Python VIII.5: Numpy : Attributs d'un tableau

E Lecture-écriture

Le parcours des tableaux est très proche de celui des listes avec le slicing.

Par contre, lorsqu'il y a plusieurs coordonnées, on accède à l'élément en mettant un `tuple` entre les crochets : on sépare les coordonnées par des virgules au lieu de les mettre dans des crochets successifs comme nous le faisions avec les listes de listes.

```
tableau = np.arange(10)
matrice = np.array([[10*j+i for i in range(10)] for j in range(8)])

```

```
tableau[4]              # élément d'indice 4 dans le tableau
matrice[4]              # tableau de la ligne d'indice 4
matrice[4,:]            # idem
matrice[4,3]            # élément d'indices 4,3
matrice[5:,5:]          # bloc inférieur droit
matrice[::-1]           # inverse l'ordre des lignes
```

Python VIII.6: Numpy : slicing et accès aux coefficients

```
# cas des listes de listes
lst =  [[10*j+i for i in range(10)] for j in range(8)]
lst[5][2]               # dans la liste lst[5], je demande l'élément 2

# cas des tableaux à deux dimensions
matrice = np.array(lst)
matrice[5,2]            # je demande le coefficient de coordonnées 5,2 du tableau.
matrice[5][2]           # Numpy est gentil : fonctionne aussi.
```

Python VIII.7: Numpy : différence d'accès listes - tableaux

```
tableau[4] = 20         # modifie l'élément d'indice 4
tableau[4] = 'a'        # erreur, pas le bon type
tableau[4] = 19.5       # remplace par int(19.5)=19
matrice[5,7] = 12
matrice[:5,:5] = 0      # remplace le bloc 5x5 en haut à gauche par des 0
matrice[:2,1] = [-1,-2] # remplace les deux premières coordonnées de la colonne 1
matrice[:2,1] = [1,2,3] # erreur, mauvaise dimension
```

Python VIII.8: Numpy : écriture dans un tableau

F Difficultés avec les pointeurs mémoire

Il faut faire attention aux effets de gestion de la mémoire dans Python.

Contrairement aux listes, lorsque vous utilisez le *slicing*, Python ne recopie pas la partie du tableau concernée, mais fait pointer une nouvelle variable vers un extrait du même tableau.

Si vous modifiez le contenu de cette nouvelle variable, cela modifiera aussi le tableau initial.

La méthode `copy()` permet de faire une copie complète du tableau.

```
# ****  avec les listes  ****
liste = [0,1,2,3,4]
liste2 = liste
liste2 is liste               # True : même objet mémoire

liste3 = liste[:]
liste3 is liste               # False : nouvel objet mémoire

liste4 = liste.copy()
liste4 is liste               # False : nouvel objet mémoire

# **** avec les tableaux ****
tab = np.array([0,1,2,3,4])
tab2 = tab
tab2 is tab                   # True : même objet mémoire

tab3 = tab[:]
tab3 is tab                   # /!\ True : même objet mémoire

tab4 = tab.copy()
tab4 is tab                   # False : nouvel objet mémoire

# ****      exemples    ****
matrice = np.array([[10*j+i for i in range 10] for j in range 7])
```

```
25
26 extrait = matrice[:5,:5]      # bloc 5x5 en haut à gauche de matrice
27 extrait[:,:] = 0              # remplit extrait avec des zéros
28 print(matrice)                # le bloc est aussi modifié dans matrice
```

Python VIII.9: Numpy : Difficultés avec les pointeurs mémoire

G Extraction d'un sous-tableau

Lorsque les cellules que l'on veut extraire sont contiguës (ou de 2 en 2...) le slicing permet d'extraire facilement un sous-tableau de celui existant.

On peut vouloir extraire de façon plus manuelle et créer un tableaux à partir des coefficients d'un autre.

Pour cela on utilise le *fancy indexing* : si on veut un tableau qui contienne certains coefficients extraits d'un autre tableau, on crée le tableau avec les coordonnées des dits coefficients. C'est plus simple à comprendre sur un exemple :

```
1 tableau = np.arange(0,40,4)          # tableau initial
2 indices = np.array([2,4,6],[1,3,5]) # indices à extraire
3 tableau[indices]                     # tableau de taille 2x3 avec les coefficients
      indiqués par indices.
```

Python VIII.10: Numpy : fancy indexing

H Parcours itératif d'un tableau

Un tableau est un objet *itérable*, il peut être parcouru avec une boucle **for** par exemple.

Par contre, seule sa première dimension est parcourue. Ainsi, pour une matrice à deux dimensions, la variable parcourra les lignes (qui sont des tableaux).

```
1  tableau = np.arange(0,40,4)          # tableau initial
2  for i in tableau:
3    print(i)
4
5  matrice = [[5*j+i for i in range(5)] for j in range(3)]
6  for i in matrice:
7    print(i)                           # affiche les lignes de matrice
8
9  # que fait cette procédure ?
10 for i in matrice:
11   for j in i:
12     print(j)
```

Python VIII.11: Numpy : Parcours d'un tableau

Pour parcourir l'intégralité d'un tableau, on peut *l'aplatir* avec la méthode `.flat`. Cela revient à mettre les lignes bout-à-bout.

Petites subtilités : la méthode `.flat` se contente de créer *un parcours*[27] de tableau (elle ne créé donc pas un nouveau tableau, mais indique seulement comment parcourir celui existant).

Il existe deux autres méthodes qui ressemblent. La méthode `.flatten()` crée qui crée un nouveau tableau applati et la méthode `.ravel()` crée *une vue* applatie du tableau.

