Programmation 1 **année 2013-2014** GB3, 2^{nd} semestre 2013-2014

Polytech'Nice Notes de cours Gilles Bernot

Initiation à la programmation impérative avec Python

Avertissement au lecteur :

Ce polycopié ne contient pas toutes les explications détaillées qui sont données en cours. En particulier tous les développements systématiques des exemples, expliquant comment le langage Python effectuerait les traitements, sont parfois absents de ces notes.

G. Bernot

Ce plan n'est que prévisionnel et est soumis à changements

COURS 0

1. Domaines de la bio-informatique
2. Métiers de la bio-informatique

COURS 1

1. L'art de programmer
2. Comment fonctionne un ordinateur, dans les grandes lignes
3. Quelques unités de capacité d'information
4. Les langages de programmation
5. Un exemple en Python

COURS 2

1. Les structures de données
2. Les booléens
3. Les entiers relatifs
4. Les nombres réels
5. Les chaînes de caractères
6. Opération de substitution de chaînes de caractères
7. Des conversions de type (cast)

COURS 3

1. Les expressions conditionnelles
2. Les définitions de fonctions
3. Les définitions de procédures
4. Variables locales et variables globales
5. Compléments sur les chaînes de caractères, et `while`
6. Parcours de chaîne de caractères avec `for`

COURS 4

1. Compléments mineurs mais utiles sur le type `str`

2. Les listes
3. Instructions de modification de liste
4. Particularités des modifications de liste en Python
5. Exemple de programme sur les listes

COURS 5

1. La programmation structurée

COURS 6

1. Les dictionnaires
2. Exemples de programmes sur les dictionnaires

COURS 7

1. Lecture de fichiers
2. Quelques méthodes utiles de lecture de fichiers
3. Écriture de fichiers

COURS 8

1. Les modules en Python
2. Gestion des fichiers : le module `os`
3. Créer son propre module en Python

Les notes de cours et les feuilles de TD sont disponibles (avec un peu de retard par rapport au déroulement du cours) à l'adresse web suivante :

`http://www.i3s.unice.fr/~bernot/Enseignement/GB3_Python1/`

1 Domaines de la bio-informatique

L'objectif de ce cours de *Programmation en langage de script* est d'apprendre les bonnes pratiques de la programmation. Ce qui motive cet enseignement dans une formation de génie biologique est qu'à l'heure actuelle aucun cadre scientifique, quelle que soit sa discipline de base, ne peut prétendre être un *ingénieur* s'il ne connaît pas les principes de la programmation. À l'issue de ce cours, vous saurez programmer en Python. Python est le langage de programmation le plus utilisé par les biologistes. Il faut de plus être conscient que les principes de programmation sont les mêmes pour la plupart des langages de programmation. Ainsi, à la sortie de ce cours, vous serez en fait capables de programmer avec presque n'importe quel langage de programmation, pourvu que vous ayez sous la main un petit résumé des mots-clefs de ce langage.

Ce cours de programmation n'est pas un cours de bio-informatique. La bio-informatique est une activité où la biologie et l'expérimentation sont à la base de la *prédiction* de propriétés biologiques. Si la programmation entre en jeu en bio-informatique, c'est uniquement de manière annexe parce que les données et des connaissances manipulées sont d'un volume tel que leur gestion requiert un ordinateur. Ce qui fait un bon ingénieur en bio-informatique n'est pas la programmation ; c'est un *mode de raisonnement* adapté au vivant pour produire de bonnes prédictions et cela suppose une bonne compréhension des protocoles expérimentaux aussi bien *in vivo* que *in vitro* et *in silico*.

Pour éviter la confusion fréquente entre la programmation et la bio-informatique, il semble utile d'inventorier comment la bio-informatique intervient dans tous les domaines de la biologie. La programmation ne sera abordée qu'au cours n°1.

1.1 Domaines de la biologie (panels ERC)

(ERC = European Research Council)

- *Biologie moléculaire et cellulaire* (biologie structurale, signalisation, métabolisme, cycles divers, apoptose...) : la bio-informatique permet d'assembler et annoter les séquences biologiques, elle prédit la structure de repliement des bio-molécules à partir de connaissances sur les séquences, sur l'énergie de repliements élémentaires, *etc.* Elle établit des modèles d'interactions et d'échanges de signaux entre entités biologiques, propose des comportements dynamiques possibles, inventorie les comportements stables, *etc.*
- *Génomique fonctionnelle* (omics, génétique, épigénétique, réseaux...) : la bio-informatique classifie les mesures aquises, évalue leur qualité, en déduit des profils d'expression, quantifie la pertinence de certaines interactions putatives, prédit la ou les fonction(s) d'un gène ou d'un réseau génétique, optimise les stratégies expérimentales en fonction des objectifs, *etc.*
- *Organes et physiologie* (systèmes, métabolisme et pathologies, infection et immunité, vieillissement...) : la bio-informatique fournit des cadres méthodologiques pour étudier les systèmes biologiques complexes, établit des lois comportementales prédictives, offre des simulations crédibles, limite les expérimentations animales, assiste l'analyse d'images, *etc.*
- *Neurosciences* (neuro..., imagerie, cerveau,...) : la bio-informatique met en place des modèles mathématiques explicatifs, prédit certains comportements, gère des simulations *in silico*, réfute des hypothèses sur la base des comportements émergeants, analyse les mesures, reconstruit la structure spatiale, *etc.*
- *Populations* (évolution, phylogénie, écologie, écotoxicologie, environnement, santé...) : la bio-informatique organise les données, en extrait des connaissances, classifie, établit des graphes de phylogénie, simule les individus pour prédire des phénomènes collectifs émergeants, établit des normes de sécurité, prédit les effets d'une perturbation, permet de tester des modifications planifiée de l'environnement (e.g. autoroute traversant une forêt), *etc.*
- *Médecine santé* (pharmacologie, diagnostic, informatique médicale) : la bio-informatique met en place des systèmes experts, établit des modèles prédictifs de comportement tant au niveaux cellulaire que tissulaire ou organique, permet le criblage de molécules, organise et stocke les informations médicales, les exploite pour extraire des connaissances, *etc.*

1.2 Domaines pertinents de l'informatique dans le cadre de la biologie

La bio-informatique exploite presque tous les domaines de l'informatique.

- *Algorithmique* (comment gérer une question reposant sur beaucoup de données en entrée pour la résoudre en un temps acceptable) : par exemple pour le traitement et la comparaîson des séquences génomiques, la classification des données, le calcul de repliement de biomolécules, l'étude systématique de grand nombre de modèles comportementaux, *etc*
- *Systèmes d'information* (gestion des bases de données et des interfaces pour les exploiter) : gestion de toutes les données omics, représentation des grands réseaux biologiques, mémorisation des protocoles expérimentaux, suivis divers, patients, gestion de bibliographie, tracabilité, *etc.*
- *Intelligence artificielle* (méthodes heuristiques pour résoudre la plupart du temps des questions qui seraient hors de portée d'une machine si elles étaient traitées exhaustivement et de manière exacte) : extraction de connaissances à partir de données biologiques, classification fonctionnelle, diagnostic à partir de données à grande échelle, pharmacovigilence, *etc.*
- *Modélisation et simulation* (créer des modèles mathématiques ou virtuels d'objets réels, raisonner automatiquement dessus ou les simuler) : biologie des systèmes, biologie intégrative, réseaux biologiques, embryologie 3D, reconstruction d'organes, cellules virtuelles, prédictions en chronobiologie, chronopharmacologie, *etc.*

1.3 Domaines de l'ingénierie biologique (panels ERC)

Les domaines d'activité des ingénieurs bio-informaticiens couvrent tous les domaines du génie biologique car tous ont une composante bio-informatique. Pour reprendre la classification ERC, on peut citer :

- *Biotechnologies* : tous les domaines (biotechnologie rouge, verte, blanche ou bleue) doivent faire appel à la bio-informatique pour organiser les données, prédire l'effet des modifications planifiées du génome, optimiser les bioréacteurs, la bio-informatique fait l'objet de nombreuses « niches » des startups, gère de bout en bout les puces à ADN et les pyroséquenceurs, *etc.*
- *Pharmacologie* : drug design, cosmétique, toxicologie, sécurité, *etc.*
- *Environnement, écologie* : développement durable, toxicologie, sécurité, *etc.*
- *Agro-alimentaire* : OGM, sécurité, toxicologie, *etc.*
- *Biologie de synthèse* : biocarburants, diagnostiques, antibiotiques, *etc.*
- *Recherche* en biologie : publique ou privée.

... Et il faut aussi mentionner l'*informatique classique*, en raison de la forte pénurie d'informaticiens diplômés, en France et dans le monde, qui rend les DRH du secteur informatique avides de jeunes scientifiques ayant touché de près ou de loin à l'informatique.

2 Métiers de la bio-informatique

Tout ce que ni un informaticien non biologiste, ni un biologiste non bio-informaticien, ne saura jamais faire :

***Maître d'ouvrage de projets multi-disciplinaires, chef de projet* :** de nombreux projets de biologie moderne font appel à toutes les autres disciplines scientifiques (physique, chimie, robotique, informatique, mathématique) voire aux sciences humaines (droit, éthique). Pour mener à bien de tels projets, les entreprises ont besoin de maîtres d'ouvrage rompus à la communication entre biologie et autres sciences. La bio-informatique est le lieu où sont formés ces ingénieurs, habitués à la cette communication pluri-disciplinaire.

***Responsable de service bio-informatique* :** une part notable de ce métier est de participer à des réunions de direction où les besoins informatiques ne sont abordés que sous l'angle des utilisateurs biologistes ou commerciaux, de faire comprendre ce que l'informatique peut apporter dans ces problématiques, de traduire les besoins de l'entreprise pharmacologique, de dégager ce qu'il est possible ou non de faire, d'assurer la qualité des produits logiciels. Le responsable de service bio-informatique est capable de diriger des informaticiens « pur jus » et d'assurer l'utilité pour la biologie des projets développés.

***Architecte et administrateur de systèmes d'information* :** partant du savoir-faire et des données spécifiques d'une entreprise en pharmacologie, seul un ingénieur pluri-disciplinaire peut dégager les informations pertinentes, les structurer, les regrouper pour faciliter l'extraction de connaissances et finalement définir l'architecture adéquate du système d'information scientifique de l'entreprise.

***Responsable de conception et réalisation web, multi-média* :** seul un bio-informaticien peut séparer judicieusement l'intranet de l'extranet de telle sorte que les informations passées à l'extérieur prouve la compétence de l'entreprise mais ne permettent pas à une autre entreprise de s'approprier cette compétence. Il s'agit de concevoir un site web attractif tout en assurant la confidentialité.

***Chef de projet de calcul intensif* :** (fermes de calcul, simulations, extraction de connaissances, ...) ; demandez à un ingénieur non biologiste d'extraire des connaissances ou des prédictions à partir de données biologiques, et après un usage intensif de puissances de calcul énormes, les résultats que vous obtiendrez seront, au mieux, des évidences pour un biologiste. Seul un bio-informaticien peut identifier les approximations pertinentes d'un point de vue biologique qui rendront les calculs incommensurablement plus efficaces et les résultats valorisables.

***Responsable d'un service de biostatistiques* :** les statistiques reposent d'abord sur le choix des indicateurs mesurés, la compréhension critique des protocoles expérimentaux est donc un préalable à toute démarche biostatistique. De plus, il est bien connu que sans un oeil critique lors de l'interprétation des résultats, on peut faire dire ce que l'on veut aux statistiques. La connaissance à la fois du contexte biologique et des limites théoriques sont donc indispensables au professionnel.

***Architecte et administrateur systèmes et réseaux* :** dans des entreprises ou laboratoires de petite taille, le volume du parc informatique ne justifie pas, parfois, d'employer un informaticien à plein temps pour gérer les systèmes et réseaux. Cette tâche incombe alors aux bio-informaticiens de l'entreprise. Il s'agit d'une part d'assurer la sécurité des données, leur confidentialité, de protéger le réseau d'entreprise contre les attaques malvaillantes, et d'autre part de jouer un rôle de conseil...Cette activité est beaucoup moins complexe qu'il n'y paraît (un cours de 15h permet d'en acquérir les principes de base).

***Responsable de la robotisation* :** (protocole, bases de données, images...) ; des activités comme le screening font un large usage de robots. Leur pilotage inclut la gestion de la bibliothèque des produits utilisés par le robot, les protocoles définissant les actions successives du robot, l'analyse des résultats en série, *etc.* Cette robotisation nécessite une parfaite coordination avec les expérimentateurs et là encore seuls les bio-informaticiens de formation peuvent mettre en place ces protocoles expérimentaux d'une manière cohérente avec les objectifs biologiques.

***Consultant pluridisciplinaire* :** après quelques annnées d'expérience dans l'un (ou plusieurs) des métiers qui précédent, les ingénieurs qui souhaitent acquérir une plus grande indépendance de travail peuvent devenir consultants. La pénurie de ressources humaines dans les métiers proches de l'informatique laisse nécessairement une place à cette activité de consultance. Par ailleurs certaines PMI ou PME peuvent aussi faire appel à des consultants pour les aider à définir et/ou externaliser leurs besoins en bio-informatique.

***Métiers de la recherche* :** La plupart des laboratoires en biologie, qu'ils soient publics ou privés, ont identifié la bio-informatique comme un point clef de leur développement. D'une part, les gros laboratoires se munissent tous d'une service de bio-informatique généralement dirigé par un ingénieur de recherche, et d'autre part, le besoin d'une équipe de recherche de bio-informatique ouvre d'assez nombreux postes de chercheurs à des jeunes diplômés possédant un doctorat.

Les notes de cours et les feuilles de TD sont disponibles (avec un peu de retard par rapport au déroulement du cours) à l'adresse web suivante :

`http://www.i3s.unice.fr/~bernot/Enseignement/GB3_Python1/`

1 L'art de programmer

Utiliser un langage de programmation pour faire enchaîner à l'ordinateur, rapidement, des opérations sur des données n'est pas très difficile. Cependant, avec un style de programmation approximatif, on peut vite écrire des programmes incompréhensibles. Ce cours a pour objectif de vous apprendre à écrire des programmes *propres*, sans « bidouillages », et tellement logiquement écrits qu'on pourra les modifier plus tard, quand on aura même oublié comment ils marchent : en programmation, tout est dans **le *style* et l'*élégance***...

On va utiliser le langage *Python* comme support mais le choix du langage n'est pas très important car avec les méthodes de programmation « dans les règles de l'art » acquise dans ce cours, il sera facile de passer d'un langage à un autre. Les concepts de base d'une programmation bien menée sont les mêmes dans tous les langages.

2 Comment fonctionne un ordinateur, dans les grandes lignes

Avant d'apprendre les premiers éléments de programmation, cette partie du cours a pour objectif de démystifier les ordinateurs, de vous donner les principaux critères utiles qui font la qualité d'un ordinateur... un vendeur lambda ne pourra plus vous bluffer avec des termes techniques approximatifs `:-)`.

2.1 Le processeur et la mémoire

Le cœur d'un ordinateur est son *processeur* : c'est une sorte de « centrale de manipulations » qui effectue des transformations électriques rapides, correspondant à diverses manipulations de *symboles* codés par des « signaux électriques ». Il travaille de concert avec une *mémoire* qui alimente le processeur en *données* et en *instructions* à réaliser : sur le fond, la mémoire d'ordinateur stocke simplement une longue suite de symboles qui peuvent représenter aussi bien des données que des instructions.

L'élément de base des codages sus-mentionnés est l'information binaire : « le courant peut passer » ou au contraire « le courant ne peut pas passer ». Cette information minimale est appelée un *bit*. Un bit peut donc prendre deux valeurs, 0 ou 1, et est matériellement réalisé par un genre de transistor largement miniaturisé.

Un symbole est codé par une séquence de bits. Ainsi avec seulement 2 bits on peut déjà encoder 4 symboles différents.

> *Par exemple on pourrait décider que 00 code « A », symbole pour l'Adénine, que 01 code « T », symbole de la Thymine, que 10 code « G », symbole de la Guanine, et que 11 code « C », symbole de la Cytosine.*

Avec 3 bits, on peut encoder 8 symboles différents (4 commençant par le bit 0 et 4 commençant par le bit 1), *etc.* En pratique, on compte les volumes d'information en *octets*. Un octet (« byte » en anglais) est formé de 8 bits et peut donc encoder $2^8 = 256$ symboles différents. Le code le plus utilisé est le code ASCII, qui encode sur un octet (presque) toutes les lettres (majuscules, minuscules), les ponctuations, l'espace, des caractères dits de contrôle, *etc.* Pour encoder les lettres accentuées et les lettres spécifiques à certaines langues, il faut faire appel à un autre code qui peut utiliser 2 octets pour ces lettres particulières ; le plus courant est le code UTF8.

L'électronique n'a aucune difficulté à manipuler les bits : éteindre un bit, l'allumer, inverser sa valeur, tester sa valeur... Il existe ainsi un ensemble (limité) d'*instructions* qui manipulent les bits (souvent par groupes de 8 bits donc par octet, ou même par groupes de plusieurs octets). Chaque instruction est elle même codée par une séquence de bits, exactement comme les symboles.

Les instructions successives à effectuer sont placées en des endroits déterminés de la mémoire et sont lues les unes après les autres, ce qui conduit à enchaîner des commandes et finalement à faire des transformations aussi complexes que l'on veut de la mémoire. La mémoire sert donc à la fois à stocker le programme des instructions successives à effectuer et à stocker les données que ce programme manipule. Pour ce faire le proceseur possède une zone dans laquelle il note la place de la mémoire où se trouve la prochaine instruction à effectuer, une zone dans laquelle il a recopié

l'instruction en cours, et quelques zones qui servent de la même façon à manipuler les données. Ces zones sont appelées des registres.