Ainsi, avec la méthode `.flatten()`, on créé un nouvel objet mémoire qui existe indépendemment du tableau initial.

Avec la méthode `.ravel()`, on pointe vers le même objet mémoire, mais présenté sous forme applatie. Cette méthode va donc plus vite, mais si vous modifiez le tableau obtenu avec cette méthode, vous modifierez aussi le tableau *source*.

La méthode `.flat` ne sert que pour un parcours (avec une boucle `for` par exemple. L'objet renvoyé n'est pas un tableau, mais un itérateur.

Pour changer la forme d'un tableau, voir les méthodes `reshape` (nouvelle vue) et `resize` (nouveau tableau).

[27] On appelle cela un itérateur en Python.

```
matrice = np.array([[5*j+i for i in range(5)] for j in range(3)])
for i in matrice.flat:
  print(i)

sum(x for x in matrice)           # renvoie un tableau avec les sommes des colonnes
sum(x for x in matrice.flat)      # somme de toute la matrice.
```

Python VIII.12: Numpy : aplatissement d'un tableau

2 Création des tableaux usuels

En général, il est suffisant de construire des tableaux à partir de listes en utilisant les listes par compréhension autant que possible. Il existe néanmoins des commandes pour créer les tableaux usuels. En voici quelques unes.

Pour chacune des commandes qui suivent, vous devez être capable de générer le même tableau avec une liste.

On commencera pas se rappeler les commandes `arange` et `linspace`, déjà vues plus haut dans le programme VIII.3.

A Les tableaux nuls

Ils sont construits par la fonction `zeros(`*`taille`*`)` où *taille* est un tuple qui contient les dimensions.

Par défaut, les éléments du tableau sont de type `float` (modifiable avec un argument optionnel).

```
np.zeros(10)                  # tableau à une dimension de taille 10 rempli de 0
np.zeros((3,5))               # taille 3x5 ne pas oublier les doubles parenthèses.
np.zeros((3,5),int)           # de type entier.
```

Python VIII.13: Numpy : command zeros

B Les tableaux unitaires

Ils ne comportent que des 1 et sont construits par la fonction `ones(`*`taille`*`)`. La syntaxe est la même que la fonction `zeros`.

```
np.ones(10)                   # tableau à une dimension de taille 10 rempli de 1
np.ones((3,5))                # taille 3x5 rempli de 1
5*np.ones((3,5))              # taille 3x5 rempli de 5
np.ones((3,5),int)            # de type entier.
```

Python VIII.14: Numpy : command ones

C La matrice identité

Elle se construit avec la commande `eye(`*`taille`*`)`.

Avec des arguments supplémentaires, elle permet de décaler la diagonale de 1 ou de réaliser des matrices rectangulaires.

```
np.eye(4)                     # identité de taille 4x4
np.eye(4,dtype=int)           # identité de taille 4x4, de type int
np.eye(4,k=1)                 # surdiagonale avec des 1
np.eye(4,k=-1)                # sousdiagonale avec des 1
```

Python VIII.15: Numpy : command eye

D Les matrices diagonales

Elles sont définies par leur diagonale.

```
np.diag([1,1,1])                # identité de taille 3x3
np.diag([1,2,3,4])              # taille 4x4 avec 1,2,3,4 en diagonale
np.diag([1,2,3],k=1)            # taille 4x4, surdiagonale avec 1, 2, 3
np.diag([1,2,3],k=-1)           # sousdiagonale
```

Python VIII.16: Numpy : command diag

E Les matrices triangulaires inférieures

Elles sont remplies de 1 sous la diagonale (au sens large).

```
np.tri(3)                       # triangulaire inférieure de taille 3x3
```

Python VIII.17: Numpy : command tri

F Éviter les listes par compréhension

Pour créer un tableau dont les valeurs dépendent des indices dans le tableau, on utilisait jusqu'à présent les listes par compréhension. On peut le faire directement avec la fonction `fromfunction`.

Cette méthode est meilleure car elle évite de créer une liste intermédiaire.

```
def f (i,j): return 5*i+j
matrice = np.fromfunction(f,(3,5))

# ou plus simplement
matrice = np.fromfunction(lambda i,j:5*i+j,(3,5))

#tableau à une dimension
matrice = np.fromfunction(lambda x:2*x-1,(10,))
```

Remarque : il ne faut pas de parenthèses autour du couple `i,j` dans la définition avec **lambda**.
Lorsque le tableau n'a qu'une seule dimension, il faut indiquer $(n,)$ pour l'argument shape.

G Tableaux aléatoires

De nombreuses fonctions de la bibliothèque `random` ont leur équivalent dans le *sous-module* `random` de `numpy`.

Malheureusement, il n'existe pas de `randrange` dans `numpy`.

```
matrice = np.random.rand(3,5)
matrice = np.random.randint(100,size=(3,5))
matrice = np.random.randint(100,size=5)
```

3 Opérations matricielles

A Opérations terme à terme

Par défaut, toutes les opérations entre tableau s'effectuent terme à terme. C'est le cas des opérations $+, \times \ldots$, mais aussi des comparaisons...

Ainsi, lorsque l'on effectue une opération usuelle entre une matrice et un scalaire, cette opération est appliquée à chacun des coefficients.

On peut également appliquer les fonctions usuelles sur les tableaux. Ces fonctions sont alors appliquées coefficient par coefficient (il faut ajouter le préfixe `np.` au nom de la fonction).