Maintenant que l'on a vu comment transitent les instructions successives à effectuer dans le processeur, parlons des données : il ne servirait à rien de faire tant de manipulations si elles ne sortaient jamais des registres du processeur. Pour cela il y a des instructions de stockage des registres du processeur vers la mémoire, et pour alimenter le processeur en données, il existe inversement des instructions de chargement des données qui peuvent copier n'importe quel emplacement de la mémoire vers les registres de données. Ainsi les suites d'instructions commencent souvent par charger certaines places de la mémoires dans les registres, puis font divers transformations de ces symboles, et enfin stockent de nouveau en mémoire les résultats présents dans les registres.

2.2 Les périphériques

Ainsi, on voit que si l'on conçoit un programme judicieux d'instructions successives, on peut modifier à volonté l'état de la mémoire pour lui faire stocker des résultats de manipulations de symboles, aussi complexes et sophistiquées que l'on veut. La question qui se pose maintenant est d'exploiter cette mémoire en la montrant à l'extérieur sous une forme compréhensible pour l'Homme, ou réexploitable ultérieurement. C'est le rôles des *périphériques* :

- Avec ce que l'on a vu jusqu'à maintenant, s'il y a une coupure de courant alors tout est perdu car les « mini-transistors » ne conservent pas leur état (0 ou 1) sans courant. Donc certains périphériques sont des mémoires « de masse » comme les disques durs, les disquettes, les CDrom, DVDrom, Blu-ray *etc.* ou certaines mémoires « flash » (=clefs USB) ayant une durée raisonnable de conservation des données.
- Il faut également pouvoir intervenir et entrer les données et les ordres dans l'ordinateur par exemple *via* un clavier, une souris, une manette de jeu, des instruments de mesure, *etc.* Ces périphériques « d'entrées » agissent en modifiant certaines parties de mémoires (indiquant les touches enfoncées, les déplacements de souris ou de doigts sur un écran tactile, *etc*) et les programmes qui gèrent le système informatique « surveillent » tout simplement régulièrement ces mémoires particulières.
- Accéder aux résultats et les visualiser est également nécessaire ; l'ordinateur doit pouvoir montrer à l'utilisateur les résultats des manipulations symboliques par exemple via une carte graphique et un écran, une imprimante ou divers appareillages de réalité virtuelle... De la même façon le rôle de ces périphériques « de sortie » (comme la carte graphique) est simplement de transcrire (par exemple en pixels de couleur) le contenu de « sa » zone de mémoire ; zone de mémoire que les programmes peuvent naturellement modifier à volonté.
- Plus généralement, entrées et sorties combinées donnent lieu à des périphériques « de communication » qui permettent non seulement de communiquer avec l'Homme mais aussi avec d'autres ordinateurs par exemple *via* des réseaux.

3 Quelques unités de capacité d'information

- Un *bit* est une valeur logique (vrai/faux codé en 0/1).
- Un *octet* (en anglais : byte) est une suite de 8 bits ; il permet de coder par exemple des entiers de 0 à 255, ou des caractères, ou des instructions comme déjà mentionné.
- Un *Ko* se prononce un *kilo-octet* (en anglais : Kb, Kilo-byte) ; c'est une suite de 1024 octets (pas exactement 1000 parce que 1024 est une puissance de 2, ce qui facilite l'adressage sur ordinateur).
- Un *Mo* se prononce *Méga-octet* (en anglais : Mb, Mega-byte) ; c'est une suite de 1024 Ko.
- Un *Go* se prononce *Giga-octet* (en anglais : Gb, Giga-byte) ; suite de 1024 Mo.
- Un *Tera-octet* est une suite de 1024 Go, puis viennent les *Peta-octet*, *Exa-octet*, *etc.*

Actuellement les valeurs significatives lorsque vous achetez un ordinateur sont :

- La vitesse à laquelle se succèdent les opérations élémentaires faites par le ou les processeurs de l'ordinateur : de 1 à 5 Giga-opérations par seconde. Il s'agit donc d'une *fréquence* exprimée en Hertz (environ 3,5GHz).
- Le nombre de processeurs mis ensemble, c'est-à-dire combien d'opérations élémentaires se font en même temps : de 1 à 4 (mono-core, dual-core, quad-core).
- La taille de la mémoire : souvent de 2 à 16 Giga-octets.
- La taille du disque dur dans lequel seront stockés vos programmes (applications, outils bureautiques, lecteurs divers, jeux, *etc.*) et vos données (textes, images, musique, films, bases de données, *etc.*) : de 200 Giga-octets à 4 Tera-octets.
- La résolution de l'écran : de 800 à 3000 pixels de large et de 700 à 2000 pixels de haut, soit de 600 Kilo-pixels à 6 Méga-pixels environ.

4 Les langages de programmation

Ce qu'on vient de voir sur la structure d'un ordinateur donne déjà des capacités de programmation importantes : il suffit de mettre en mémoire une suite d'instructions à effectuer, et le processeur les chargera et les effectuera les unes après les autres. C'est ce qu'on appelle le *langage machine*. Comme son nom l'indique, c'est un langage très efficace pour une machine... mais il est très indigeste à lire (et à écrire) pour un être humain. Plutôt que d'écrire du langage machine, l'humain préfère écrire des ordres à l'ordinateur dans un langage plus évolué (par exemple Python).

C'est là qu'intervient un raisonnement astucieux :

1. Un programme écrit par un humain en Python est finalement un texte, c'est-à-dire une suite de caractères qui peut donc être stockée dans la mémoire de l'ordinateur.
2. Les programmes en langage machine sont des suites d'instructions qui doivent aussi être stockées en mémoire.
3. *Or* un programme en langage machine est finalement un processus qui transforme des données en mémoire en d'autres données en mémoire, quelles qu'elles soient. Un programme en langage machine peut donc considérer un programme écrit en Python comme des données, de même qu'il peut considérer un autre programme en langage machine comme des données.
4. *Donc*, si quelques spécialistes se chargent de la corvée d'écrire un programme en langage machine qui transforme :
   — un (texte de) programme en python
   — en une suite d'instructions en langage machine
   alors nous pouvons écrire un texte en Python, laisser le programme des spécialistes en faire un programme en langage machine, et faire tourner le résultat.

Cette technique peut s'appliquer de différentes façons que nous ne détaillerons pas ici. Elle est appelée selon les cas la *compilation* ou l'*interprétation* du langage (ici Python). Ceci permet d'avoir l'impression que c'est le programme écrit en Python qui « tourne » directement sur l'ordinateur.

> *D'une certaine façon, l'interprétation d'un langage de programmation joue pour l'informatique un rôle inverse de la transcription et la traduction pour la biologie moléculaire : la transcription et la traduction permettent de construire des structures de plus haut niveau à partir du « langage machine » qu'est le génome, alors que l'interprétation produit du langage machine à partir de textes bien structurés.*

Grâce à cette technique classique, la notion de *langage de programmation* devient plus large qu'une simple suite d'instructions données à l'ordinateur.

Un langage de programmation est un « vocabulaire » restreint et des règles de formation de « phrases » très strictes pour donner des instructions à un ordinateur. Le *moins* par rapport au français ou aux mathématiques reste malgré tout sa pauvreté, mais le *plus* est qu'aucune « phrase » (=*expression*) n'est ambigüe : il n'existe pas plurieurs interprétations possibles.

On peut alors :
— regrouper puis abstraire un grand nombre de données élémentaires (nombres, caractères...) pour caractériser une *structure de données* abstraite (`protéine` = suite de ses acides aminés *et* ses conformations possibles *et* ses sites actifs en fonction de conditions, *etc*).
— regrouper des suites de commandes élémentaires (additions, multiplications, statistiques, classifications, décisions...) pour organiser le *contrôle du programme* et le rendre plus lisible et logique (déterminer si deux protéines ont de l'`affinité` = inventorier les conditions du compartiment qui les contient, calculer la conformation la plus probable, inventorier les sites actifs qui en résultent, comparer deux à deux si un domaine d'une protéine a de l'affinité avec un domaine de l'autre protéine, *etc*).

Donc : ce qui définit les langages de programmation, c'est
— la façon de représenter symboliquement les **structures de données** (e.g. protéine)
— et la façon de gérer le **contrôle** des programmes (e.g. que faire et dans quel ordre pour résoudre une question d'affinité de protéines).

5 Un exemple en Python

On reviendra plus précisément sur chacune des notions utilisées ici. Il s'agit simplement pour l'instant de *voir* à quoi ressemble un programme et son utilisation.

```
>>> borne = 100
>>> def evalue (n) :
...   if n < (borne / 2) :
```

```
...     print("petit")
...   elif n < borne :
...     print("moyen")
...   else :
...     print("grand")
...
>>> evalue (70)
moyen
>>> evalue (150)
grand
>>> borne = 50
>>> evalue (70)
grand
```

Selon le système d'exploitation avec lequel on travaille, la présentation peut différer (les « >>> » ou les « ... ») mais en revanche, ce qu'on tape au clavier et les réponses de l'ordinateur sont *toujours* identiques.

- La première ligne « `borne = 100` » est une *affectation* de variable. Elle a pour effet qu'à partir de cette ligne, il est équivalent d'écrire `borne` ou d'écrire `100`. On dit que `borne` est une *variable* et que sa *valeur* est 100. L'avantage de cette ligne est double :
  - Partout où l'on a utilisé `borne` au lieu de `100`, il sera facile de « changer d'avis » et de considérer que finalement la borne ne vaut que 50. Il suffit de faire une nouvelle affectation *sans avoir besoin de chercher partout* où se trouvait le nombre 100 et le remplacer par 50. Toutes les valeurs qui peuvent changer un jour (taux de TVA, mesure de diverses quantités, *etc*) doivent donc être mises dans des variables.
  - Après des affectations de variables, les programmes sont plus lisibles car les noms de variables permettent de leur donner une signification. On peut écrire des programmes compréhensibles où les valeurs sont remplacées par des noms parlants, qui aident à la compréhension. L'idéal serait de pouvoir programme presque comme en français « si l'entier `n` est plus petit que la moitié de la borne alors il est considéré comme petit, sinon s'il reste plus petit que la borne alors il est moyen, sinon il est grand. »
- Le mot clef `def` ressemble lui aussi à une affectation en ce sens qu'il donne un nom à quelquechose. Il s'agit de donner un nom à un petit programme (`evalue`) qui fait des manipulations symboliques en fonction d'une valeur qu'on lui donne (`n`). Ici, le nom `evalue`, chaque fois qu'il sera appelé, lancera les comparaisons du nombre qu'on lui donne entre parenthèses avec la `borne` et/ou sa moitié, et il imprimera à l'écran le résultat qui en découle (grand, moyen ou petit).
  La construction `if..elif..else..` n'a rien de difficile :
  - `elif..` est une contraction de « else if.. »
  - juste derrière le « `if` » et avant le « : » on écrit la condition qui, si elle est vraie, conduit à effectuer les commandes qui suivent, et si par contre elle est fausse, conduit à effectuer les commandes qui suivent le « `else` ». C'est logique.
- Enfin la commande `print` imprime à l'écran ce qui est écrit entre guillemets.

On dit que la « fonction `evalue` prend `n` en entrée » ou encore « en argument », pour dire que le résultat que fournit `evalue` dépend de la valeur « mise à la place de `n` » entre parenthèses. On remarque que ce résultat dépend aussi de `borne`, mais cette fois de manière implicite (c'est-à-dire que `borne` n'est pas un argument de `evalue`).

Puisqu'un langage de programmation est défini par ses *structures de données* et par son *contrôle*, il suffit de connaître les parties les plus utiles de ces deux aspects en Python pour savoir programmer en Python. Mieux : sur le fond, les principales structures de données et les principales primitives de contrôle *sont les mêmes* pour tous les langages dits impératifs (qui eux-mêmes représentent la quasi-totalité des langages industriels). Les seules variations d'un langage à l'autre sont les choix des mots clefs : par exemple certains langages utilisent « `function` », « `procedure` » ou encore « `let` » au lieu de « `def` ». C'est dire à quel point un cours de programmation peut être universel, sous réserve de ne pas se noyer dans les particularités de détail du langage choisi.

Nous allons commencer par les structures de données les plus simples.

6 Les structures de données

Une structure de données est définie par quatre choses :

1. *Un type* qui est simplement un nom permettant de classifier les expressions syntaxiques relevant de cette structure de donnée. Par exemple, `"grand"`, avec ses guillemets autour, est une donnée de type `string` (chaîne de caractères en français) et `"petit"` et `"moyen"` sont deux autres données de type `string` elles aussi.

2. *Un ensemble* qui définit avec précision quelles sont les *valeurs pertinentes* de ce type. Par exemple l'ensemble de toutes les suites de caractères, de n'importe quelle longueur, sachant qu'un caractère peut aussi bien être une lettre qu'un chiffre, une ponctuation ou un caractère dit « de contrôle ».
3. *La liste des opérations* que l'ordinateur peut utiliser pour effectuer des « calculs » sur les valeurs précédentes. Cela comprend le nom de l'opération, par exemple `print`, et comment l'utiliser, c'est-à-dire quels *arguments* elle accepte et la nature du résultat. Dans l'exemple précédent, on a vu par exemple que `print` accepte un argument de type `string` (en réalité il en accepte aussi d'autres, on verra ça plus tard) et a pour résultat d'écrire à l'écran cette chaîne de caractère (sans les guillemets).
4. *La sémantique des opérations* c'est-à-dire une description précise du résultat fourni en fonction des arguments.

Il existe en Python une commande `type`, qui prend en argument une valeur ou une variable quelconque et fournit comme résultat le type de cette valeur.

7 Les booléens

Il s'agit d'un ensemble de seulement 2 données pertinentes, dites *valeurs de vérité* : {`True`,`False`}. Le type est appelé `bool`, du nom de George Boole [1815-1864] : mathématicien anglais qui s'est intéressé aux propriétés algébriques des valeurs de vérité.

Opérations qui travaillent dessus :

Opération	Entrée	Sortie
`not`	`bool`	`bool`
`and`	`bool`×`bool`	`bool`
`or`	`bool`×`bool`	`bool`

De manière similaire aux tables de multiplications, on écrit facilement les tables de ces fonctions puisque le type est fini (et petit).

Exemples de calculs sur les booléens :

```
>>> print(True)
True
>>> print(False)
False
>>> print(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> print(True and False)
False
>>> type(True)
<type 'bool'>
>>> True and False
False
>>> True and
  File "<stdin>", line 1
    True and
            ^
SyntaxError: invalid syntax
```

Comme on le voit, l'opération `type` écrit bien le type d'une donnée.

8 Les entiers relatifs

Théoriquement le type `int` correspond à l'ensemble $\mathbb{Z}$ des mathématiques. En fait il est borné en Python, avec borne dépendante de la machine, mais assez grande pour ignorer ce fait ici. (`int` comme « integers »).

Les opérations qui travaillent dessus :

Opérations	Entrée	Sortie
`+` `-` `*` `//` `%` `**`	`int`×`int`	`int`
`==` `<` `>` `<=` `>=` `!=`	`int`×`int`	`bool`

Exemples de calculs sur les entiers :

```
>>> type(46)
<type 'int'>
>>> 5 * 7
35
>>> print(5 * 7)
35
>>> 5 // 2
2
>>> 5 % 2
1
>>> 5 ** 3
125
```

On remarque que la division `//` est *euclidienne* (pas de virgule dans le résultat) et que `%` donne le reste de la division euclidienne.

Pour les comparaisons :

```
>>> 5 < 3
False
>>> 5 == 3 + 2
True
>>> 5 = 3 + 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> 5 <= 5
True
```

Noter la différence entre « `==` » (qui effectue une comparaison et fournit un booléen en résultat) et « `=` » qui est une affectation (et doit avoir un nom de variable à sa gauche et une valeur à sa droite). On remarque aussi que les opérations de calcul `+ - * // % **` sont *prioritaires* sur les opérations de comparaison `== < > <= >= !=` car sinon « `5 == 3 + 2` » n'aurait pas eu plus de sens que « `False + 2` » (on n'additionne pas un booléen et un entier).

9 Les nombres réels

On les baptisent « les nombres flottants » pour des raisons historiques.

Le type `float` représente l'ensemble des nombres réels, avec cependant une précision limitée par la machine, mais les erreurs seront négligeables dans le cadre de ce cours. Par abus d'approximations on considère donc que `float` correspond à l'ensemble $\mathbb{R}$.

On dispose des opérations suivantes :

Opérations	Entrée	Sortie
`+ - * / **`	`float`×`float`	`float`
`int`	`float`	`int`
`float`	`int`	`float`
`== < > <= >= !=`	`float`×`float`	`bool`

Exemples d'utilisation :

```
>>> type(46.8)
<type 'float'>
>>> 5.0 / 2.0
2.5
>>> 7.5 * 2
15.0
>>> int(7.5 * 2)
15
```

```
>>> int(7.75)
7
>>> float(3)
3.0
```

Noter la différence entre `/` et `//` : depuis la version 3 de Python, `/` est la division exacte sur les réels uniquement, et `//` est la division euclidienne sur les entiers relatifs uniquement.

10 Les chaînes de caractères

Il existe 256 « *caractères* » qui couvrent à peu près tout ce que l'on peut taper au clavier en une fois, en utilisant les touches de nombres, lettres ou ponctuations, éventuellement en combinaison avec la touche *majuscule* (*shift* en anglais) ou bien la touche *contrôle*. Une *chaîne de caractères*, comme son nom l'indique, est une suite finie de caractères.

Le type des chaînes de caractères est généralement appelé `string` mais en Python il est appelé de manière abrégée `str`. Pour produire une chaîne de caractères, il suffit de l'écrire entre guillemets, simple ou doubles :

```
>>> "toto"
'toto'
>>> 'toto'
'toto'
>>> toto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'toto' is not defined
>>> toto = 56
>>> toto
56
```

Les guillemets sont nécessaires car sinon, Python interprète la chaîne de caractères comme le nom d'une variable, et la commande est alors interprétée par Python en « donner la valeur de `toto` ».

L'opération la plus courante sur les chaînes de caractères est la concaténation, notée avec un « + ». Les autres opérations fréquentes sont la longueur (nombre de caractères) notée `len` et les comparaisons (ordre du dictionnaire).

```
>>> "toto" + "tutu"
'tototutu'
>>> len("toto")
4
>>> "toto" < "tutu"
True
>>> "ab" < "aab"
False
>>> "a b" == "ab"
False
```

Noter l'absence d'espace entre `"toto"` et `"tutu"` après concaténation.

Opérations	Entrée	Sortie
`+`	`str`×`str`	`str`
`len`	`str`	`int`
`==` `<` `>` `<=` `>=` `!=`	`str`×`str`	`bool`

11 Opération de substitution de chaînes de caractères

La structure de données des chaînes de caractères a encore une opération qui mérite une section d'explication à elle seule : la *substitution*, notée « `%` ».

Si la chaîne de caractères s_0 contient « `%s` » et si s_1 est une autre chaîne de caractères, alors s_0 `%` s_1 est la chaîne obtenue en remplaçant dans s_0 les deux caractères `%s` par la chaîne s_1. Par exemple :

```
>>> "bonjour %s ; il faut beau aujourd'hui." % "Pierre Dupond"
"bonjour Pierre Dupond ; il faut beau aujourd'hui."
```

Plus généralement, s'il y a plusieurs « `%s` », il faut mettre entre parenthèses autant de chaînes $(s_1, \ldots, s_n)$ après l'opération `%`. Par exemple :

```
>>> nom = "Dupond"
>>> prenom="Pierre"
>>> genre = "homme"
>>> "Ce %s s'appelle %s et c'est un %s" % (prenom,nom,genre)
"Ce Pierre s'appelle Dupond et c'est un homme"
```

Si l'on veut mettre un entier relatif, on utilise « `%d` » ou « `%i` » au lieu de « `%s` » (`d` comme décimal ou `i` comme integer, et `s` comme string).

```
>>> age = 41
>>> "%s %s a %d ans." % (prenom,nom,age)
'Pierre Dupond a 41 ans.'
```

Pour un nombre réel, c'est « `%f` » (`f` comme float).

```
>>> "%s %s mesure %fm." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.850000m.'
```

et si l'on veut imposer le nombre de chiffres après la virgule :

```
>>> "%s %s mesure %.2fm." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.85m.'
```

Les différents encodages des substitutions présentés ici sont les plus utiles. Il y en a d'autres, plus rarement utilisés, qu'on trouve aisément dans tous les manuels.

12 Des conversions de type (cast)

Connaître type d'une valeur ou d'un calcul est *primordial* lorsque l'on programme.
Par exemple `print("solde="+10+"euros")` est mal typé et constitue une erreur. En effet, on veut ici utiliser l'opération « `+` » qui effectue la concaténation des *chaînes de caractères*, par conséquent il faut lui fournir des *chaînes de caractères* en arguments : `print("solde="+"10"+"euros")`.

La valeur notée `"10"` est une chaîne de caractères constituée de deux caractères consécutifs `"1"` suivi de `"0"` il ne s'agit en aucun cas d'un nombre entier. La chaîne `"10"` est de type `str` et le nombre `10` est de type `int` : ils ne sont pas du tout encodés pareil dans la machine.

Pour passer de l'un à l'autre, il faut une fonction, qui est fournie par Python. Plus généralement, les fonctions qui permettent d'effectuer une conversion d'un type à l'autre de manière « naturelle » sont souvent appelées des *cast* :

- La fonction `str` transforme son argument en chaîne de caractère chaque fois que le type de l"argument est sufissament simple pour le faire sans ambiguïté.
- La fonction `int` transforme de même son argument en entier relatif chaque fois que possible.
- La fonction `float` transforme de même son argument en réel chaque fois que possible.
- La fonction `bool` existe aussi mais n'a aucun intérêt du fait des opérations de comparaisons, bien plus claires.

```
>>> str(10)
'10'
```

```
>>> str(True)
'True'
>>> str(15.33)
'15.33'
>>> int("10")
10
>>> int("toto")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'toto'
>>> int(12.3)
12
>>> int(12.9)
12
>>> int("12.9")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.9'
>>> float("12.6")
12.6
>>> float(12)
12.0
```

13 Les expressions conditionnelles

Abandonnons pour quelques temps la description des structures de données classiques en programmation pour étudier deux éléments cruciaux du contrôle : les expressions conditionnelles et les définitions de fonctions ou de procédures.

Une expression conditionnelle « de base » est de la forme *SI condition ALORS action1 SINON action2*. En python cela donne par exemple, comme on l'a déjà vu :

```
>>> if "a b" == "ab" :
...     print("L'espace ne compte pas.")
... else :
...     print("L'espace compte.")
...
L'espace compte.
```

Dans le cas où l'on ne souhaite rien faire lorsque la condition est fausse, on peut omettre la partie `else`.

```
>>> if len("abc") != 3 :
...     print("YA UN PROBLEME !")
...
>>>
```

Enfin il arrive qu'on ait des « cascades » d'expressions conditionnelles et dans ce cas le mot-clef `elif` est une contraction de `else if` (exemple dans la section suivante).

1 Les définitions de fonctions

Lorsque certains calculs sont assez bien identifiés pour porter un nom, on a intérêt à leur donner un nom, ce qui simplifiera leur usage ultérieurement. Pour cela on définit des *fonctions*. Par exemple :

```
>>> def complementaire (n) :
...     if n == 'A' :
...         return 'T'
...     elif n == 'T' :
...         return 'A'
...     elif n == 'G' :
...         return 'C'
...     elif n == 'C' :
...         return 'G'
...     else :
...         return 'N'
...
>>> complementaire ('G')
'C'
>>> complementaire ('N')
'N'
>>> complementaire(4)
'N'  (mais cela devrait retourner une erreur de typage !)
>>> complementaire(complementaire('A'))
'A'
```

Le mot clef `def` signifie que toutes les lignes qui sont *sous sa portée* constituent une définition de la fonction `complémentaire`. Le « (n) » entre parenthèses signifie que par la suite, on devra donner en entrée de cette fonction un seul argument, et que cet argument remplacera toutes les occurrences de `n` dans la définition pour calculer le résultat de la fonction `complementaire`.

- Les 10 lignes qui suivent la ligne « `def complementaire (n) :` » sont *sous la portée* de `def` parce qu'elles ont un décalage de marge par rapport à `def`. La ligne vide qui suit ces 10 lignes indique que la marge revient à zéro, donc la fin de la portée de `def`.
- De la même façon, « `return 'T'` » est sous la portée de « `if n == 'A' :` » à cause d'un second décalage de marge, et le fait que le `elif` qui suit revienne au premier décalage de marge signifie la fin de la portée de « `if n == 'A' :` ».
- Si l'on veut programmer une fonction avec plusieurs entrées, il suffit de les séparer avec des virgules à l'intérieur de la parenthèse.

Le mot-clef `return` indique ce que la fonction fournit comme résultat. Dans chacun des cas, il faut donc n'avoir qu'un seul `return` possible. On peut alors réutiliser la fonction à toutes les sauces dans d'autres fonctions, exactement comme si le langage Python la fournissait parmi ses opérations :

```
>>> def nucleotide(n) :
...     return complementaire(n) != 'N'
...
>>> nucleotide ('A')
True
>>> nucleotide ('B')
False
>>> def transcription (adn) :
...     if adn == 'A' :
...         return 'U'
...     else :
...         return complementaire(adn)
...
>>> transcription ('A')
```

```
'U'
>>> transcription ('B')
'N'
>>> transcription ('G')
'C'
```

Noter qu'*une variable indique une place* : `adn`, aurait pu être remplacé par `n` ou `glop`...

2 Les définitions de procédures, dialogues avec un utilisateur

Il est possible de définir des « fonctions » qui ne fournissent aucun résultat ! On appelle cela des *procédures*. Naturellement cela est d'un intérêt moindre et on préfèrera utiliser des fonctions chaque fois que possible.

Pour écrire une procédure, il suffit de ne jamais utiliser `return` dans la programmation. Par exemple, on peut se contenter d'écrire à l'écran le résultat du calcul, sans le fournir pour autant comme résultat « déclaré ».

```
>>> def trans (n) :
...     if n == 'A' :
...         print('U')
...     else :
...         print(complementaire(n))
...
>>> trans ('A')
U
>>> trans ('G')
C
```

La différence ne saute pas aux yeux, si ce n'est que les guillemets n'apparaissent pas dans le « résultat » lors des appels de `trans`. En fait, `trans` imprime lui même `U` ou `C` à l'écran (commande `print`), alors que pour `transcription`, c'est le langage Python qui se chargeait d'écrire le résultat. La différence majeure, c'est que `trans` ne retourne aucun résultat. En voici la preuve :

```
>>> len(transcription('A'))
1
>>> type(transcription('A'))
<type 'str'>
>>> len(trans('A'))
U
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'NoneType' has no len()
>>> type(trans('A'))
U
<type 'NoneType'>
```

On remarque que `trans` est appelé dans chacune des deux commandes précédentes parce qu'il imprime « bêtement » à l'écran son résultat. Cependant il ne le fournit pas à la fonction qui l'appelle, donc `len` qui attend une chaîne de caractères, produit une erreur, et le « résultat » de `trans` est sans type.

Ainsi donc, on n'utilisera les procédures que pour imprimer divers résultats finaux à l'écran. Plus généralement, les procédures ne devraient être utilisées que pour gérer les dialogues avec les utilisateurs (IHM = Interface Homme-Machine).

Lorsque l'on veut « récupérer » des informations en les demandant à un utilisateur, on utilise une fonction appelée `input`. La « fonction » `input` prend en entrée une chaîne de caractères et fournit en sortie une chaîne de caractère. Elle écrit à l'écran la chaîne qu'on lui donne en argument et attend que l'utilisateur tape une ligne (terminée par un retour chariot) ; le résultat de la « fonction » est alors la chaîne de caractère qu'a tapée l'utilisateur et est donc toujours de type `str`.

```
>>> def bonjour () :
...     prenom = input("Quel est votre prénom ? : ")
...     nom = input("Quel est votre nom de famille ? : ")
...     print("Bonjour %s %s ! bonne journée." % (prenom,nom))
...
>>> bonjour()
Quel est votre prénom ? : Albert
Quel est votre nom de famille ? : Durand
Bonjour Albert Durand ! bonne journée.
```

Il est cependant souvent utile de demander à l'utilisateur une valeur d'un autre type que `str`. Il faut alors gérer « à la main » la conversion de `str` vers le type que l'on souhaite.

Rappel : la fonction `int` convertit en entier (si possible) l'argument qu'on lui donne. Ainsi les nombre flottants (type `float`) sont tronqués par la fonction `int`. Cette fonction `int` marche aussi pour les chaînes de caractères *ne contenant que des chiffres* (au passage, `str` fait donc l'inverse).

```
>>> int(12.7)
12
>>> int("12")
12
>>> int("12.7")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.7'
>>> str(126)
'126'
```

Si l'on veut ignorer les caractères qui ne sont pas des chiffres, il faut le programmer soi-même :

```
>>> def intsouple (s) :
...     r = 0
...     for c in s :
...         if c >= '0' and c <= '9' :
...             r = 10 * r + int(c)
...         #sinon on ignore le caractère
...     return r
...
>>> intsouple ("to3to4glop0")
340
```

et ainsi par exemple :

```
>>> def askint () :
...     s = input("entrez un entier positif : ")
...     n = intsouple(s)
...     if str(n) != s :
...         print("Vous tapez à coté des touches !")
...     return n
...
>>> 2 * askint()
entrez un entier positif : 45
90
>>> 2 * askint()
entrez un entier positif : glop9u0
Vous tapez à coté des touches !
180
```

3 Variables locales et variables globales

Lorsqu'une fonction ou une procédure modifie une variable, elle est toujours « locale » à cette fonction ou procédure. Cela signifie que sitôt que l'on est sorti de la fonction, on a « oublié » la variable.

```
>>> def triple (n) :
...     resultat = n * 3
...     return resultat
...
>>> triple (4)
12
>>> resultat
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'resultat' is not defined
```

Mieux : si la variable était définie préalablement, elle n'est pas modifiée par la fonction :

```
>>> resultat = 128
>>> triple(2)
6
>>> resultat
128
```

En revanche, une variable *globale* qui n'est pas modifiée par la fonction est visible dans la fonction :

```
>>> def teste (n) :
...     if n > resultat :
...         print("plus grand")
...     else :
...         print("plus petit")
...
>>> teste (4)
plus petit
>>> teste (200)
plus grand
```

MAIS, par défaut, en Python une variable ne peut pas être *à la fois* globale et locale :

```
>>> def ajuste (n) :
...     if n > resultat :
...         resultat = n
...
>>> ajuste (200)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in ajuste
UnboundLocalError: local variable 'resultat' referenced before assignment
```

Moralité : par défaut en Python, une fonction ou une procédure ne peut pas modifier une variable globale. Les autres langages l'autorisent par défaut mais c'est de toute façon un style de programmation dangeureux, puisque l'état des variables serait alors modifié implicitement. Mieux vaut écrire :

```
>>> def ajuste (n) :
...     if n > resultat :
...         return n
...     else :
...         return resultat
```

```
...
>>> resultat = ajuste(200)
>>> resultat
200
```

Si l'on tient absolument à modifier une variable globale dans une fonction, il faut utiliser une *déclaration* dans la fonction qui change le comportement par défaut pour la variable en question. On utilise alors le mot `global` :

```
>>> nbUsage=0
>>> def triple(n) :
...   global nbUsage
...   resultat = 3 * n
...   nbUsage = nbUsage + 1
...   return resultat
...
>>> nbUsage
0
>>> triple(2)
6
>>> nbUsage
1
>>> triple(6)
18
>>> nbUsage
2
```

1 Compléments sur les chaînes de caractères, et `while`

Revenons un peu aux structures de données et commençons par des moyens complémentaires d'accéder aux caractères dans une chaîne de caractères.

Il est souvent pratique de pouvoir extraire le n-ième caractère d'une chaîne de caractères. Si s est une chaîne de caractères, alors l'opération d'*accès direct* `s[i]` fournit le (`i`+1)-ième caractère de la liste (c'est à dire que le premier caractère est en fait numéroté 0, le deuxième est numéroté 1, etc).

```
>>> "abcdef"[0]
'a'
>>> "abcdef"[1]
'b'
>>> "abcdef"[2]
'c'
>>> "abcdef"[5]
'f'
>>> "abcdef"[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

On peut alors par exemple calculer le brin complémentaire d'une portion de génome (et plus seulement d'un seul caractère à la fois comme le fait la fonction `complementaire` programmée au cours précédent) en parcourant le brin d'origine « à l'envers » :

```
>>> def brinCompl (c) :
...     resultat = ""
...     i = len(c)
...     while i > 0 :
...         i = i - 1
...         resultat = resultat + complementaire(c[i])
...     return resultat
...
>>> brinCompl ("AAATCCGT")
'ACGGATTT'
```

À cette occasion, on rencontre une nouvelle commande de contrôle : `while`, qui s'utilise syntaxiquement comme un `if` sans `else`, mais dont le bloc de programme est exécuté et ré-exécuté tant que la condition reste vraie.

ATTENTION : si l'on gère mal le programme, une instruction `while` peut boucler indéfiniment (par exemple si l'on oublie `i = i - 1`). La boucle ci-dessus met donc deux variables locales à jour : l'indice `i` des caractères que l'on traite les uns après les autres et la chaîne de caractères, construite pas à pas, qui fournira le résultat final.

Encore une opération qui est parfois utile sur les chaînes de caractères : s_0 `in` s prend en entrée deux chaînes de caractères et retourne un booléen qui indique si s_0 est une sous-chaîne (consécutive) de s (test d'appartenance).

```
>>> "to" in "tatetitotu"
True
>>> "io" in "tatetitotu"
False
```

Pour calculer le nombre de voyelles d'une chaîne de caractères, on peut alors programmer :

```
>>> def nbvoyelles (s) :
...     n = 0
```

```
...     i = 0
...     while i < len(s) :
...         if s[i] in "aeiouyAEIOUY" :
...             n = n + 1
...         i = i + 1
...     return n
...
>>> nbvoyelles("toto")
2
```

(cette fois on a parcouru la chaîne dans l'ordre).

Terminons sur les chaînes de caractères en signalant qu'il est possible d'extraire plus de un caractère à la fois avec la forme suivante : *s*`[i:j]`. Cette forme fournit la chaîne de caractères commençant à l'indice `i` et se terminant à `j-1` (et non pas `j`, ce serait trop simple...).