Attention : Le produit * entre matrice est un produit coefficient par coefficient, ce n'est pas le produit matriciel.

```
J2 = 2*np.ones((3,3))          # matrice 3x3 remplie de 2
id = np.eye(3)
tableau = np.arange(10)

2*J2                           # matrice 3x3 remplie de 4
id+J2                          # addition matricielle : terme à terme
id*J2                          # différent de J2, c'est la matrice 2*id

np.sqrt(tableau)               # racine carrée des coefficients
np.log(tableau+1)              #log népérien
np.sin(tableau)

tableau < 3                    # [True,True,True,False,False,...]
```

Python VIII.18: Numpy : Opérations usuelles

Pour modifier la matrice, on peut utiliser les commandes `+=`, `*=`...auxquelles nous sommes habitués. Par contre, le comportement est un peu différent de celui des listes. En particulier, la matrice n'est pas réaffectée, mais modifiée en place.

```
J2 = np.ones((3,3))          # matrice 3x3 remplie de 1
J2 *= 2                      # mutliplie les coefficients par 2 -> 2
J2 += 1                      # et ajoute un à chaque coefficients -> 3
```

Python VIII.19: Numpy : Opérations en place

B Vectorisation d'une fonction

Lorsque l'on définit une fonction, en général, on ne peut pas l'appliquer à un tableau numpy (terme à terme).

Pour pouvoir le faire, on *vectorise* cette fonction avec la commande `vectorize`.

```
from math import sqrt

tableau = np.arange(10)
def f(x):
  return (x-1)/(sqrt(x+1))

f(tableau)                    #erreur

vf = np.vectorize(f)
vf(tableau)
```

Python VIII.20: Numpy : Vectorisation

C Transposition

```
matrice = np.fromfunction(lambda i,j:5*i+j,(3,5))
matrice.transpose()
matrice.T
```

Python VIII.21: Numpy : Transposition d'une matrice

D Produit matriciel, puissances, inverse

```
matrice = np.fromfunction(lambda i,j:5*i+j,(3,5))
J = np.ones((3,3))
id = np.eye(3)
```

```
#Produit matriciel
np.dot(J,matrice)        # produit matriciel matrice*J
np.dot(id,J)             # =J
np.dot(J,id)             # =J
np.dot(matrice,J)        # erreur dû au dimensions
np.dot(matrice.T,J)

#Puissances
np.linalg.matrix_power(J,5)

#Inverse
np.linalg.inv(J)         # erreur, pas inversible
np.linalg.inv(id)        # elle-même
np.linalg.inv(id+J)      # inverse (valeurs approchées)

#Système matriciel
b = np.array([0,0,4,5,-1])
a = np.ones(5)-np.tri(5,k=-1)
np.linalg.solve(a,b)     # résout ax=b
```

Python VIII.22: Numpy : Operations matricielles

La résolution des systèmes avec `linalg.solve` ne fonctionne que lorsqu'il s'agit d'un système de Cramer.

4 Tests avec les tableaux

Pour tester si deux tableaux sont égaux (terme à terme), on utilise la commande `array_equal`.

Nota : la commande "`a==b`" renvoie une matrice de même taille que `a` et `b` contenant `True` ou `False` pour chaque position dans le tableau. C'est le tableau des $(a_k == b_k)k \in I$.

Il en est de même avec les autres opérateurs logiques.

```
a = np.ones((3,5))
b = a.copy()
c = eye(3,5)
np.array_equal(a,b)          # True
np.array_equal(a,c)          # False
a == b                       # Tableau de taille 3x5 rempli de True
```

Python VIII.23: Numpy : Égalité de tableaux

On peut aussi utiliser les méthodes "`.any()`" et "`.all()`" qui testent respectivement si l'un ou la totalité des éléments du tableau valent "`True`".

```
a = np.ones((3,5))
b = a.copy()
c = np.eye(3,5)
d = np.zeros((3,5))
(a == b).any()        # True
(a == b).all()        # True
(a == c).any()        # True
(a == c).all()        # False
(a == d).any()        # False
(a < 1).any()         # False
(c < 1).any()         # False
```

Python VIII.24: Numpy : méthodes any et all

Remarque importante : La fonction `allclose` teste si deux tableaux sont égaux à une incertitude près (absolue et relative). Souvent, cette fonction sera plus utile que le test de l'égalité exacte. En effet, Python réalise des erreurs d'arrondis dans ses calculs, et souvent, deux objets qui devraient être rigoureusement égaux mathématiquement, diffèrent très légèrement dans Python pour ces raisons d'arrondis.

On pourra aussi consulter la fonction `isclose`.

5 Des statistiques et probabilités avec Numpy

A Fonctions statistiques

```
serie = np.random.randint(10,100)
serie.max()                              # max
serie.min()                              # min
serie.sum()                              # somme
serie.mean()                             # moyenne
serie.mean()==serie.sum()/serie.size     # True
serie.var()                              # variance
serie.std()                              # écart-type
serie.median()                           # médiane
```

Python VIII.25: Numpy : Statistiques univariées

Python donne aussi la matrice de covariance de deux séries statistiques. La matrice est ainsi définie

$$\text{cov(x,y)} = \begin{pmatrix} \sigma_x^2 & \sigma_{x,y} \\ \sigma_{x,y} & \sigma_y^2 \end{pmatrix}$$

La coefficient de corrélation est aussi donné dans la matrice de corrélation

$$\text{cov(x,y)} = \begin{pmatrix} 1 & \rho(x,y) \\ \rho(x,y) & 1 \end{pmatrix}$$

```
seriea = np.random.randint(10,100)
serieb = np.random.randint(10,100)
np.cov(seriea,serieb)
np.corrcoef(seriea,serieb)
```

Python VIII.26: Numpy : Statistiques bivariées

Dans la suite de cette section, on utilisera le sous-module `random` de numpy. Nous l'avons déjà utilisé pour créer des matrices avec des coefficients aléatoires dans le paragraphe G.