```
>>> "abcdef"[2:5]
'cde'
>>> "abcdef"[2:2]
''
```

Cela permet d'extraire n'importe quelle sous-chaîne d'une chaîne de caractères.

2 Parcours de chaîne de caractères avec `for`

La primitive `for` permet de parcourir une chaîne de caractères du début à la fin en énumérant les caractères les uns après les autres, de la gauche vers la droite.

Reprenons l'exemple de `brinCompl` et commençons par remarquer que cette autre version, qui utilise toujours un `while`, est équivalente à la précédente parce qu'elle concatène à gauche de `resultat` :

```
>>> def brinCompl (c) :
...     resultat = ""
...     i = 0
...     while i < len(c) :
...         resultat = complementaire(c[i]) + resultat
...         i = i + 1
...     return resultat
```

Cette nouvelle version est d'un certain point de vue plus simple que la première version car elle parcourt la chaîne de caractères « dans le sens normal » de gauche à droite, ainsi, l'indice `i` part de 0 et augmente de 1 en 1 jusqu'à la longueur de `c` moins 1.

Ce qui reste pénible dans cette version, c'est qu'on passse du temps à réfléchir comment gérer proprement l'indice `i` : ne pas oublier d'initialiser l'indice `i` ; faut-il commencer à 0 ou à 1 ? faut-il finir les tours de boucle `while` avec `i=len(c)` ou `i=len(c)-1` ? faut-il mettre « inférieur ou égal » ou un « inférieur strict » ?

Dans les cas où l'on doit parcourir la chaîne de caractère *du début à la fin* en prenant les caractères les uns après les autres, on peut utiliser la primitive « `for` » qui évite de gérer l'indice `i`. En effet, la seule chose qui nous intéresse dans le programme précédent, ce n'est pas vraiment la valeur de `i`, c'est le *caractère* `c[i]` de la chaîne `c`. Une boucle `for` permet justement de ne pas gérer l'indice `i` du tout : l'ordinateur le fera de manière cachée afin de nous fournir directement les caractères les uns après les autres.

Ainsi, par définition, la version ci-dessous est équivalente à la précédente :

```
>>> def brinCompl (c) :
...     resultat = ""
...     for n in c :
...         resultat = complementaire(n) + resultat
...     return resultat
```

La variable `n` dans cette version remplace avantageusement le nucléotide qu'était précédemment `c[i]` : on n'a plus besoin de faire appel à un entier (`i`) pour obtenir chaque caractère (`c[i]`). La variable `n` vaut successivement, à chaque tour de la boucle, les caractères de la chaîne `c` du début à la fin. Il y a donc autant de tour de boucle `for` que de caractères dans `c` et le programmeur n'a plus à réfléchir sur les questions à propos de l'un indice `i`.
Nota : en revanche, l'indice `i` n'existe plus dans une boucle `for`, de sorte que si on en a besoin, il faut continuer à gérer une boucle `while` « à la main » !

3 Quelques exemples

Calcul du pourcentage de `G` ou `C` dans un brin d'ADN :

```
>>> def teneurGC(b) :
...     if len(b) == 0 :
...         print("Le brin est vide!")
...     else :
...         nbGC = 0
...         for n in b :
...             if n in "GC" :
...                 nbGC = nbGC + 1
...         return((nbGC * 100) // len(b))
```

Position du premier nucléotide illisible (X ou N) à l'issue d'un séquençage :

```
>>> def firstbad(b) :
...     i = 0
...     while i < len(b) :
...         if b[i] in "XN" :
...             return(i)
...         i = i + 1
...     print("Aucun nucléotide douteux dans ce brin")
```

Pourcentage de nucléotides exactement placés pareil sur deux brins d'ADN :

```
>>> def perfectmatch(u,v) :
...     if len(u) < len(v) :
...         lmin = len(u)
...         lmax = len(v)
...     else :
...         lmin = len(v)
...         lmax = len(u)
...     if lmax == 0 :
...         return(100)
...     else :
...         nbOK = 0
...         i = 0
...         while i < lmin :
...             if u[i] == v[i] :
...                 nbOK = nbOK + 1
...             i = i + 1
...         return((nbOK * 100) // lmax)
```

1 Les listes

Une *liste* en Python est une suite finie d'élements quelconques. Le type des listes est appelé `list`.

L'ensemble des listes est constitué des expressions de la forme

$$[e_1, \ldots, e_n]$$

où les e_i sont des données absolument quelconques (elles peuvent même être aussi elles-mêmes des listes).

```
>>> [2+3,"toto",2.5]
[5, 'toto', 2.5]
>>> type([5,"toto",2.5])
<type 'list'>
>>> [5,[9,5.5],"toto"]
[5, [9, 5.5], 'toto']
>>> type([5,[9,5.5],"toto"])
<type 'list'>
```

Il se peut qu'il n'y ait aucun élément dans la liste ; il s'agit alors de la *liste vide*, notée `[]`.

Beaucoup d'opérations travaillent sur les listes, à commencer par les opérations d'accès direct que nous venons de voir sur les chaînes de caractères. Ces dernières se transcrivent sur les listes de manière naturelle :

```
>>> ["toto",5,"tutu",5.5][1]
5
>>> ["toto",5,"tutu",5.5][1:3]
[5, 'tutu']
```

Le test d'appartenance ne fonctionne que pour un unique élément dans une liste, et non pas pour une sous-liste contrairement aux chaînes de caractères.

```
>>> "glop" in ["toto",5,"tutu",5.5]
False
>>> "tutu" in ["toto",5,"tutu",5.5]
True
>>> "tu" in ["toto",5,"tutu",5.5]
False
>>> [5,"tutu"] in ["toto",5,"tutu",5.5]
False
>>> [5,"tutu"] in ["toto",[5,"tutu"],5.5]
True
```

Comme pour les chaînes de caractères, on trouve également les opérations : `len` pour la longueur (nombre d'éléments dans la liste) et `+` pour la concaténation de deux listes.

```
>>> def homogene (l) :
...     if len(l) == 0 :
...         return True
...     else :
...         t = type(l[0])
...         i = 1
...         vu = True
...         while vu and i < len(l) :
...             vu = vu and t == type(l[i])
```

```
...             i = i + 1
...         return vu
...
>>> homogene (["toto",5,"tutu",5.5])
False
>>> homogene (["toto","glop","titi"])
True
>>> homogene ([])
True
```

Lorsqu'une liste `l` est homogène, les opérations `min(l)` et `max(l)` fournissent respectivement le plus petit et le plus grand élément de la liste.

On peut également énumérer les éléments d'une liste homogène avec `for` :

```
>>> def somme (l) :
...     s = 0
...     for n in l :
...         s = s + n
...     return s
...
>>> somme ([])
0
>>> somme ([2,2,5,3])
12
>>> min ([2,2,5,3])
2
>>> max ([2,2,5,3])
5
```

2 Instructions de modification de liste

Contrairement aux chaînes de caractères, une variable de type liste peut être partiellement modifiée sans réaffecter toute la variable. Par exemple :

```
>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire
['beurre', 'pain', 'artichaut', 'vin rouge']
>>> coursesAfaire[2]="avocat"
>>> coursesAfaire
['beurre', 'pain', 'avocat', 'vin rouge']
```

Ainsi, si v est une variable de type liste, alors l'instruction « v`[`i`]=`*expr* » remplace l'élément d'indice i dans la liste par la valeur du calcul de l'expression *expr*.

En revanche, avec des chaînes de caractères c'est impossible :

```
>>> nom="Samon"
>>> nom[1]
'a'
>>> nom[1]='i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On peut aussi remplacer une portion entière de liste par une autre liste :

```
>>> coursesAfaire[1:3]=["baguette","chou","vinaigre"]
```

```
>>> coursesAfaire
['beurre', 'baguette', 'chou', 'vinaigre', 'vin rouge']
```

Remarquez que dans l'instruction « v`[`i`:`j`]=`*expr* » on remplace les indices de i à $(j-1)$...

3 Particularités des modifications de liste en Python

Par défaut, l'expression doit être de type liste mais Python « fait ce qu'il peut » pour transformer l'expression en liste. Ici : une chaîne en liste de caractères...

```
>>> coursesAfaire = ['beurre', 'baguette', 'chou', 'vinaigre', 'vin rouge']
>>> coursesAfaire[1:3]="toto"
>>> coursesAfaire
['beurre', 't', 'o', 't', 'o', 'vinaigre', 'vin rouge']
>>> coursesAfaire[1:3]=8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
```

Plus généralement, une structure de données qui peut être transformée en liste de manière naturelle est dite « itérable » en Python.

Il est également possible de supprimer un élément ou une tranche d'éléments d'une variable de type liste :

```
>>> coursesAfaire
['beurre', 't', 'o', 't', 'o', 'vinaigre', 'vin rouge']
>>> del coursesAfaire[2]
>>> coursesAfaire
['beurre', 't', 't', 'o', 'vinaigre', 'vin rouge']
>>> del coursesAfaire[1:4]
>>> coursesAfaire
['beurre', 'vinaigre', 'vin rouge']
```

Évitez à tout prix de prendre des indices hors des bornes, le comportement n'est pas garanti ; cela ne produit pas forcément une erreur !

```
>>> del coursesAfaire[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> del coursesAfaire[1:6]
>>> coursesAfaire
['beurre']
```

En revanche, il est possible de donner des indices négatifs : cela signifie « en partant de la fin ». Ainsi « v`[-`i`]=`*expr* » est équivalent à « v`[len(`v`)-`i`]=`*expr* » :

```
>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire[-1]
'vin rouge'
>>> coursesAfaire[1:-1]
['pain', 'artichaut']
>>> del coursesAfaire[1:-1]
>>> coursesAfaire
['beurre', 'vin rouge']
```

Enfin pour les tranches d'éléments (avec le « `:` »), un indice manquant va jusqu'au bout de la liste :

```
>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire[1:]
['pain', 'artichaut', 'vin rouge']
>>> coursesAfaire[2:]
['artichaut', 'vin rouge']
>>> coursesAfaire[:2]
['beurre', 'pain']
>>> coursesAfaire[:]
['beurre', 'pain', 'artichaut', 'vin rouge']
```

4 Exemple de programme sur les listes

Ecrire une fonction `combien` qui prend en entrée deux arguments : le premier est une valeur `e` de type quelconque et le second est une liste `l`. Cette fonction doit retourner en résultat le nombre de fois où l'élément `e` apparaît dans la liste `l` (en particulier 0 fois si l'élément n'apparaît jamais dans la liste).

```
>>> def combien (e,l) :
...     compteur = 0
...     for x in l :
...         if x == e :
...             compteur = compteur + 1
...     return(compteur)
```

1 Lecture de fichiers

Un fichier est une suite de caractères mémorisés sur le disque dur de la machine dans un endroit caractérisé par un « nom de fichier ». Les contraintes techniques font qu'on doit charger du disque dur vers la mémoire ces caractères les uns après les autres pour pouvoir travailler dessus. Ainsi, en programmation, on gère un fichier comme un « flux de caractères » sur lequel on peut « ouvrir un robinet » pour charger un certain nombre de caractères, en partant du début du fichier. Après usage, il faut « fermer le robinet ».

Il y a beaucoup de fichiers sur un disque dur, donc beaucoup de robinets potentiels. L'ensemble de tous ces robinets potentiels constitue le type de données appelé `file`.

Les opérations qui travaillent sur le type `file` sont nombreuses. La première de toutes consiste à « ouvrir un fichier », ce qui met le robinet correspondant à disposition du programme qui a ouvert le fichier. La fonction `open` prend en argument un nom de fichier (qui est une chaîne de caractères) et retourne un objet de type `file` (le robinet).

Par la suite, le programme disposera du « robinet », qu'il pourra ouvrir à chaque instant pour récupérer des quantités déterminées de caractères, ceci avec la « fonction » `read` qui prend en argument le nombre de caractères qu'on veut lire. La première ouverture de robinet « lit » les premiers caractères du fichiers ; l'ouverture suivante, toujours avec `read` reprendra à partir du premier caractère pas encore lu du fichier, et ainsi de suite.

Après utilisation du fichier, il est *très important* de « rendre le robinet au système » pour que d'autres programmes puissent l'utiliser ou le modifier à leur tour. La fonction `close` assure cette tâche en rendant le robinet de nouveau inaccessible au programme. Imaginons un fichier sur le disque, appelé « toto.txt », qui contiendrait la suite de caractères « `TitiTrucMachinChouette` ». On peut par exemple écrire :

```
>>> rob = open("toto.txt")
>>> w = rob.read(4)
>>> x = rob.read(4)
>>> y = rob.read(6)
>>> z = rob.read(8)
>>> rob.close()
>>> w
'Titi'
>>> x
'Truc'
>>> y
'Machin'
>>> z
'Chouette'
```

On observe une syntaxe nouvelle : au lieu d'écrire `read(rob,4)` ou `close(rob)`, on écrit `rob.read(4)` et `rob.close()`. Ceci est dû au fait que le robinet `rob` n'est pas une donnée comme celles qu'on a vues jusqu'à maintenant : c'est un *objet*.

Un objet en informatique est une entité qui peut « réagir » à un programme de plusieurs manières différentes en fonction de l'état dans lequel il est. Par exemple, après `open`, l'état de l'objet `rob` est d'être en début de fichier. Ainsi la première lecture `rob.read(4)` vaut `"Titi"` et change de manière implicite l'état de `rob` : maintenant le robinet en est au cinquième caractère,et on le voit parce que le prochain appel de `rob.read(4)` ne fournit plus `Titi` mais `Truc`.

Les opérations qui travaillent spécifiquement sur des *objets* en informatique sont appelées des *méthodes* et s'utilisent avec cette notation « pointée » qui se veut rappeler qu'une méthode ne prend pas seulement sont objet en argument mais peut aussi changer son état.

`close` est finalement la méthode « d'apoptose du robinet »... et change radicalement l'état de son objet en le faisant disparaître (le robinet disparaît du programme mais le fichier reste parfaitement visible dans le disque dur).

2 Quelques méthodes utiles de lecture de fichiers

`readline()` : cette méthode lit autant de caractères que nécessaire pour aller jusqu'en fin de ligne (le retour-chariot inclus). Par exemple si `gamin.txt` contient les 4 lignes

```
premier doigt
deuxième doigt
second doigt
troisième doigt
```

alors on peut programmer :

```
>>> def affiche (f) :
...     fich = open(f)
...     i = 1
...     ligne = fich.readline()
...     while ligne != "" :
...         print("%i : %s" % (i,ligne))
...         i = i + 1
...         ligne = fich.readline()
...     fich.close()
...
>>> affiche("gamin.txt")
1 : premier doigt

2 : deuxième doigt

3 : second doigt

4 : troisième doigt
```

Lorsque l'objet fichier arrive en fin de fichier (plus rien à lire), la méthode `readline` retourne une chaîne vide. La procédure précédente s'arrête donc à la fin du fichier. Remarquons aussi que les trois premières lignes sont suivies d'une ligne vide : c'est dû au fait que `readline` inclût le retour-chariot (c'est-à-dire le passage à la ligne : la touche « Entrée » du clavier) *et* que `print` en écrit également un de sorte qu'il y a 2 passages à la ligne successifs.
À ce propos, dans la fonction `affiche`, si le fichier `f` contient une ligne vide au milieu des autres lignes, on pourrait croîre que cette ligne vide fasse sortir du `while` avant la fin du fichier. Il n'en est rien car pour une « ligne vide » la chaîne `s` n'est pas vide : elle contient le retour-chariot.

Enfin, précisons aussi que si le fichier ne se termine pas par un retour-chariot, alors le dernier `readline` fournit les caractères qui restent jusqu'à la fin de fichier, sans retour-chariot final.

`tell` : cette méthode retourne la position du robinet, c'est-à-dire le nombre de caractères déjà lus.

```
>>> r = open("toto.txt")
>>> r.read(4)
'Titi'
>>> r.read(4)
'Truc'
>>> r.tell()
8L
>>> r.close()
```

Le « `L` » signifie « entier `long` » car la taille d'un fichier peut être vraiment très grande et python encode la taille dans ce type, qui se manipule exactement comme les entiers de type `int`.

Ajoutons qu'on peut faire un `for` sur un fichier : cela énumère les lignes du fichier :

```
>>> f = open("gamin.txt")
>>> for ligne in f :
```

```
...     print(len(ligne))
...
14
15
13
15
>>> f.close()
```

On peut par exemple écrire d'une procédure `more(fichier)` qui affiche à l'écran le contenu du fichier, par pages de 24 lignes :

```
>>> def more(f):
...   hauteur=24
...   r=open(f)
...   vues=0
...   for l in r:
...     print(l[:-1])   # enlève le passage à la ligne de l
...     if vues < hauteur:
...       vues=vues+1
...     else:
...       s=input("suite... ")
...       vues=0
...   r.close
```

La dernière ligne, si elle ne se termine pas par un retour chariot, voit son dernier caractère disparaître. Pour bien faire, il faudrait tester si la dernière ligne contient une fin de ligne (`"\n"`) ou non.

3 Ecriture de fichiers

Il est possible « d'entrer un flux contraire dans un robinet » c'est-à-dire d'écrire dans un fichier. Il faut alors ouvrir le fichier non plus en lecture mais en écriture :
`open(nom,'w')` est une fonction qui crée un fichier de ce `nom`. Si un fichier du même `nom` existait, il est remis à zéro (vidé) par `open`.

Ensuite, il suffit d'utiliser la méthode `write` qui prend en argument une chaine de caractères et a pour effet de l'écrire dans le fichier.

```
>>> f = open("nouveau.txt","w")
>>> f.write("Toto")
>>> f.write("Tutu")
>>> f.write("Glop\n")
>>> f.write("ligne numero 2 seulement\n")
>>> f.close()
```

Le fichier contient alors :

```
TotoTutuGlop
ligne numero 2 seulement
```

et l'on remarque qu'il faut explicitement écrire, avec `write`, les retour-chariots sous la forme « `\n` ».

Ainsi la fonction `open` peut prendre 2 arguments. Le deuxième argument peut être
- `"r"` (comme *read*) : c'est l'argument par défaut, voir les sections précédentes.
- `"w"` (comme *write*) : si le fichier à l'adresse du premier argument n'existe pas alors il est créé ; s'il existe déjà alors il est entièrement écrasé.
- `"a"` (comme *append*) : si le fichier à l'adresse du premier argument n'existe pas alors il est créé ; s'il existe déjà alors les caractères sont ajoutés à la fin du fichier.