B Mélange aléatoire d'un tableau

La commande `random.shuffle` permet de mélanger un tableau en place.

```
urne = np.arange(10)
np.random.shuffle(urne)                  # urne est modifiée
```

Python VIII.27: Numpy : Mélange d'un tableau

C Tirage dans une urne

La commande `random.choice` permet de tirer un élément (ou un échantillon) de façon aléatoire dans un tableau à une dimension.

Lorsque l'on tire plusieurs éléments successifs, l'argument optionnel `replace=False` permet de faire un tirage sans remise.

```
urne = np.arange(10)
elt = np.random.choice(urne)             # tirage d'un élément dans urne
np.random.choice(urne,5)                 # tirage de 5 éléments avec remise
np.random.choice(urne,5,replace=False)   # tirage de 5 éléments sans remise
np.random.choice(urne,11,replace=False)  # erreur
np.random.choice(6,1,replace=False)      # tirage de 1 élément dans [0,6[
```

Python VIII.28: Numpy : Tirage dans une urne

On peut aussi rajouter un argument optionnel pour préciser les probabilités de tirage de chaque élément (par défaut, les tirages sont équiprobables).

D Loi binomiale

La loi binomiale $\mathscr{B}(n,p)$ est modélisée par la fonction `random.binomial(n,p)`. Celle-ci renvoie le nombre de succès après n obtenus lors de la réalistion de n épreuves de Bernoulli indépendantes. La probabilité de succès à chaque épreuve est donnée par $p \in [0,1]$.

La probabilité d'obtenir k succès est alors

$$P(X=k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Un argument optionnel `size=nb` permet de réaliser `nb` fois l'expérience.

```
np.random.binomial(50,0.6)          # nombre de succès selon la loi binomiale B(n,p)
np.random.binomial(50,0.6,size=10)  # réaliser 10 fois l'expérience
np.random.binomial(50,0.6,10)       # idem
```

Python VIII.29: Numpy : Loi binomiale

E Loi géométrique

La loi géométrique est modélisée par la fonction `random.geometric(n,p)`. La fonction renvoie le numéro du tirage correspondant au premier succès lors de la répétition d'épreuves de Bernoulli indépendantes et de paramètre $p \in [0,1]$.

La probabilité d'obtenir le premier succès au $k^{\text{ième}}$ tirage est alors

$$P(X=k) = (1-p)^{k-1} p$$

```
np.random.geometric(0.001)          # premier succès pour p=0.001
np.random.geometric(0.001,size=10)  # réaliser 10 fois l'expérience
np.random.geometric(0.001,10)       # idem
```

Python VIII.30: Numpy : Loi géométrique

F Loi hypergéometrique

La loi hypergéométrique est modélisée par la fonction `random.hypergeometric(ngood,nbad,nsample)`. La fonction renvoie le nombre de succès lors du tirage (sans remise) d'un échantillon de taille `nsample` dans une urne contenant `ngood` cas favorables et `nbad` cas défavorables.

La probabilité d'obtenir k succès est alors

$$P(X=k) = \frac{\binom{\texttt{ngood}}{k}\binom{\texttt{nbad}}{\texttt{nsample}-k}}{\binom{\texttt{ngood+nbad}}{\texttt{nsample}}}$$

```
np.random.hypergeometric(80,20,5)        # nombre de boules blanches lorsque l'on en
    tire 5 dans un urne qui en contient 80 et 20 noires
np.random.hypergeometric(80,20,5,size=10)# réaliser 10 fois l'expérience
np.random.hypergeometric(80,20,5,10)     # idem
```

Python VIII.31: Numpy : Loi hypergéométrique

6 Quelques commandes pour aller plus loin

Ces commandes sont présentées ici pour leur intérêt intrinsèque, mais aussi parce que nous les utiliserons dans le TD sur les images.

A Notation d'Einstein pour les sommes

Cette écriture des sommes n'est pas due à Einstein lui-même, mais il l'a popularisé.

Elle est très largement utilisée par les Physiciens.

Ce n'est qu'une notation. Elle n'apporte donc rien de nouveau conceptuellement, mais elle permet d'alléger les écritures.

L'idée de base et de ne pas écrire le signe somme $\sum$, mais de le laisser sous-entendu.

Lorsqu'un indice se répète entre deux coefficients, alors, il s'agit d'une somme sur cet indice (les bornes de sommation sont sous-entendues).

Par exemple, le produit matriciel peut s'écrire très simplement :

$$a_{i,k}b_{k,j} = \sum_{k=1}^{n} a_{i,k}b_{k,j}$$

Dans la première écriture, l'indice k se répète et indique donc que l'on fait la somme de ces produits pour k variant dans son ensemble d'indices naturels.

L'indice c_k du produit entre deux polynômes s'écrit :

$$a_i b_{k-i} = \sum_i a_i b_{k-i}$$

Ici, c'est l'indice i qui est répété et qui sert donc d'indice de sommation.

On peut aussi avoir des sommes doubles, triples... si on repète plusieurs indices.

Numpy propose une méthode de calcul très puissante qui se base sur cette notation.

Pour simplifier, nous ne l'utiliserons qu'avec deux tableaux.

Exemple avec le produit matriciel :

On part de deux matrices A et B avec deux dimensions chacune. Chaque coefficient est donc défini par deux coordonnées. On les a appelé ik pour A et kj pour B.

À partir de ces coefficients, on construit une nouvelle matrice aussi à deux dimensions ij, dont les coefficients valent avec la notation d'Einstein

$$C[i,j]] = A[i,k]B[k,j] = \sum_k A[i,k]B[k,j]$$

On écrit donc le code :

```
B = np.array([[1,0,0],[1,2,1],[-1,0,1]])

C = np.einsum('ik,kj->ij',A,B)
(C == A.dot(B)).all()
```

La dernière commande sert à vérifier que l'on obtient la même matrice qu'avec le produit matriciel obtenu par la commande `dot`.

Somme des coefficients diagonaux d'une matrice :

```

tr = np.einsum('ii->',A)
```

Extraction de la diagonale :

On construit le tableau dont les coefficients sont ceux de la diagonale de A : on extraie les $A[i,i]$. Pour cela, il ne faut pas faire un somme d'indice i. Cet indice i ne disparait donc pas et se retrouve dans le résultat (après le flèche).

```

D = np.einsum('ii->i',A)
```

(a) Fonction vstack

(b) Fonction hstack

(c) Fonction tile

Figure VIII.11: Manipulation d'images

B Modifier la forme du tableau, les concaténer...

Ces fonctions sont particulièrement utiles lors d'un travail sur des images, la figure VIII.11 donne des exemples.

Les fonctions de concaténation

- **vtack** concatène deux tableaux *verticalement* : les lignes sont mises à la suite les unes des autres (voir la figure VIII.11a).
- **hstack** réalise la même chose avec les colonnes (voir la figure VIII.11b).
- **concatenate** généralise les fonctions précédentes en donnant le choix de l'axe suivant lequel faire la concaténation.

  Par exemple `concatenate((A,B),axis = 0` revient à faire `vstack(A,B)` et `concatenate((A,B),axis = 1` revient à faire `hstack(A,B)` et

La fonction de réplication **tile(A,reps)** crée un nouveau tableau en répétant A, un certain nombre de fois.

`reps` indique les répétitions à effectuer. Par exemple, si A est une matrice de taille 2×2, alors `reps` doit être un couple. Chaque coefficient correspond au nombre de répétitions de l'axe considéré.

Par exemple, si A est de dimensions $M\times N$, alors `tile(A,[2,3])` est de dimension $2M\times 3N$. Chaque ligne de A est "doublée" et chaque colonne est triplée.

```
A = np.array([[1, 2], [3, 4]])
  array([[1, 2],
         [3, 4]])
np.tile(A, 2)
  array([[1, 2, 1, 2],
         [3, 4, 3, 4]])
```

```
np.tile(A, (2, 1))
  array([[1, 2],
         [3, 4],
         [1, 2],
         [3, 4]])
B = np.array([1,2,3,4])
np.tile(B,(4,1))
  array([[1, 2, 3, 4],
         [1, 2, 3, 4],
         [1, 2, 3, 4],
         [1, 2, 3, 4]])
```

On a utilisé cette fonction pour l'image VIII.11c. Cela revient à utiliser la fonction `concatenate` avec plusieurs fois le même tableau.

La commande newaxis permet de créer un nouvel axe (une nouvelle dimension) dans un tableau lors d'un slicing.

```
A = np.array([[1, 2], [3, 4]])
A.shape
    (2,2)
A[:,newaxis,:].shape
    (2,1,2)
```

C Comparaisons avancées entre tableaux :

La fonction where renvoie un tableau issu de la comparaison de deux autres.

Par exemple, si x et y sont deux tableaux de même taille, `np.where(x<y,x,y)` renvoie un tableau de même taille où chaque coefficient est le plus petit entre celui de x et celui de y.

`np.where(x<y,100,y)` renvoie le tableau où chaque coefficient vaut 100 si le coefficient de x est plus grand que celui de y, et celui de y sinon.

```
x = np.arange(9.).reshape(3, 3)
  array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
np.where(x < 5, x, -1)
  array([[ 0.,  1.,  2.],
         [ 3.,  4., -1.],
         [-1., -1., -1.]])
```

Les fonctions maximum, minimum sont des cas particuliers de `where`.

`maximum(x,y)` renvoie le tableau dont chaque coefficient est le plus grand entre celui de x et celui de y. C'est la même chose que `np.where(x1 >= x2, x1, x2)`.

```
np.maximum([2, 3, 4], [1, 5, 2])
  array([2, 5, 4])
```

D À vous de chercher

Le module numpy possède de nombreuses autres possibilités : recherche dans un tableau, calculs statistiques, calculs avec les polynômes (Lagrange...)...À vous d'aller trouver la bonne fonction si vous ne voulez pas la programmer à nouveau.

N'hésitez pas à consulter l'aide en ligne : https://docs.scipy.org/doc/numpy/.

Revenir au sommaire

IX - MANIPULATION DES FICHIERS PAR PYTHON

1 Motivation

Un programme sera souvent amené à traiter un grand nombre de données et à en générer.
Il ne serait pas réaliste de vouloir tout rentrer à chaque fois à la main avec des commandes `input`. Nous n'allons pas non plus écrire ces données directement dans le programme, car il faudrait modifier notre script pour chaque changement de données, au risque d'introduire de nombreuses erreurs. Nous utiliserons donc des fichiers externes que Python lira et éventuellement modifiera.
Ceci permet aussi de pouvoir sauvegarder des données générées par Python : par exemple des scores pour un jeu, ou bien des données statistiques sur une étude... L'avantage de ce fichier et qu'il restera en mémoire même après la fermeture de Python, contrairement aux variables qui sont effacées lorsque l'on ferme Python. De la sorte, vous pourrez par exemple traiter un fichier de données généré par un autre logiciel pour en extraire les informations qui vous intéressent, les enregistrer dans un fichier `.csv`, puis les ouvrir ensuite avec un tableur.