On peut par exemple écrire une procédure `merge` qui prend 3 arguments de type chaîne de caractères (`f1`, `f2` et `f3`) contenant des noms de fichiers : en supposant que les fichiers `f1` et `f2` contiennent des lignes triées par ordre alphabétique (selon l'ordre `ascii`), la procédure `merge` crée le fichier `f3` contenant l'union des lignes de `f1` et `f2`, également triées.

```
>>> def merge(f1,f2,f3) :
...   a = open(f1)
...   b = open(f2)
...   c = open(f3,"w")
...   x = a.readline()
...   y = b.readline()
...   while x != "" and y != "" :  # une ligne à voir dans chaque fichier
...     if x < y :
...       c.write(x)               # on écrit la plus petite des 2 lignes
...       x = a.readline()         # et on lit la suite du fichier utilisé
...     else :
...       c.write(y)               # idem
...       y = b.readline()
...   while x != "" :              # on termine le fichier non épuisé,
...     c.write(x)
...     x = a.readline()
...   while y != "" :              # un seul des 2 derniers while est exécuté
...     c.write(y)
...     y = b.readline()
...   a.close()
...   b.close()
...   c.close()
...   print("Le fichier " + f3 + " est rempli.")
```

En supposant que le fichier `toto.txt` contienne

```
alain
marc
pierre
robert
yves
zorro
```

et que `titi.txt` contienne

```
bernard
joe
luc
thomas
```

la commande `merge("toto.txt","titi.txt","tutu.txt")` fabrique le fichier `tutu.txt` contenant :

```
alain
bernard
joe
luc
marc
pierre
robert
thomas
yves
zorro
```

1 La programmation structurée

sur l'exemple du calcul de la date du lendemain...

Le problème s'énonce : on se donne une date sous la forme de trois entiers `(jour,mois,an)`, par exemple `(31,3,1995)` pour *le 31 mars 1995* et il faut que la fonction `lendemain` retourne le lendemain de ce jour, par exemple `(1,4,1995)` pour *le 1^{er} avril 1995*.

La phrase qui précède définit ce que le programme doit faire. Une telle description est appelée la *spécification* du programme. Passer d'une spécification en langue naturelle à un programme d'ordinateur n'est pas toujours facile *a priori*, cependant dans presque tous les cas la technique de *programmation structurée* permet d'aboutir à une solution élégante sans trop de difficulté.

Face à un problème, la technique de programmation structurée consiste, d'une certaine façon, à toujours choisir la solution de facilité en premier lieu. Non seulement ce n'est pas désagréable..., mais en plus, après avoir « inversé l'ordre de programmation » comme on le verra plus loin, on aboutit à un style de programmation extrêmement clair et facile à lire.

Essayons, et comme déjà vu mieux vaut commencer par éliminer les cas d'erreur, c'est-à-dire lorsque le triplet ne représente pas une date :

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > ??? :
...         print("La date fournie n'existe pas !")
...     else :
...         .... la suite ....
```

Bon, on tombe sur une première difficulté (les `???`) car le nombre maximal de jours possibles dépend du mois et c'est un calcul compliqué...

C'est là qu'intervient la programmation structurée : ***c'est compliqué ? qu'à cela ne tienne, on suppose qu'il existe déjà une fonction qui résoud le problème !***

Dans l'exemple qui nous occupe, on suppose donc qu'il existe une fonction `nbjours` qui prend en entrée le mois et retourne en résultat le nombre de jours de ce mois. Dès lors, la gestion des cas d'erreur est résolue et passons à `.... la suite ....`

Là, il y a plusieurs cas, selon qu'on est en fin de mois ou pas, en fin d'année ou pas. Le plus facile est naturellement lorsqu'il suffit d'ajouter 1 au jour. La programmation structurée, qui repousse comme on l'a dit toutes les difficultés à plus tard, nous dit de commencer par ce cas le plus facile.

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois) :
...         return ( jour+1 , mois, an )
...     else :
...         .... la suite ....
```

Jusque là, on s'en est sorti sans mettre trop de jus de cerveau, on imagine donc que les difficultés vont commencer dans `.... la suite ....` (ou bien dans la programmation de `nbjours`). Pour la suite, puisqu'on est dans le `else` et que la date est correcte, c'est que nous sommes le dernier jour du mois. Bref, finalement, si nous ne sommes pas en décembre ce sera facile car il suffit d'ajouter 1 au mois et de revenir au premier du mois. La programmation structurée nous dit donc de commencer par ce cas facile.

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois) :
```

```
...         return ( jour+1 , mois, an )
...     elif mois < 12 :
...         return ( 1 , mois+1 , an )
...     else :
...         .... la suite ....
```

Les difficultés seraient-elles derrière ce dernier `else` ? pas vraiment puisque derrière ce dernier `else` nous sommes nécessairement le 31 décembre, il suffit donc de passer au premier janvier de l'année d'après :

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois) :
...         return ( jour+1 , mois, an )
...     elil mois < 12 :
...         return ( 1 , mois+1 , an )
...     else :
...         return ( 1 , 1 , an+1 )
```

C'était donc simple finalement : en passant en revue un à un tous les cas les plus faciles, on arrive à la conclusion qu'il n'y a pas de cas difficile !

OK, la difficulté sera donc sans doute de programmer `nbjours`... La démarche de programmation structurée consiste, de manière assez naturelle, à ré-appliquer la même technique. On commence par la spécification :

> Le sous-problème s'énonce : on se donne un mois sous forme d'un entier compris entre 1 et 12 et la fonction `nbjours` doit renvoyer le nombre de jours de ce mois.

Programmation structurée : on commence par les cas faciles, c'est-à-dire les mois de 31 jours et les mois de 30 jours :

```
>>> def nbjours(m) :
...     if m in [4,6,9,11] :
...         return(30)
...     elif m != 2 :
...         return(31)
...     else :
...         .... aie aie aie ....
```

Là, on découvre qu'on a été un peu optimiste : les autres mois étaient faciles mais le mois de février est insoluble en l'état car le nombre de jours du mois de février dépend de l'année.

Donc il nous manque une donnée ; il faut que `nbjours` prenne aussi l'année en argument ! Cela nous obligera à modifier la manière d'appeler la fonction `nbjours` dans la fonction `lendemain` :

```
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois,an) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois,an) :
...         return ( jour+1 , mois, an )
...     elil mois < 12 :
...         return ( 1 , mois+1 , an )
...     else :
...         return ( 1 , 1 , an+1 )
```

et reprenons la résolution du sous-problème de `nbjours`... Là encore, on cherche à ne faire que des choses faciles. Si l'année est bissextile, il y a 29 jours, sinon il y en a 28. Ce qui est difficile, c'est de savoir si l'année est bissextile.

```
>>> def nbjours(m,a) :
...     if n == 4 or n == 6 or n == 9 or n == 11 :
...         return(30)
...     elif n != 2 :
```

```
...         return(31)
...     elif ??? :
...         return(29)
...     else :
...         return(28)
```

On commence à en avoir l'habitude maintenant : ***c'est compliqué ? qu'à cela ne tienne, on suppose qu'il existe déjà une fonction qui résoud le problème !***
On suppose donc qu'il existe une fonction `bissextile` qui résoud la question. Cette fonction devra renvoyer `True` si l'année est bissextile et `False` sinon.

```
>>> def nbjours(m,a) :
...     if n == 4 or n == 6 or n == 9 or n == 11 :
...         return(30)
...     elif n != 2 :
...         return(31)
...     elif bissextile(a) :
...         return(29)
...     else :
...         return(28)
```

C'était donc simple finalement. On imagine donc que c'est la fonction `bissextile` qui sera difficile à programmer...

La programmation structurée nous dit de commencer par spécifier cette fonction. Là, on fait un clic sur wikipedia par exemple et on finit par spécifier :

> Le sous-sous-problème s'énonce : on se donne une année sous la forme d'un nombre entier positif et la fonction `bissextile` doit retourner un booléen qui dit si l'année est bissextile. Une année est bissextile :
> — si elle est divisible par 4
> — mais qu'elle n'est pas divisible par 100
> — sauf que si elle est divisible par 400 alors elle est quand même bissextile.

Heu... bon, la programmation structurée dit de commencer par les cas faciles : si une année n'est pas divisible par 4, on est sûr qu'elle n'est pas bissextile.

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     else :
...         .... la suite ....
```

Il ne reste donc plus qu'à traiter dans `la suite` que les années divisibles par 4. C'est facile quand l'année est divisible par 400 car elle est dans ce cas toujours bissextile.

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     elif a % 400 == 0 :
...         return(True)
...     else :
...         .... la suite ....
```

Maintenant, une fois traitées les années multiples de 400, les années multiples de 4 sont bissextiles sauf si elles sont divisibles par 100 :

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     elif a % 400 == 0 :
...         return(True)
```

```
...     if a % 100 == 0 :
...         return(False)
...     else :
...         return(True)
```

Evidemment, Python n'est pas capable « d'attendre » qu'on ait programmé `nbjours` et `bissextile` pour traiter `lendemain`. Il faut donc inverser l'ordre de programmation. On programme donc en sens inverse de la manière dont on a réfléchi [1] :

```
>>> def bissextile(a) :
...     if a % 4 != 0 :
...         return(False)
...     elif a % 400 == 0 :
...         return(True)
...     else :
...         return (not (a % 100 == 0))
...
>>> def nbjours(m,a) :
...     if n == 4 or n == 6 or n == 9 or n == 11 :
...         return(30)
...     elif n != 2 :
...         return(31)
...     elif bissextile(a) :
...         return(29)
...     else :
...         return(28)
...
>>> def lendemain (jour,mois,an) :
...     if an < 0 or mois < 1 or mois > 12 or jour < 1 or jour > nbjours(mois,an) :
...         print("La date fournie n'existe pas !!")
...     elif jour < nbjours(mois,an) :
...         return ( jour+1 , mois, an )
...     elil mois < 12 :
...         return ( 1 , mois+1 , an )
...     else :
...         return ( 1 , 1 , an+1 )
```

Voilà, c'est résolu. La programmation structurée est fondée sur un mode de pensée plutôt confortable pour l'esprit : on commence par faire ce qui est simple, et on suppose déjà résolus (par des fonctions) les problèmes qui paraissent difficiles. On attaque ensuite chacune des fonctions qu'on a laissées de côté en suivant le même mode de pensée... et on finit par découvrir qu'on arrive au bout du problème sans jamais avoir eu à résoudre de problème très complexe !

Remarque subsidiaire : comparez le nombre de lignes du programme précédent et le nombre de millions de dollars qu'a coûté le fameux « bug de l'an 2000 »...

2 Les dictionnaires

Si je dois acheter 5 pains, 2 artichauts, 2 plaquettes de beurre et 1 bouteille de vin ? ... Il faut *associer*, d'une part, des éléments (`beurre`, `pain`, etc), et d'autre part des informations sur chaque élément (ici une quantité entière mais ce pourrait être une information aussi complexe que l'on veut). La structure de données qui mémorise cela est classiquement appelée une « liste d'association » mais en raison de la ressemblance avec un dictionnaire qui associe à chaque mot une définition, Python appelle ce type `dict` (comme dictionnaire).

L'ensemble des dictionnaires est constitué des expressions de la forme

$$\texttt{\{}\ e_1\texttt{:}d_1\ \texttt{,...,}\ e_n\texttt{:}d_n\ \texttt{\}}$$

1. Notez la petite simplification de `bissextile` que l'on a faite au passage

où les e_i sont des données *élémentaires* quelconques et les d_i sont des données quelconques. Par donnée élémentaire, on entend ici une donnée qui n'est pas elle-même une liste ou un dictionnaire.

```
>>> coursesPrecises = { "baguette":2 , "vin":1 , "saumon":2 }
>>> coursesPrecises[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> coursesPrecises["vin"]
1
```

On constate que l'on n'accède pas au contenu d'un dictionnaire par un indice entier donnant la position dans le dictionnaire mais, et c'est plus simple, par le nom des éléments appartenant au dictionnaire.

Attention : l'accès à un dictionnaire se fait par les éléments, pas par les informations qui lui sont associées.

On peut compléter ou modifier un dictionnaire avec une instruction d'affectation :

```
>>> coursesPrecises
{'baguette': 2, 'saumon': 2, 'vin': 1}
>>> coursesPrecises["saumon"]=1
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1}
>>> coursesPrecises["gateau"]=8
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'vin': 1, 'gateau': 8}
```

On peut aussi supprimer un élément (et son information associée) avec `del`, obtenir le nombre d'éléments avec `len` et tester si un élément est dans le dictionnaire avec `in`.

```
>>> del coursesPrecises["vin"]
>>> coursesPrecises
{'baguette': 2, 'saumon': 1, 'gateau': 8}
>>> len(coursesPrecises)
3
>>> "gateau" in coursesPrecises
True
>>> "chou" in coursesPrecises
False
```

Enfin, on peut *itérer* un bloc d'instructions sur tous les éléments d'un dictionnaire, un peu comme avec un `while` sur une chaîne de caractères ou sur une liste. Pour cela on utilise `for...in...`

```
>>> def total (d) :
...     s=0
...     for e in d :
...         s = s + d[e]
...     return s
...
>>> total(coursesPrecises)
11
```

Note : un dictionnaire avec des éléments ayant tous le même type et des informations ayant toutes le même type, c'est souvent mieux !

3 Exemples de programmes sur les dictionnaires

Un dictionnaire peut être typiquement utilisé pour collecter un ensemble d'informations à propos d'une collection d'objets. Prenons comme premier exemple un stock de médicaments :

```
>>> NbreDeBoites = {
...     "Doliprane":10,
...     "Efferalgan":3,
...     "Dafalgan":1,
...     "Levothyrox":0,
...     "Kardegic":15,
...     "Spasfon":20,
...     "Tahor":11,
...     "Voltarene":4,
...     "Methadone Aphp":5,
...     "Eludril":6,
...     "Ixprim":3,
...     "Paracetamol Biogaran":2
...     }
```

On peut alors par exemple calculer le nombre total de boîtes de médicaments dans un tel stock :

```
>>> def totalDico (d) :
...     r=0
...     for produit in d :
...         r = r + d[produit]
...     return(r)
...
>>> totalDico(NbreDeBoites)
80
```

On peut aussi calculer la liste des médicaments à renouveller dans le stock, en fonction du nombre minimum de boîtes que l'on souhaite :

```
>>> def renouveler(d,mini) :
...     commande=[]
...     for article in d :
...         if d[article] < mini :
...             commande=commande+[article]
...     return(commande)
...
>>> renouveler(NbreDeBoites,4)
['Levothyrox', 'Paracetamol Biogaran', 'Dafalgan', 'Ixprim', 'Efferalgan']
>>> renouveler(NbreDeBoites,2)
['Levothyrox', 'Dafalgan']
```

Si les clefs doivent être des données élémentaires, les informations qui leur sont associées peuvent en revanche être aussi complexes que l'on veut. Entre autres, ce peuvent être elles-mêmes des dictionnaires, comme pour ce second exemple de programmes sur les dictionnaires.

```
>>> anniv = { "Pierre" : {"jour":3,"mois":"jan","an":1965} ,
...           "Paul" : { "jour":18,"mois":"nov","an":1998} ,
...           "Irène" : {"jour":25,"mois":"mar","an":1982}  }
>>> anniv["Irène"]
{'mois': 'mar', 'jour': 25, 'an': 1982}
>>> anniv["Paul"]["an"]
1998
```

Supposons que l'on veuille écrire une fonction `lesAges` qui prend en entrée un dictionnaire `d` d'anniversaires similaire à `anniv` et une date courante `t`, et retourne en sortie la liste des âges des personnes du dictionnaire `d`.

```
def lesAges (d,t) :
    r = []
    for p in d :
```

```
        r = r + [...???...]
    return r
```

calculer l'âge d'une personne en fonction de son « dictionnaire anniversaire » et du dictionnaire représentant la date courante est de prime abord compliqué. Dans de tels cas, il faut toujours procéder de la manière suivante :

- on fait l'hypothèse qu'il existe des fonctions qui résolvent les problèmes compliqués (ici faire la différence de deux dates en nombre d'années entières : `diffDates(t1,t2)`) ;
- on programme comme si ces fonctions existaient (ici, `r = r + [diffDates(d[p],t)]`) ; naturellement Python n'acceptera cette programmation qu'après avoir réellement programmé ces fonctions ;
- on s'attaque ensuite, une par une, aux fonctions identifiées par le processus précédent, et on leur applique exactement la même approche ;
- la programmation est terminée lorsque les fonctions deviennent suffisamment simples pour ne plus avoir besoin de faire appel à des fonctions hypothétiques.

Ici, pour programmer `diffDates`, si la date de `t1` vient après celle de `t2` dans l'année, il suffit de faire la soustraction des années ; sinon il faut soustraire 1 au résultat :

```
def diffDates (t1,t2) :
    if apresAnnee(t1,t2) :
        return t1["an"] - t2["an"]
    else :
        return t1["an"] - t2["an"] - 1
```

La fonction hypothétique qu'il faut maintenant programmer est `apresAnnee`. Ce qui est difficile est alors de comparer des mois qui ne sont pas des entiers mais des chaînes de caractères. Qu'à cela ne tienne, on fait l'hypothèse que la fonction `numero` fournit le numéro du mois.

```
def apresAnnee (u,v) :
    if numero(u["mois"]) > numero(v["mois"]) :
        return True
    elif numero(u["mois"]) == numero(v["mois"]) :
        return u["jour"] > v["jour"]
    else :
        return False
```

Ensuite, lorsque l'on veut programmer la fonction `numero`, on se rend vite compte qu'il est préférable d'avoir un tableau plutôt qu'une fonction :

```
numero={"jan":1, "Jan":1, "janv":1, "Janv":1, "janvier":1, "janvier":1,
        "fev":2, "Fev":2, "fevrier":2, "Fevrier":2, "fév":2, "Fév":2,
        "février":2, "Février":2,
        "mar":3, "Mar":3, "mars":3, "Mars":3,
        ...
       }
```

et du coup il faut revoir `apresAnnee`, en mieux :

```
def apresAnnee (u,v) :
    if not (u["mois"] in numero) or not (v["mois"] in numero)
        print("mois mal formé !")
    elif numero[u["mois"]] == numero[v["mois"]] :
        return u["jour"] >= v["jour"]
    else :
        return numero[u["mois"]] > numero[v["mois"]]
```

1 Les modules en Python

En ce point du cours, nous avons étudié la majorité des structures de données classiques (manquent les *arbres* et les *graphes* qui relèvent de cours plus avancés) et la quasi-totalité des structures de contrôle (le *traitement d'exceptions* et la *récursivité* relèvent de cours plus avancés). On peut même affirmer que quel que soit le problème posé, s'il existe un programme qui le résoud alors nous avons vu tous les éléments pour le faire.

Il existe cependant des questions « standard » qu'il serait peu productif de résoudre chacun pour soi : ces questions sont tellement courantes que les programmes qui les résolvent sont stockés dans des *bibliothèques* (*libraries* en englais). En Python, un telle bibliothèque est constituée de nombreux *modules* et un module est constitué de plusieurs fonctions (ou procédures) déjà programmées. Dans un module, on prend soin de regrouper les fonctions qui permettent de gérer un type de problème donné, bien souvent ce sont simplement les opérations associées à une structure de données adaptée au problème.

On peut même écrire soi-même des modules. Lorsque l'on veut écrire un gros logiciel, c'est généralement une bonne idée de le découper en modules (qui seront du même coup réutilisables par ailleurs), en suivant généralement le principe « au moins un module par nouvelle structure de donnée ». Il suffit alors d'importer les modules pour utiliser les fonctions qui y ont été programmées.

2 Gestion des fichiers : le module « `os` »

Pour utiliser un module, il faut d'abord le charger avec la commande `import`. Parmi les modules vraiment utiles, il y a celui qui permet de gérer l'arborescence des fichiers de l'ordinateur. Il se nomme `os`, comme « operating system », et nous nous servirons de ce module comme exemple.

```
>>> import os
```

Remarquez que, exceptionnellement, `os` n'est pas écrit comme une chaîne de caractère (donc entre guillemets) et pourtant `import` ne considère pas `os` comme un nom de variable...Avant d'utiliser `os` pour gérer le système de fichiers, il faut savoir :

- qu'il peut y avoir plusieurs disques dur sur un ordinateur, ou qu'un disque dur peut être partagé en plusieurs partitions. Sous Mac ou un système Unix comme Linux cette séparation des disques et des partitions est gérée pour qu'on ait l'impression de n'avoir qu'un seul disque. Sous Windows, ces partitions sont vues comme plusieurs « lecteurs » différents numérotés `A`, `B`, `C`, `D`, etc. En général `A` et `B` sont présents pour des raisons historiques et sont réservés à deux lecteurs de disquettes. Le disque `C` est généralement celui sur lequel le système Windows est installé et les suivants sont des disques de données ou des lecteurs/graveurs de CD, DVD, etc.
- Un disque est organisé en arbre avec des répertoires (directories en anglais ou « dossiers » pour les utilisateurs naïfs) qui peuvent contenir des sous-arbres et des fichiers (files) qui contiennent les « vraies » données et sont nécessairement des feuilles de l'arbre.
- Un nœud de l'arbre est repéré par sont chemin depuis la racine, qui est souvent appelé son adresse.
- Compte tenu de la taille de cet arbre et de la longueur des chemins, et du fait qu'on travaille longtemps sous un même répertoire, il est pratique d'avoir un répertoire dit « courant ». Cela permet de ne donner que les adresses à partir du répertoire courant.
- On note généralement « `..` » le répertoire père du répertoire de travail, et « `.` » ce répertoire lui-même.

Pour utiliser une fonction ou une procédure d'un module, il faut la préfixer par le nom du module (une fois qu'il a été importé). Par exemple `getcwd` est une fonction qui retourne l'adresse du répertoire dans lequel on est en train de travailler (« cwd » pour current working directory).

```
>>> os.getcwd()
'/home/bernot'
```

Le module `os` offre aussi une procédure qui crée un répertoire, appelée `mkdir` (make directory) :

```
>>> os.mkdir("test")
```

Pour se déplacer dans les répertoires, et donc changer l'adresse du répertoire de travail, on utilise la procédure `chdir` (change directory) et l'on peut lister le contenu d'un répertoire avec la fonction `listdir`, qui retourne la liste des éléments contenus dans le répertoire :

```
>>> os.chdir("test")
>>> os.listdir("/home/bernot/test")
[]
>>> f=open("toto.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'toto.txt'
>>> f=open("toto.txt","w")
>>> f.close()
>>> os.listdir("/home/bernot/test")
['toto.txt']
```

Il existe des abréviations : la chaîne `"."` remplace le répertoire de travail courant et la chaîne `".."` désigne son répertoire parent :

```
>>> os.listdir(".")
['toto.txt']
>>> os.mkdir("repfils")
>>> os.listdir(".")
['repfils', 'toto.txt']
>>> os.chdir("repfils")
>>> os.getcwd()
'/home/bernot/test/repfils'
>>> os.listdir(".")
[]
>>> os.listdir("..")
['repfils', 'toto.txt']
```

D'autres fonctions ou procédures utiles pour la gestion du système de fichiers sont :

- `rename` prend en argument les chemins source et cible : peut déplacer dans l'arbre, aussi bien un fichier qu'un sous-arbre complet
- `remove` prend en argument le chemin d'un fichier et le supprime
- `rmdir` comme `remove`, mais pour un répertoire vide

On remarque que la liste obtenue par `listdir` mélange des répertoires et les « vrais » fichiers sans moyen de les distinguer. Le module `os` contient un « sous-module » `os.path` qui est automatiquement importé avec `os` et fournit d'autres fonctions bien utiles :

- `isfile` dit si le chemin pointe sur une fichier standard
- `isdir` ... si c'est un répertoire
- `islink` ... si c'est un lien symbolique (« raccourci » sous windows)
- `exists` dit si le chemin donné en argument existe (fonction booléenne)
- `getsize` taille en octets du fichier désigné par le chemin donné en argument
- `abspath` prend un chemin en entrée et retourne sa version en adresse absolue.
- `basename` fournit le dernier nom du chemin
- `dirname` le contraire

```
>>> os.isfile("toto.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'isfile'
>>> os.path.isfile("toto.txt")
True
>>> os.path.isdir("toto.txt")
False
```

3 Créer son propre module en Python

Pour créer un module, il suffit de programmer les fonctions qui le constituent dans un fichier portant le nom du module, suivi du suffixe « `.py` ». Depuis un (autre) programme en Python, il suffit alors d'utiliser la primitive `import` pour pouvoir utiliser ces fonctions. On peut aussi définir des variables globales dans un module.

Par exemple, si l'on crée un fichier appelé `pile.py` et contenant :

```
contenu = {}

def push(e,p) :
  global contenu
  if p in contenu :
    contenu[p] = contenu[p] + [e]
  else :
    contenu[p] = [e]

def height(p) :
  global contenu
  if p in contenu :
    return len(contenu[p])
  else :
    return 0

def pop(p) :
  global contenu
  if p in contenu and len(contenu[p]) > 0 :
    top = contenu[p][-1]
    contenu[p] = contenu[p][:-1]
    return top
  else :
    print ("pile.pop: ERREUR la pile %s est vide !" % p)

def show(p) :
  global contenu
  if p in contenu :
     for e in contenu[p] :
       print (e)
```

alors sous Python on peut écrire :

```
>>> import pile
>>> pile.contenu
{}
>>> pile.push(36,"test")
>>> pile.contenu
{'test': [36]}
>>> pile.push(2,"test")
>>> pile.contenu
{'test': [36, 2]}
>>> pile.push(2,"seconde")
>>> pile.contenu
{'test': [36, 2], 'seconde': [2]}
>>> pile.height("test")
2
>>> pile.height("seconde")
1
>>> pile.height("autre")
0
>>> pile.show("test")
36
```

```
2
>>> pile.pop("test")
2
>>> pile.show("test")
36
>>> pile.pop("test")
36
>>> pile.show("test")
>>> pile.height("test")
0
>>> pile.contenu
{'test': [], 'seconde': [2]}
```

On remarque que :

- `import` est une instruction de contrôle de Python très particulière puisqu'il n'est pas nécessaire d'écrire `pile` entre guillements pour en faire une chaîne de caractères, pourtant, `import` ne considère pas `pile` comme un nom de variable !
- On n'écrit pas le suffixe du fichier (`.py`) mais seulement le nom du module.
- Pour utiliser les fonctions et procédures du module, il faut les préfixer du nom du module avec un point au milieu.

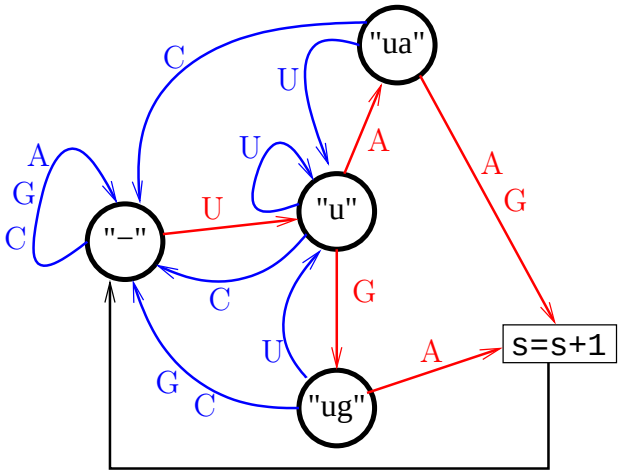
Python trouve ses modules dans le répertoire courant, comme on vient de le voir, mais aussi dans des répertoires prédéfinis où se trouvent des modules utiles écrits par d'autres programmeurs, comme ce fut le cas pour le module `os`.

Un module peut faire appel à un autre module : il suffit qu'il contienne lui-même une instruction `import`.

Les modules dits « standard » sont documentés dans de nombreux livres sur Python et, avec un peu d'habitude, l'appel à des modules renforce considérablement la rapidité de programmation. Libre à vous d'explorer la liste presque sans fin des modules existants (et qui s'enrichit tous les jours) : vous avez maintenant toutes les bases de programmation nécessaires pour comprendre leur usage à partir de leur documentation.

1 Les automates... et l'instruction manquante en Python !

Les « automates » en informatique désignent en fait une « façon de penser » pour écrire des programmes qui doivent traiter des données qui arrivent les unes après les autres, et desquelles on doit extraire des informations. Cette technique permet de ne lire les données qu'une seule fois.
Par exemple, lorsqu'on programme la recherche de codons STOP dans un brin d'ARN `b`, la technique qui consiste à comparer, pour chaque indice `i`, la tranche de 3 nucléotides `b[i:i+3]` avec les codons STOP (`UAA`, `UAG` ou `UGA`) revient à lire 3 fois chaque nucléotide puisqu'un codon `b[j]` est lu dans les trois tranches `b[j-2:j+1]`, `b[j-1:j+2]` et `b[j:j+3]`.
Si l'on veut n'utiliser qu'une seule fois chaque nucléodite successivement, par conséquent en utilisant un `for` sur le brin `b`, c'est en faisant un dessin que l'on comprend ce qu'il faut faire (figure de droite).



— Au départ, n'ayant encore « lu » aucun nucléotide, on n'a « reconnu » aucun début de codon STOP. On se place alors dans l'état `"-"` qui indique ce fait.
— Depuis cet état où l'on n'a rien reconnu encore, si on lit un `A`, un `G` ou un `C` alors on n'a toujours rien reconnu, puisque tous les codonc STOP commencent par `U`.
— En revanche, si l'on a lu un `U`, alors on passe dans un état où l'on a reconnu le premier nucléotide d'un codon STOP. On passe donc de l'état `"-"` à l'état appelé `"u"` sur le dessin, qui signifie conventionnellement qu'on a reconnu le premier nucléotide `U`.
— Depuis cet état `"u"`, si on lit un `C` alors ce n'était pas le début d'un codon STOP et l'on retourne à l'état de départ `"-"` où l'on n'a rien reconnu. En revanche, si on lit un `G` (ou un `A`) alors on passe dans un état où l'on a reconnu 2 nucléotides `"ug"` (ou respectivement `"ua"`). Enfin si on lit un second `U`, alors le premier n'était pas le premier nucléotide d'un codon STOP, mais celui qu'on vient de lire, peut-être... donc on reste dans l'étau `"u"`.
— Depuis l'état `"ug"`, si on lit un `G` ou un `C` alors tout est à refaire, on repart de l'état `"-"` où aucun nucléotide n'a été reconnu. Si on lit un `U` alors `UGU` n'est pas un codon STOP mais le `U` qu'on vient de lire est peut-être le premier nucléotide d'un codon STOP. Donc, on retourne dans l'état `"u"` où l'on a reconnu un seul nucléotide. Enfin, si on lit un `A`, alors on a reconnu le condon STOP `UGA`, donc on ajoute 1 au nombre de codons STOP rencontrés et on repart pour la suite dans l'état `"-"`.
— De même, depuis l'état `"ua"`, si on lit `A` ou `G` alors on a reconnu un codon STOP et l'on ajoute 1 au nombre de codons STOP rencontrés. En revanche si on lit un `C`, il faut rapartie à zéro (état `"-"`). Enfin si on lit un `U`, on repart vers l'état où l'on n'a reconnu qu'un nucléotide (état `"u"`).

À la fin de la boucle `for`, la somme `s` contiendra exactement le nombre de codonc STOP reconnus :

```
>>> def nbSTOP(b):
...     s=0
...     etat="-"
...     for c in b :
...       if etat == "-" :
...         if c == "U" :
...           etat = "u"
...       elif etat == "u" :
...         if c == "C" :
...           etat = "-"
...         elif c == "A" :
...           etat = "ua"
...         elif c == "G" :
...           etat = "ug"
...       elif etat == "ua" :
...         if c == "C" :
```

```
...         etat = "-"
...       elif c == "U" :
...         etat = "u"
...       else :
...         s = s + 1
...         etat = "-"
...     elif etat == "ug" :
...       if c == "U" :
...         etat = "u"
...       elif c == "A" :
...         s = s + 1
...         etat = "-"
...       else :
...         etat = "-"
...     else :
...       print("PROBLEME !")
...   return(s)
```

De ce point de vue, Python est l'un des seuls langages de programmation qui n'offrent pas une instruction de « distribution des calculs en fonction des cas ». On en arrive donc à cette accumulation de `elif` assez inélégante. Les autres langages de programmation impérative permettrait d'écrire l'automate de manière légèrement plus lisible :