2 Ouverture, fermeture et parcours d'un fichier texte

L'ouverture d'un fichier se fait avec la commande `open`.
Qui dit ouverture, dit parfois *création* lorsque le fichier n'existe pas encore.

Bien sûr, tout ceci nécessitera d'avoir les bons droits d'accès au fichier en lecture et/ou en écriture comme nous l'avons vu en début d'année.

Par exemple, pour créer un fichier[28] `test.txt` dans le dossier `C:/MonDossier/` :

```
1 #création du fichier et ouverture en mode écriture.
2 f=open("C:/MonDossier/test.txt","w")
3 f.close()                              #fermeture
4
5 f=open("C:/MonDossier/test.txt",mode="w") #synonyme de la première instruction.
6 f.close()                              #fermeture
```

Python IX.1: Ouverture/Fermeture d'un fichier

- Le premier argument correspond au chemin d'accès au fichier. Notez bien l'utilisation des "/" et non des "\" pour indiquer les dossiers, quelque soit l'OS.
- Le deuxième argument indique le mode d'ouverture. Les principaux modes possibles sont expliqués dans le tableau IX.9.

[28] S'il n'existe pas encore

- on peut aussi rajouter un argument pour spécifier l'encodage du texte (voir la section 4), et quelques autres options...

mode	comportement
r	Ouvre en mode lecture seule. Le flux est positionné au début du fichier.
r+	Comme **r**, mais permet aussi l'écriture.
w	Crée un nouveau fichier et l'ouvre en écriture. S'il existait déjà un fichier du même nom, il l'écrase.
x	Comme **w**, mais renvoie une erreur s'il existe déjà un fichier du même nom. Évite d'écraser un fichier par mégarde[29].
w+	Comme **w**, mais permet aussi la lecture.
a	Ouvre le fichier en écriture, crée le fichier s'il n'existe pas. Le flux est positionné en **fin** de fichier. L'écriture est toujours ajoutée en fin de fichier.
a+	Comme **a**, mais permet aussi la lecture.

Table IX.9: Principaux modes pour l'ouverture d'un fichier en mode texte.

Attention : Un fichier est considéré comme un flux (*stream*) que l'on parcourt. Les opérations sont faites directement à l'endroit où l'on se trouve. Lorsque l'on ouvre, en fonction du mode choisi, on est soit au début, soit à la fin du fichier, et les commandes de lecture et d'écriture n'auront pas le même effet.
Si vous lisez le fichier alors que vous êtes placé au début, vous aurez l'intégralité du fichier et vous retrouverez à la fin.
Si vous voulez lire à nouveau, vous n'aurez plus rien devant vous : il faut déjà revenir au début pour recommencer à lire.

La métaphore du flux se veut assez parlante. Vous êtes comme une embarcation sur une rivière : vous n'avez pas la vision globale de la rivière, mais uniquement des quelques mètres devant vous.
Si vous voulez parcourir la rivière, il faut d'abord aller à la source et puis tout descendre. Tant que vous n'êtes pas arrivé au bout, vous ne savez pas du tout s'il vous reste beaucoup ou très peu à parcourir.
Une fois arrivé au bout, si vous voulez revoir toute la rivière, il faut déjà repartir à la source avant de recommencer la descente.
La méthode `seek` permet d'avancer à une certaine position. Avec la même métaphore, vous pouvez demander à aller directement au kilomètre 25 sur la rivière. Aller à la source, c'est revenir au kilomètre 0.

Voici quelques commandes que vous pouvez essayer (changez l'endroit du fichier `C:/MonDossier/` en fonction de votre arborescence personnelle). Ces scripts sont à écrire les uns à la suite des autres.

⚠ si vous chargez la bibliothèque `os`, alors il utilisera la fonction `open` de cette bibliothèque qui a un comportement différent de celui présenté ici.

```
#*****************************************************************************
# Crée un fichier et écrit un texte dedans. Si le fichier existe, il est écrasé.

f = open("C:/MonDossier/test.txt","w")
f.write("Python est le nom scientifique d'un genre de serpents ")
f.write("de la famille des Pythonidae.")
f.close()                              #fermeture

#*****************************************************************************
# Remplace les premiers caractères du fichier par le mot 'Ecrase'

f = open("C:/MonDossier/test.txt","r+") #flux au début du fichier.
f.write("Ecrase")
f.close()                              #fermeture

#*****************************************************************************
# Différents parcours de fichier avec lectures

f = open("C:/MonDossier/test.txt","r+") #flux au début du fichier.
f.read() #flux à la fin du fichier
```

[29] Cette fonctionnalité n'existe que depuis Python 3.3. Notez qu'un fichier écrasé est définitivement perdu, il n'est pas envoyé à la corbeille.

```
f.read()
f.seek(0) #place le flux à l'indice 0, au début
contenu = f.read()
type(contenu)
print(contenu)
f.write("\nC'est la fin")

#*****************************************************************************
# Lit globalement par lignes

f.seek(0)
lignes = f.readlines() #tableau de toutes les lignes du fichier
f.close()
for s in lignes:
  print(s,end = "") #empêche print de mettre un entrée a la fin.

# résultat similaire
for s in open("C:/MonDossier/test.txt","r+"):
  print(s,end = "")
```

Python IX.2: Exemples de manipulations de fichiers

La commande `read` peut prendre un argument qui indique le nombre (maximum) de caractères à lire.
Le caractère "`\n`" utilisé avec la méthode `write` indique un passage à la ligne.

Un objet et ses méthodes
Vous observez que dans l'exemple précédent, on ne travaille pas directement sur le fichier, mais sur un objet obtenu par la fonction `open`. On a donné à cet objet le nom `f` qui n'a rien à voir avec le nom du fichier.
Cet objet propose de nombreuses *méthodes* : qui sont comme des fonctions embarquées, ou attachées à cet objet.
On fait appel à une méthode en faisant suivre l'objet d'un point "." et du nom de la méthode avec ses arguments.
par exemple, pour appeler la méthode de fermeture du fichier, on écrit `f.close()`.

Nous n'insistons pas là dessus, mais sachez que toute la programmation actuelle s'appuie sur ce concept : on parle de *programmation objet* : on crée des objets qui ont des attributs et des méthodes.
Nous avons déjà utilisé certaines méthodes avec les listes par exemple. On dit que chaque liste que vous créez, est une *instanciation* de la classe "`list`". Elle en hérite les méthodes `append`, `remove`...

Exemple

Comptez le nombre de lignes d'un fichier.

```
#source est le nom du fichier avec le chemin d'accès.
fs = open(source, 'r')
nb = 0
for line in fs:
  nb += 1
print (nb) #écrit le nombre de lignes du fichier
```

Python IX.3: Nombre de lignes d'un fichier

Exemple

1) Programmez une fonction `copieFichier` qui copie la `source` vers la `destination`.

2) Programmez une fonction `copieFichier2` qui fait la même chose, mais en utilisant la méthode `readlines`.

3) Faites de même, une fonction `copieFichier3` avec la méthode `readline`.

Solution :

```
def copieFichier(source, destination):
  '''copie intégrale d'un fichier source vers un fichier destination'''
  fs = open(source, 'r')
  fd = open(destination, 'w')
  fd.write(fs.read())
```

```
  fs.close()
  fd.close()

def copieFichier2(source, destination):
  '''copie intégrale d'un fichier source vers un fichier destination
  ligne à ligne'''
  fs = open(source, 'r')
  fd = open(destination, 'w')
  for ligne in fs.readlines():
    fd.write(ligne)
  fs.close()
  fd.close()

def copieFichier3(source, destination):
  fs = open(source, 'r')
  fd = open(destination, 'w')
  test = True
  while test:
    ligne = fs.readline()
    if ligne == '':
      test = False
    else:
      fd.write(ligne)
  fs.close()
  fd.close()
```

Python IX.4: Copie d'un fichier

3 Où suis-je ?

Principe du chemin dans l'arborescence :
Pour indiquer un emplacement, on utilise un chemin dans l'arborescence. Par exemple, pour trouver `C:/MonDossier/test.txt`, on va dans le disque `C:`, puis dans le dossier `MonDossier`, dans lequel on trouve le fichier `text.txt`.

Caractère / :
Windows utilise habituellement le caractère "\" pour séparer les étapes successives du chemin dans l'arborescence. Unix (et donc Mac, Linux...) utilisent le caractère "/". C'est ce dernier caractère qu'utilise Python, même si vous êtes sous Windows (car le caractère \ est un caractère d'échappement qui a une autre signification).

Chemin absolu versus chemin relatif :
Le **chemin absolu** est un chemin qui part de la *racine* de l'arborescence (par exemple le nom du disque sous Windows, ou simplement le dossier racine sous Unix), pour descendre jusqu'à l'emplacement souhaité.
Le **chemin relatif** est un chemin à partir du dossier de travail (par exemple le dossier Mes documents, ou le dossier home).
Pour donner une analogie géographique, donner le chemin absolu pour trouver une ville dans un pays, c'est par exemple donner le chemin pour y aller depuis la capitale. En revanche, si je sais que je suis à Saint Maur et si je souhaite aller à Nogent-sur-Marne, il peut être plus judicieux de donner la route à suivre depuis mon emplacement (Saint-Maur) et non depuis la capitale.
L'avantage du chemin absolu est qu'il est valable même si je ne sais pas bien où je suis (et j'arrive toujours à revenir facilement au dossier racine).
L'avantage du chemin relatif est que si je travaille uniquement dans un dossier (et ses sous-dossiers) et que je déplace tout ce dossier sur une nouvelle machine, mon chemin relatif restera valable alors que le chemin absolu devra être modifié.
En général, on privilégie le chemin relatif quand c'est possible.

Le dossier courant :
Utiliser le chemin relatif nécessite de savoir où l'on est. On l'obtient avec la commande `pwd` en Python.
Une fois que l'on connaît le dossier de travail de Python, on peut se déplacer dans l'arborescence facilement.

Se déplacer dans l'arborescence :
Le point "." indique le dossier courant (ou dossier de travail).

Les double points ".." indiquent que l'on monte au dossier parent.
Le slash en début de chemin "/" indique le dossier racine.
Le tilde " ~ " indique le dossier *home*.

Par exemple,

- pour aller dans le sous dossier `info` y chercher le document `text.txt`, on écrit le chemin `./info/text.txt`.
- pour aller remonter d'un dossier pour chercher le document `test.py`, on écrit le chemin `../test.py`.

La bibliothèque **os** dans python permet de faire des manipulations comme dans le shell (sous réserve de disposer des droits).