```
def nbSTOP(b):
  s=0
  etat="-"
  for c in b :
    switch etat :
      case "-" :
          switch c :
            case "U" :
              etat = "u"
      case "u" :
          switch c :
            case "C" :
              etat = "-"
            case "A" :
              etat = "ua"
            case "G" :
              etat = "ug"
      case "ua" :
          switch c :
            case "C" :
              etat = "-"
            case "U" :
              etat = "u"
            default :
              s = s + 1
              etat = "-"
      case "ug" :
          switch c :
            case "U" :
              etat = "u"
            case "A" :
              s = s + 1
              etat = "-"
            default :
              etat = "-"
  return(s)
```

Malheureusement ceci n'est pas un programme Python par manque de l'instruction `switch`.

Nota : sur la page web `http://www.i3s.unice.fr/~bernot/Enseignement/GB3_Python1` vous trouverez, avec un retard de quelques jours par rapport au cours, des notes de cours et les feuilles de TD.

1 Installation de Python

Vérifiez tout d'abord que Python ne soit pas déjà installé sur votre machine. Si oui, passez cette partie du TD...

- Cherchez (avec un moteur de recherche, par exemple startpage) la page officielle de Python et choisissez la version stable la plus récente.
- Ne pas utiliser les « sources » : ce sont des textes de programmes qui doivent être compilés pour fabriquer les commandes utiles pour le langage Python (python, IDLE, etc.). Mieux vaut utiliser les installations déjà compilées, qui sont alors dépendantes du système que vous utilisez : Windows, Linux, Mac OS 10, *etc.*
- Chargez celle qui convient. Cela installera dans vos menus diverses commandes utiles pour Python.

Lancez parmi ces commandes « IDLE ».

- Une fenêtre apparaît, avec « >>> » et le curseur qui attend que vous commenciez à programmer, et donnera les résultats au fur et à mesure de vos commandes.
- Dans le menu de IDLE, *File/New* permet d'accéder à un éditeur. Dans cet éditeur on peut modifier à loisir un texte de programme en Python mais, contrairement à la première fenêtre IDLE, cela ne l'exécute pas au fur et à mesure des lignes tapées. Pour exécuter le programme, il faut choisir *Run* dans le menu de l'éditeur : cela demande un nom de fichier la première fois, où le texte du programme sera sauvegardé avant d'être exécuté.
- Ultérieurement, dans le menu de IDLE, *File/Open* permet d'éditer un ancien fichier de programme de son choix. Il faut donc choisir des noms de fichier « parlants »...

2 Premières fonctions et procédures en Python

Exercice 1 : Écrivez en python une fonction `carre` qui prend en entrée un nombre entier relatif (de type `int`) et retourne en sortie son carré.

Par exemple `carre(5)` retourne `25`.

Faites des essais d'utilisation avec plusieurs valeurs entières.
Faites aussi un essai avec une valeur réelle (type `float`) : ça marche aussi ! pourquoi à votre avis ?

Exercice 2 : Écrivez en python une procédure `double` qui prend en entrée un nombre entier relatif `n` et imprime à l'écran `"le double de ... est ..."` qui indique à combien est égal le double de `n`.

Par exemple `double(6)` imprime à l'écran `"le double de 6 est 12"` (mais ne retourne aucun valeur en tant que fonction).

Faites 2 versions, l'une utilisant la concaténation `+`, l'autre avec l'opération de substitution `%`.

Exercice 3 : Quels résultats donneraient respectivement les expressions `1 + carre(2)` et `1 + double(2)` ?

Vérifiez vos prédictions *après* les avoir formulées...

Exercice 4 : Écrivez une fonction `triangle` de 3 arguments `a`, `b` et `c` qui indique si ces 3 réels définissent les côtés d'un triangle, en suivant le procédé suivant : le plus grand des côtés doit être inférieur à la somme des deux autres.

À l'occasion de ces premiers exercices, on découvre entre autres :

- que l'indentation (les marges) doivent impérativement être scrupuleusement respectées,
- qu'il faut une ligne vide à la fin d'un « `def ... :` » pour que l'ordinateur comprenne qu'on a fini la définition de fonction,
- que les calculs que l'on écrit doivent tous respecter une forme « arborescente » dans laquelle le type retourné par un sous-arbre de calcul doit coïncider avec celui attendu par chaque fonction en argument, en particulier il ne faut pas écrire « `a >= b and c` » qui est mal typé si `a`, `b` et `c` sont des entiers, alors que l'on voulait écrire « `a >= b and a >= c` »
- *etc.*

3 Quelques autres fonctions et procédures en Python

Exercice 5 : Écrivez en python une fonction `direAge` qui prend en entrée une chaîne de caractères `p` (qui sera en fait un prénom) et un entier `n` (sa date de naissance), et qui retourne une chaîne de caractères en suivant l'exemple suivant :

`direAge("Max",1993)` retourne la chaîne `"Max a 21 ans."`

Si la différence entre l'année courante (2014) et la date de naissance est supérieure à 150, la fonction devra imprimer à l'écran une erreur indiquant que la date de naissance est improbable.

Exercice 6 : Écrivez en python une fonction `filtre` qui prend en entrée un entier relatif `n`, qui imprime à l'écran un message d'erreur si `n` est négatif, qui retourne `n` lui même s'il est inférieur à `1000` et la moitié (entière) de `n` s'il est supérieur à `1000`.

Par exemple `filtre(-36)` imprime `Erreur!` à l'écran ; `filtre(358)` retourne `358`, `filtre(1050)` retourne `525` et `filtre(1051)` aussi.

Exercice 7 : Écrivez en python une procédure `ecrit_ligne` qui prend en entrée une chaîne de caractères et l'imprime à l'écran sous réserve qu'elle fasse moins de 50 caractères de long. Si elle fait plus de 50 caractères, la procédure écrit seulement la ligne « `TROP LONG!` ».

Par exemple, `ecrit_ligne("ligne assez courte.")` imprime à l'écran « `ligne assez courte.` ». Par contre, si on lui donne en entrée `"01234567890123456789012345678901234567890123456789glop"` la procédure `ecrit_ligne` imprime à l'écran « `TROP LONG!` ».

Exercice 1 : Écrivez en python une fonction `arrondi_pair` qui prend en entrée un nombre entier relatif `n` et qui retourne le nombre pair immédiatement inférieur ou égal à `n`.

Par exemple, `arrondi_pair(5)` retourne `4`, `arrondi_pair(6)` retourne `6`, `arrondi_pair(7)` retourne `6`, `arrondi_pair(8)` retourne `8`, etc.

Indication : on peut utiliser la division entière. `(7 // 2)` vaut `3` (et il reste 1), donc `(7 // 2) * 2` vaut `6`, le résultat attendu.

Exercice 2 : Écrivez une fonction `avant` qui prend en entrée 4 entiers, `a`, `b`, `x` et `y`, et qui dit si le mois numéro `b` de l'année `a` arrive strictement avant le mois numéro `y` de l'année `x`. On doit donner un message d'erreur si une année est négative ou si un mois n'existe pas.

Exercice 3 : Écrivez une fonction `age` qui prend en entrée 4 entiers, `a`, `b`, `x` et `y`, et qui dit quel âge a (ou avait ou aura) une personne née l'année `x` au mois numéro `y`, à la date dont l'année est `a` et le mois est `b`. Prenez soin de bien gérer les cas d'erreur. Toute solution raisonnable est acceptée si `b`=`y`.

Exercice 4 : Écrivez une procédure `accueil` sans arguments qui demande à l'utilisateur ses noms, prénom, année de naissance et mois de naissance, et qui imprime à l'écran une phrase complète lui souhaitant la bienvenue et lui indiquant son âge.

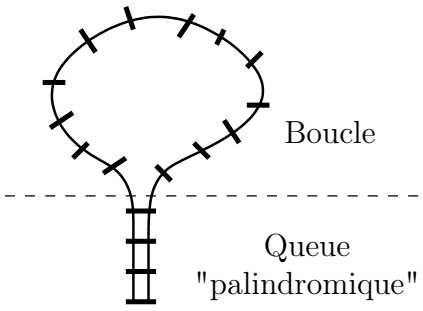
Chaînes de caractères et palindromes

Exercice 1 : Écrivez une fonction `inverse` qui prend en entrée une chaîne de caractères et fournit en sortie la chaîne inversée. Par exemple `inverse ("Gaston")` fournit la chaîne `"notsaG"` en résultat.

Exercice 2 : Écrivez une fonction `palindrome` qui prend en entrée une chaîne de caractères et fournit en sortie un booléen qui dit si c'est un palindrome.

- Faites une première version simple qui consiste à comparer la chaîne avec son inverse (calculé grâce à la fonction `inverse` de l'exercice précédent).
- La première version a le désavantage de parcourir 2 fois la chaîne de caractères (une fois pour l'inverser et une fois pour la comparer avec l'original). Écrivez une seconde version, 4 fois plus rapide, qui se limite à deux fois moins de comparaisons que de caractères dans la chaîne.

Exercice 3 : Écrivez une fonction `boucle` qui prend en entrée une séquence d'ARN et qui fournit en sortie la boucle maximale obtenue en supprimant aux extrémités les sous-séquences qui se replient l'une sur l'autre selon le dessin suivant :



Limitations : cette fonction présente plusieurs limitations. Lequelles ? Si vous finissez le TD avant la fin, essayez de programmer une meilleure fonction, même plus générale que celle de l'exercice 5 (ce n'est pas vraiment facile... `:-)`).

Exercice 4 : Écrivez une fonction `phase` :

- qui prend en entrée deux chaînes de caractères `c` et `b`, où `c` est supposé être un codon (donc chaîne d'`ATGC` de 3 caractères exactement) et `b` un brin d'ADN (donc une chaîne ne contenant que des `ATGC`),
- qui fournit en sortie la phase de lecture (1, 2 ou 3) du codon dans le brin,
- et 0 dans le cas où le codon n'est pas dans le brin.

Exercice 5 : *(exercice subsidiaire)*

Écrivez une fonction `boucle2` qui étend la fonction `boucle` de l'exercice 3 en acceptant que l'une des deux extrémités du brin d'ARN « dépasse » de la queue palindromique.

Encore mieux (`boucle3`) : ne vous arrêtez pas au premier palindrome rencontré, ne retenez que le plus long d'entre eux...

Rappel : chacune de vos fonctions peut faire appel à des fonctions que vous avez déjà programmées auparavant.

Premières fonctions sur les listes

Exercice 1 : Écrivez une fonction `indice` qui prend en entrée une valeur quelconque `e` et une liste `l`, et fournit en sortie :
- l'indice de l'élément `e` dans la liste `l` si `e` est dans `l` (l'indice est alors compris entre *zéro* et la longueur de la liste *moins un*),
- la longueur de la liste `l` si `e` n'est pas dans `l`.

Exercice 2 : Écrivez une fonction `longueurMoyenne` qui prend en entrée une liste dont les éléments sont des chaînes de catactères (par exemple des séquences d'ARN messagers), et qui retourne en sortie la longueur moyenne des chaînes appartenant à la liste.
- On prendra soin de renvoyer un nombre réel (plus précis qu'un entier).
- Si la liste est vide, renvoyer 0.

Exercice 3 : Certaines expériences consistent à mesurer à intervalles assez réguliers (par exemple une fois par jour) le niveau d'expression de plusieurs gènes (typiquement *via* des mesures de fluorescence sur des puces à ADN). On obtient alors, pour chaque gène, un *profil d'expression* au cours de l'expérience, c'est-à-dire la suite des mesures de niveau d'expression de ce gène. Le plus souvent, les niveaux d'expression sont des entiers compris entre 0 et 255 et un profil d'expression est donc en Python une simple liste d'entiers. La longueur de la liste est alors égale au nombre de mesures effectuées sur la durée de l'expérience.

On peut par exemple mener une expérience sur 2 organismes, l'un soumis à un stress et l'autre pas : les gènes ayant des profils d'expression différents d'un organisme à l'autre participent probablement à la réponse de l'organisme au stress appliqué. Pour les gènes qui ne sont pas impliqués, on ne peut naturellement pas attendre des profils d'expression exactement égaux, cependant on peut attendre qu'il soient « co-exprimés » c'est-à-dire que si l'un augmente d'une mesure à la suivante, alors l'autre aussi, et de même s'il diminue.

Écrivez une fonction `coexprime` qui prend en entrée deux profils d'expression pour un gène donné (p1 et p2) et qui retourne un booléen qui dit s'ils sont covariants. Les deux listes sont supposées de même longueur puisqu'elles représentent des mesures successives faites en même temps sur les deux organismes.

Exercice 4 : Bien comprendre les deux fonctions suivantes :

Compresser un brin d'ADN où il y a beaucoup de répétitions successives de nucléotides. L'idée est de ne pas recopier plusieurs fois un nucléotide répété mais de mémoriser dans la liste son nombre d'occurrences. Par exemple la fonction `comprime` appliquée à la chaîne `"AAAGCTTCCCG"` retourne comme résultat la liste `['A',3,'G','C','T',2,'C',3,'G']`. On suppose sans le verifier que le brin est correct (i.e. ne comprend que des `A`, `T`, `G` ou `C`).

```
>>> def comprime (brin) :
...     r = []
...     compteur = 0
...     precedent = ""
...     for nucl in brin :
...         if nucl == precedent :
...             compteur = compteur + 1
...         else :
...             # mémoriser le nucleotide precedent:
...             if compteur > 1 :
...                 r = r + [ precedent , compteur ]
...             elif compteur == 1 :
...                 r = r + [ precedent ]
...             # préparer la suite:
...             precedent = nucl
...             compteur = 1
...     #traiter le dernier nucleotide en sortant du for:
...     if compteur >= 2 :
...         return r + [ precedent , compteur ]
...     else :
...         return r + [ precedent ]
```

T.S.V.P. ⟶

et pour décompresser :

```
>>> def deploie (l) :
...     r = ""
...     precedent = ""
...     for elem in l :
...         if type(elem) != type(0) :
...             r = r + elem
...             precedent = elem
...         else :
...             while elem > 1 :
...                 r = r + precedent
...                 elem = elem - 1
...     return r
```

Fonctions sur les listes et les chaînes de caractères

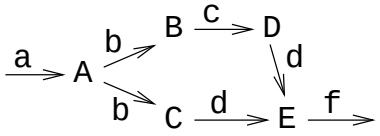
Exercice 1 : Écrivez une fonction `coupe` qui prend en entrées un nombre strictement positif `n` et une chaîne de caractères `s` représentant un chromosome, et qui retourne la liste de brins d'ADN obtenue en coupant le chromosome par morceaux de `n` nucléotides de long (le dernier morceau peut être plus court).

Exercice 2 : Écrivez une fonction `stopPhases` qui prend en entrée une chaîne de caractères `b` représentant un brin d'ARN et qui fournit en sortie une liste contenant trois entiers : le nombre de condons stop (`UAA`, `UAG` ou `UGA`) contenus dans chacune des trois phases de lecture.

Exercice 3 : On veut représenter les réseaux métaboliques par des listes de la forme

$$[E_0, a_0, b_0, E_1, a_1, b_1, E_2, a_2, b_2, \ \ldots\ ]$$

où les E_i sont des chaînes de caractères qui représentent des noms d'enzymes, les a_i et b_i sont chaînes de caractères qui représentent des noms de substrats, et « E_i,a_i,b_i » indique que l'enzyme E_i transforme a_i en b_i. On suppose qu'il n'y a pas d'enzyme qui porte le même nom qu'un substrat.



1. Définissez une liste `M` qui représente le réseau métabolique suivant :
2. Écrivez une fonction `produit` qui prend en entrées deux métabolites, `s` et `p`, et une liste `m` représentant un réseau métabolique, et qui dit s'il existe un enzyme ayant `s` comme substrat et `p` comme produit. Testez-la avec quelques exemples.
3. Écrivez une fonction `existe` qui prend en entrées une liste de métabolites successifs `c` (`c` comme « chemin ») et une liste `m` représentant un réseau métabolique, et qui dit si le chemin `c` existe dans le réseau métabolique `m`. Testez-la avec quelques exemples.

Exercice 4 : (DIFFICILE) Écrivez une fonction `consecutifs` qui prend en entrée une liste d'enzymes successifs `c` (`c` comme « chemin ») et une liste `m` représentant un réseau métabolique, et qui dit si le chemin `c` existe dans le réseau métabolique `m`. Testez-la avec quelques exemples.

Programmation structurée, types de données, volumes de données

L'énoncé du problème : Concevez (i.e. réfléchissez aux structures de données utiles *puis* aux fonctions utiles... sans programmer une ligne encore) une procédure `voirChromosome` qui prend en entrée un chromosome en tant que séquence sous la forme :

- soit (lorsque le cours sur la lecture de fichier aura eu lieu) d'un fichier de nom `f` qui contient une suite de caractères `A`, `T`, `G` ou `C`,
- soit directement d'une chaîne de caractères `c` ne contenant que des caractères `A`, `T`, `G` ou `C`,

a priori le nombre de caractères est donc très grand. La procédure `voirChromosome` affiche le chromosome sur un écran de 100 caractères de large avec :

- une première ligne indiquant le pourcentage en `GC` du chromosome,
- puis le chromosome lui même avec une numérotation des lignes commençant à 1,
- et le pourcentage en `GC` de chaque ligne du chromosome (indiqué en fin de ligne).

Voici un exemple de début de sortie d'écran :

```
Pourcentage en GC = 56 %

1: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG (50%)
2: ATATATATATATATATATATGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGC (75%)
3: .......etc.....
```

La démarche : on commence par évaluer les besoins des utilisateurs :

- le nombre approximatif de nucléotides par ligne
- le nombre de lignes incroyablement grand
- l'intérêt de calculer le pourcentage global à partir des pourcentages locaux en terme de temps de calcul
- le coût de mettre le pourcentage global de GC en premier plutôt qu'à la fin...
- l'intérêt de savoir si mono- ou multi-utilisateur sur une station
- les conséquences du choix mono-utilisateur avec beaucoup de mémoire en rabe

... et on en arrive au choix des structures de données :

- liste de chaînes de caractères pour stocker les lignes
- liste d'entiers pour stocker les pourcentages locaux
- entiers pour les numéros de ligne, le calcul des pourcentages, *etc.*

... qui implique les fonctions intermédiaires (et pas le contraire) :

- fonction de découpage de `c` en tranches de 80 caractères (et intérêt de définir une variable égale à 80) → fonction `coupe`
- fonction de calcul du pourcentage en GC d'un brin → fonction `pourcent`
- fonction de remplissage de la liste des pourcentages à partir de la liste des brins de 80 caractères → fonction `mapPourcent`
- fonction de moyenne de la liste des pourcentages → fonction `moyenne`
- procédure principale pour gérer l'IHM → procédure `voirChromosome`

Cela conduit à la suite du TD :

Exercice 1 : En supposant les fonctions précédentes déjà écrites, écrivez la procédure `voirChromosome`.

Réaliser les fonctions utiles :

Exercice 2 : Écrivez une fonction `coupe` qui prend en entrée un entier `n` et un nom de fichier `f` (ou une chaîne de caractères `c`) représentant un chromosome et fournit en sortie une liste dont les éléments :

- sont des chaînes de caractères de longueur exactement `n`, sauf le dernier qui peut être plus court
- qui représentent des brins consécutifs d'ADN tels que leur concaténation soit égale au chromosome `c`.

Exercice 3 : Écrivez la fonction `pourcent` qui prend en entrée un brin `b` (c'est-à-dire une chaîne de caractères contenant des `ATGC`) et retourne son pourcentage (entier) en `GC`.

Exercice 4 : Écrivez la fonction `mapPourcent` qui prend en entrée une liste de brins `l` et retourne la liste des pourcentages en `GC` correspondants (donc une liste de même taille).

Exercice 5 : Écrivez la fonction `moyenne` qui fournit la moyenne (entière) d'une liste d'entiers.

Fichiers, dictionnaires et listes

On suppose que l'on mène une expérience (d'une durée inférieure à une journée) au cours de laquelle on mesure de temps en temps le niveau d'expression d'un certain nombre de gènes. Chaque gène porte un nom (par exemple `Yea302` ou `TEL1` ou tout autre chaîne de caractère) et son niveau d'expression a été mesuré en divers instants, partant du début de l'expérience en `00h00` et ensuite à des temps pas nécessairement régulièrement espacés (par exemple `00h37` puis `01h53` puis `05h00` puis `13h21`, *etc*).

Les temps exacts de mesure ne sont pas obligatoirement les mêmes d'un gène à un autre (sauf si les résultats sont issus d'une unique plaque de puce à ADN mais ce n'est pas toujours le cas). En revanche, les mesures sont supposées toutes de type `float`, disons comprises entre 0 et 100.

L'ensemble des mesures est géré par un système automatisé et ce système produit un fichier classé par gène. Pour chaque gène, une première ligne donne son nom puis les divers temps où ce gène a été mesuré avec succès sont fournis avec la valeur de la mesure effectuée, à raison de une mesure par ligne. À la fin d'un gène, le gène suivant commence sans laisser de ligne vide entre les deux. Le fichier global à l'issue de l'expérience est alors d'une forme comme l'exemple suivant :