4 Encodage

Nous n'insistons pas particulièrement sur l'encodage ici, car les programmes que vous rédigerez seront généralement réservés à un usage très local.
L'encodage correspond à la manière de coder chaque caractère en "langage machine", et cela diffère en fonction des besoins des utilisateurs. Ainsi, les francophones ont besoin d'accents, les russes utilisent l'alphabet cyrillique...
Cela fait qu'au cours de l'histoire (informatique), en fonction des besoins des utilisateurs, différentes façon de coder les caractères ont été utilisées. Votre OS utilise un encodage par défaut, et on s'arrange souvent pour que les logiciels en héritent tous. C'est la raison pour laquelle, Python devrait utiliser le même encodage que tous les logiciels sur un même PC (et plus généralement dans une région) et que vous n'aurez pas de problèmes à ce niveau.
Mais si vous récupérez des fichiers de l'extérieur, le problème peut arriver et il faut y penser. Il vous est sûrement déjà arrivé d'ouvrir un fichier ou un mail où les lettres accentuées devenaient des points d'interrogation ou des caractères bizarres. C'est simplement qu'il était ouvert avec le mauvais encodage (même si la plupart des logiciels intègrent des systèmes pour deviner le bon encodage).
Lors de l'ouverture du fichier, en cas de doute, vous pourrez donc préciser l'encodage du fichier pour éviter des erreurs. Mais cela demande bien sûr de savoir quel est cet encodage !
De même, pour indiquer à Python dans quel encodage est écrit le fichier source, il est conseillé d'écrire en première ligne du fichier qui contient votre programme (ou en deuxième) :

- si vous travaillez en encodage Latin-1 aussi appelé ISO-8859-1 (en général Windows) :

```
1 # -*- coding:Latin-1 -*-
```

- si vous travaillez en utf-8

```
1 # -*- coding:Utf-8 -*-
```

Si vous voulez en savoir plus, n'hésitez pas à lire l'excellent article (très simple) :
http://www.joelonsoftware.com/articles/Unicode.html traduit ici.

Revenir au sommaire

X - LISTE D'ALGORITHMES À CONNAÎTRE

1 Suites récurrentes

```
# suite récurrente d'ordre 1
# exemple simple u(n+1) = u(n)**2 + 1
def recc(u0,n):
    u = u0
    for i in range(n):
        u = u**2 + 1
    return u

# suite non majorée : premier rang >= M
def rang(u0,M):
    u = u0; n = 0
    while u < M:
        u = u**2 + 1
        n += 1
    return n

# suite non majorée : dernier terme < M
# on suppose M > u0
def terme(u0,M):
    u = u0
    while u < M:
        (u,v) = (u**2 + 1,u)
    return v

# conjecture sur la croissance de la suite
def croissante(u0,n):
    u = u0
    for i in range(n):
        (u,v) = (u**2 + 1, u)
        if u < v:
            return False
    return True


# somme des termes de la suite
def somme(u0,n):
    u = u0; s = u0
```

```
    for i in range(n):
        u = u**2 + 1
        s += u
    return s

# suite récurrente d'ordre 2
# exemple simple u(n+2) = 2.u(n+1)-u(n)+3
def recc2(u0,u1,n):
    (u,v) = (u0,u1)
    for i in range(n):
        (u,v) = (v,2*v-u+3)
    return u
```

2 Parcours de liste

```
# somme des éléments d'une liste de nombres
def somme(liste):
    s = 0
    for k in liste:
        s += k
    return s

# maximum d'une liste de nombres
def maximum(liste):
    m = liste[0]
    for i in liste[1:]:
        if i > m:
            m = i
    return m

# (premier) indice du maximum d'une liste de nombres
def maxIndice(liste):
    m = 0
    for i in range(1,len(liste)):
        if liste[i] > liste[m]:
            m = i
    return m

# sous-liste des éléments > M
def listeSup(liste,M):
    res = []
    for i in liste:
        if i > M:
            res.append(i)
    return res

# Par compréhension
def listeSup(liste,M):
    return [i for i in liste if i>M]

# liste de 20 nombres aléatoires entre 1 et 6
[randrange(1,7) for i in range(20)]

# moyenne pondérée arrondie à 0,1
def moyPonderee(notes,coeffs):
    s,c = 0,0
    for i in range(len(notes)):
        s += notes[i]*coeffs[i]
        c += coeffs[i]
    return round(s/c,1)
```

3 Tris de liste avec effets de bord

```
def triSelection(liste):
    for i in range(len(liste)):
        mini = i
        for j in range(i+1, len(liste)):
            if liste[j] < liste[mini]:
                mini = j
        liste[i],liste[mini] = liste[mini],liste[i]

def triInsertion(liste):
    for i in range(len(liste)):
        j = i
        while j>0 and liste[j] < liste[j-1]:
            liste[j],liste[j-1] = liste[j-1],liste[j]
            j = j-1
# ou
def triInsertion2(liste):
    for i in range(len(liste)):
        j = i
        carte = liste.pop(i)
        while j>0 and carte < liste[j-1]:
            j = j-1
        liste.insert(j,carte)

def triBulles(liste):
    chaos = True
    while chaos:
        chaos = False
        for i in range(len(liste)-1):
            if liste[i] > liste[i+1]:
                liste[i],liste[i+1]= liste[i+1],liste[i]
                chaos = True
#ou
def triBulles2(liste):
    fin = len(liste)
    while fin > 1:
        last = 0
        for i in range(fin-1):
            if liste[i] > liste[i+1]:
                liste[i],liste[i+1]= liste[i+1],liste[i]
                last = i+1
          fin = last
```

4 Fonctions

```
# tracé cos
x = linspace(0,2*pi, 100)
y = cos(x)
plot(x,y)

#dichotomie
def dichotomie(f,a,b,eps = 1e-3):
    while abs(b-a) > eps:
        c = (a+b)/2
        if f(c) == 0:
            return c
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
    return (a+b)/2
```

5 Tirages aléatoires

Revenir au sommaire