```
nom = Yea302
00h00 : 6.3
01h21 : 1.4
02h32 : 3.0
02h40 : 1.4
03h35 : 0.7
06h45 : 5.6
07h50 : 0.7
09h12 : 9
nom = TEL1
00h00 : 2.4
05h28 : 0.6
12h02 : 3.9
12h48 : 5.5
20h51 : 10.2
nom = SRO77
00h00 : 36
03h08 : 55.2
10h00 : 59.8
15h33 : 100
18h25 : 100
```

Exercice : Écrivez une fonction `lireProfils` qui prend en entrée l'adresse d'un fichier produit par une expérience comme décrite au-dessus, et retourne en sortie une liste de dictionnaires construite de telle sorte que : (1) la liste contient un dictionnaire par gène mesuré durant l'expérience, et (2) chaque dictionnaire contient un élément `"nom"` dont la donnée associée est le nom du gène ainsi qu'autant d'autres éléments que de temps de mesure : `"00h00"`, `"01h21"`, `"02h32"`, *etc.* dont la donnée associée est la mesure de type `float` correspondante.

Par exemple, si le fichier précédent porte le nom `mesures.txt` alors la commande `lireProfils("mesures.txt")` fournit la liste de dictionnaires suivante :

```
[  { "nom":"Yea302", "00h00":6.3, "01h21":1.4, "02h32":3.0, "02h40":1.4, "03h35":0.7,
"06h45":5.6, "07h50":0.7, "09h12":9.0 }  ,  { "nom":"TEL1", "00h00":2.4, "05h28":0.6,
"12h02":3.9, "12h48":5.5, "20h51":10.2 }  ,  { "nom":"SRO77", "00h00":36.0, "03h08":55.2,
"10h00":59.8, "15h33":100.0, "18h25":100.0 }  ]
```

Nota : sauvegardez cette fonction pour pouvoir la réutiliser lors des prochains TD.

Fichiers et dictionnaires (suite du précédent TD)

Le but de ce TD est d'écrire la procédure « inverse » de la procédure de lecture de profils d'expression faite au TD précéden (`lireProfils`). Il s'agit maintenant d'écrire dans un fichier à partir d'une liste de dictionnaires représentant des profils d'expression de gènes. Par exemple en partant de la liste de dictionnaires suivante :

```
[  { "nom":"Yea302", "00h00":6.3, "01h21":1.4, "02h32":3.0, "02h40":1.4, "03h35":0.7,
"06h45":5.6, "07h50":0.7, "09h12":9.0 }  ,  { "nom":"TEL1", "00h00":2.4, "05h28":0.6,
"12h02":3.9, "12h48":5.5, "20h51":10.2 }  ,  { "nom":"SRO77", "00h00":36.0, "03h08":55.2,
"10h00":59.8, "15h33":100.0, "18h25":100.0 }  ]
```

On voudra écrire dans un fichier les lignes suivantes :

```
nom = Yea302
00h00 : 6.3
01h21 : 1.4
02h32 : 3.0
02h40 : 1.4
03h35 : 0.7
06h45 : 5.6
07h50 : 0.7
09h12 : 9
nom = TEL1
00h00 : 2.4
05h28 : 0.6
12h02 : 3.9
12h48 : 5.5
20h51 : 10.2
nom = SRO77
00h00 : 36
03h08 : 55.2
10h00 : 59.8
15h33 : 100
18h25 : 100
```

Nota : *sauvegardez toutes vos fonctions pour pouvoir les réutiliser lors du prochain TD.*

Exercice 1 : Le premier problème rencontré est que les dictionnaires « mélangent » les champs et les énumèrent donc dans n'importe quel ordre. Commencez par le vérifier en examinant le résultat de la fonction `lireProfils` écrite au précédent TD. Profitez-en pour définir une variable globale `yea` contenant le premier dictionnaire de la liste (celui de nom `Yea302`) ; ce sera utile pour tester les exercices suivants.

Exercice 2 : Pour résoudre ce problème d'ordre, écrivez une fonction `horloge` qui prend en entrée un dictionnaire `d` comme précédemment, et fournit en sortie la liste triée des temps du dictionnaire.
INDICATION : `sorted` est une fonction qui prend une liste en entrée et retourne cette liste triée par ordre croissant.

Exercice 3 : Écrivez une procédure `ecrireProfils` qui prend en entrée un nom de fichier `f` et une liste de dictionnaires `ld` représentant des profils d'expression comme dans le précédent TD, et crée le fichier de nom `f` contenant successivement les profils d'expression, comme dans l'exemple précédent, avec les temps en ordre croissant pour chaque gène.

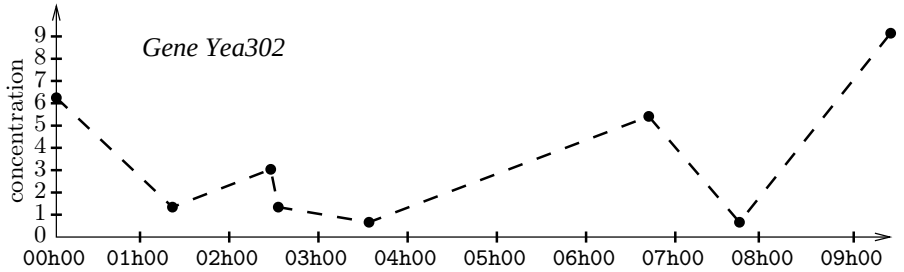
Exercice 4 : Écrivez une fonction `horaire` qui prend en entrée deux nombres entiers `X` et `Y`, qui fournit un message d'erreur si `X` n'est pas une heure valide (c'est-à-dire si elle n'est pas comprise entre 0 et 23) ou si `Y` n'est pas un nombre de minutes valide (c'est-à-dire compris entre 0 et 59), et qui retourne dans tous les autres cas la chaîne de caractères normalisée correspondante. Par exemple `horaire(2,28)` retourne `"02h28"` et `horaire(19,7)` retourne `"19h07"`.

Automates, dictionnaires et listes

Le but de ce TD est de repérer les gènes covariants dans une liste de profils d'expression similaire à celles manipulées dans les TD précédents. Pour cela on veut utiliser la fonction `coexprime` du TD n°4.

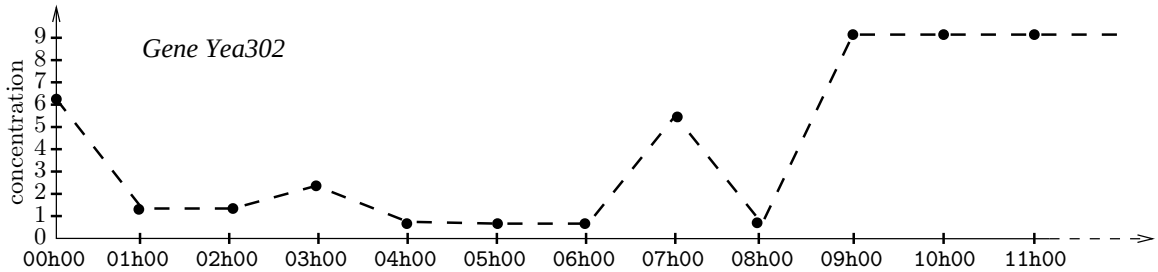
La difficulté vient du fait que nos profils d'expressions ne sont pas tous mesurés aux mêmes horaires d'un gène à l'autre, alors que la fonction `coexprime` prend en entrée deux listes de niveaux d'expressions qui représentent des mesures faites aux *mêmes heures*.

Il faut donc commencer par « normaliser » chaque profil d'expression pour avoir un niveau d'expression « théorique » pris aux mêmes heures pour chaque gène. On va calculer ce niveau théorique pour chaque heure exacte. Par exemple, le dictionnaire `yea` du TD précédent représente le profil d'expression ci-dessous :




```
{ "nom":"Yea302", "00h00":6.3, "01h21":1.4, "02h32":3.0, "02h40":1.4, "03h35":0.7,
"06h45":5.6, "07h50":0.7, "09h12":9.0 }
```

On voit bien que les mesures ne sont pas prises à chaque heure. On va donc « lisser » ce profil d'expression comme suit :



Ce profil d'expression « théorique » est obtenu selon le principe de l'exercice ci-dessous.

Exercice 1 : Écrivez une fonction `normalise` qui prend en entrée un dictionnaire `d` représentant un profil d'expression quelconque et le « lisse » en donnant une liste de mesures « théoriques » avec une mesure pour chaque heure (`"00h00"`, `"01h00"`, ..., `"23h00"`). On procède de la manière suivante :

- On fournit un message d'erreur si le dictionnaire de départ ne possède aucune mesure comprise entre les temps `"00h00"` et `"00h30"`, sinon la première mesure de la liste (temps 0) est la moyenne des mesures comprises entre ces deux temps.
- Ensuite, pour chaque heure `X`, on associe la moyenne des valeurs du dictionnaire de départ entre les temps `"(X-1)h30"` et `"Xh30"`.
  Il peut arriver cependant qu'il n'y ait aucune mesure entre ces deux temps. Dans ce cas, on reporte la moyenne précédente (celle de `X-1`).
- La liste retournée par la fonction `normalise` contiendra 24 mesures théoriques (de `"00h00"` à `"23h00"`).

INDICATION : *Il faut raisonner en utilisant la notion d'automate...*

Note : cet algorithme de lissage est peu performant, comme on le voit en comparant les courbes données en exemple ; n'hésitez pas à en inventer de nouvelles versions plus crédibles s'il vous reste du temps ou en exercice à la maison.

Exercice 2 : Écrivez la fonction `cluster` qui prend en entrées :

- d'une part, un dictionnaire `ref` pour un gène de référence,
- et d'autre part, une liste de dictionnaires `ld` représentant une liste de profils d'expression,

et fournit en sortie la liste des noms des gènes co-exprimés avec `ref`.

Révisions...

Programmation 1 **Examen du 16 avril 2014** GB3, 2^{nd} semestre 2013-2014

Durée = 2h30. Langage Python.
Inscrivez **lisiblement** vos NOM et Prénom en tête de la copie double qui vous a été distribuée ;
ensuite cachetez le coin de la copie double ;
enfin inscrivez EN GROS CHIFFRES sur la copie double le numéro suivant : nUmErO

VOUS DEVEZ INSCRIRE VOS RÉPONSES DIRECTEMENT SUR CETTE FEUILLE D'ÉNONCÉ, QUE VOUS PLACEREZ À L'INTÉRIEUR DE LA COPIE DOUBLE AVANT DE LA RENDRE.

Ordinateurs, téléphones et autres moyens de communication sont interdits.

Exercice 1 : Quel est le résultat des 8 expressions suivantes, et, *si applicable*, quel est leur type respectif ?

(1) `True == False` **(2)** `(5+4) < (5*2)` **(3)** `not("True" and "False")` **(4)** `print("True" + "False")`

.

(5) `(10/3)>(11//3)` **(6)** `(10+4) % 5` **(7)** `"abcd"[2]` **(8)** `"Fait %s à %i degrés" % ("chaud",31)`

.

Exercice 2 : Écrivez une fonction `nucleotide` qui prend en entrées une chaîne de caractères `c` et un booléen `b` et qui retourne un booléen qui dit si `c` est un nucléotide reconnu. Lorsque `b` vaut `True` il s'agit d'ADN (`A`, `T`, `G` ou `C` reconnus) et lorsque `b` vaut `False` il s'agit d'ARN (`A`, `U`, `G` ou `C` reconnus). Si la chaîne `c` ne contient pas qu'un seul caractère, on écrit une erreur à l'écran.

.

.

.

.

.

.

.

Exercice 3 : Écrivez une fonction `nbRepetitions` qui prend en argument une chaîne de caractères `b` qui représente un brin d'ADN, et retourne le nombre de fois où un nucléotide de ce brin est le même que celui qui le précède. Par exemple `nbRepetitions("ATTGCCCTC")` retourne `3`. On supposera sans le vérifier que `b` est réellement un brin d'ADN (i.e. ne contient que des `A`, `T`, `G` ou `C`).

.

.

.

.

.

.

.

.

Exercice 4 : Écrivez une fonction `codons` qui prend en entrées une chaîne de caractères `b` représentant un brin d'ARN et un entier `p` (compris entre 1 et 3) représentant la phase de lecture, et qui retourne la liste de ses codons pour cette phase de lecture (on suppose qu'il n'y a jamais de décalage de phase en cours de lecture). Par exemple `codons("AUAGCCUAAGCCUUC",2)` retourne la liste `["UAG","CCU","AAG","CCU"]` ; comme on le voit, les derniers nucléotides sont ignorés (ici `UC`).

.

.

.

.

.

.

.

T.S.V.P. ⟶

Exercice 5 : Écrivez une fonction `complementaire` qui prend en entrée une chaîne de caractères `b` et qui écrit une erreur à l'écran si `b` n'est pas un brin d'ADN ou sinon retourne le brin complémentaire. ATTENTION, pour écrire cette fonction, on utilisera obligatoirement le dictionnaire `d={"A":"T", "T":"A", "G":"C", "C":"G"}` et aucun appel à une sous-fonction n'est nécessaire.

.

.

.

.

.

.

.

.

.

.

.

Exercice 6 : Écrivez une fonction `lireGenes` qui prend en entrée un nom de fichier `f` tel que le fichier existe et contient la description d'un ensemble de gènes avec pour chaque gène son nom et les exons qui le composent, en suivant la structure donnée en exemple ci-dessous. Le résultat de la fonction est alors une liste de dictionnaires. Chaque dictionnaire de la liste contient deux éléments : `"nom"` qui a pour donnée associée le nom du gène et `"exons"` qui a pour donnée associée la liste des exons fournis dans le fichier (il est possible de trouver plusieurs fois un même nom de gène en raison de l'épissage alternatif, comme c'est le cas dans l'exemple).

Contenu de fichier	Liste de dictionnaires obtenue

`NOM Yea38`
`exon ATTAGGCAA`
`exon CCGCCTTA`
`exon GTGTGTAT`
`NOM ST67`
`exon GGGAATGCTCATAATGCC`
`NOM Yea38`
`exon ATTAGGCAA`
`exon GCGCGCATATAT`
`exon GTGTGTAT`
`NOM YRT`
`exon ATGTGTGAT`
`exon ATGCCGGCCT`
`exon AGGGTGA`
`exon TAAATAAA`
	`[ {"nom":"Yea38", "exons":["ATTAGGCAA","CCGCCTTA","GTGTGTAT"]} ,`
	`{"nom":"ST67", "exons":["GGGAATGCTCATAATGCC"]} ,`
	`{"nom":"Yea38", "exons":["ATTAGGCAA","GCGCGCATATAT","GTGTGTAT"]} ,`
	`{"nom":"YRT", "exons":["ATGTGTGAT","ATGCCGGCCT","AGGGTGA","TAAATAAA"]} ]`

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

Les structures de données

Une structure de données est définie par quatre choses :

1. *Un type* qui est simplement un nom permettant de classifier les expressions syntaxiques relevant de cette structure de donnée.
2. *Un ensemble* qui définit avec précision quelles sont les *valeurs pertinentes* de ce type.
3. *La liste des opérations* que l'ordinateur peut utiliser pour effectuer des « calculs » sur les valeurs précédentes. Cela comprend le *nom* de l'opération et comment l'utiliser, c'est-à-dire quels types elle accepte en *arguments* et le type du *résultat*.
4. *La sémantique des opérations* c'est-à-dire une description précise du résultat fourni en fonction des arguments.

Il existe en Python une commande `type`, qui prend en argument une valeur ou une variable quelconque et fournit comme résultat le type de cette valeur.

Fiche n°1
La structure des booléens

1 Le type

Le nom du type des booléens est `bool`, du nom de George Boole [1815-1864] : mathématicien anglais qui s'est intéressé aux propriétés algébriques des valeurs de vérité.

2 L'ensemble des valeurs

Il s'agit d'un ensemble de seulement 2 données pertinentes, dites *valeurs de vérité* : {`True`,`False`}.

3 Les opérations

Opérations booléennes les plus courantes sont :

Opération	Entrée	Sortie
`not`	`bool`	`bool`
`and`	`bool`×`bool`	`bool`
`or`	`bool`×`bool`	`bool`

4 La sémantique des opérations

De manière similaire aux tables de multiplications, on écrit facilement les tables de ces fonctions puisque le type est fini (et petit).

not	True	False
	False	True

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

5 Exemples

Exemples de calculs sur les booléens :

```
>>> True or False
True
>>> true or false
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
>>> True and False
False
>>> not(True)
False
>>> type(True)
<type 'bool'>
>>> True and
  File "<stdin>", line 1
    True and
           ^
SyntaxError: invalid syntax
```

Comme on le voit :
- l'opération `type` écrit bien le type d'une donnée ;
- la syntaxe est stricte, un oubli de majuscule suffit à faire une erreur ;
- ces calculs ont peu d'intérêt en soi, mais deviennent indispensables lorsqu'on veut manipuler des conditions sur les types plus élaborés :

```
>>> if (5 > 2+3) or (8 == 2 ** 3) :
...   print("OK!")
... else :
...   print("Faux!")
...
OK!
```

La structure des entiers relatifs

1 Le type

Le type est appelé `int` (comme « integers »).

2 L'ensemble des valeurs

Théoriquement le type `int` correspond à l'ensemble $\mathbb{Z}$ des mathématiques. En fait il est borné en Python, avec borne dépendante de la machine, mais assez grande pour ignorer ce fait ici.

3 Les opérations

Les opérations les plus courantes sont :

Opération	Entrée	Sortie
`+` `-` `*` `//` `%` `**`	`int`×`int`	`int`
`==` `<` `>` `<=` `>=` `!=`	`int`×`int`	`bool`

Les opérations `==` `<` `>` `<=` `>=` `!=` sont appelées *les comparaisons* et se retrouveront dans d'autres structures de données également.

4 La sémantique des opérations

Il s'agit respectivement des addition, soustraction, multiplication, division « euclidienne », modulo et puissance, qui sont habituelles sur les entiers relatifs. La division euclidienne fournit toujours un résultat entier (le quotient) et le modulo n'est autre que le reste de cette division.

Noter que la comparaison d'égalité se note `==` et non pas `=` pour ne pas la confondre avec l'affectation de variable en Python. Les comparaisons `<=` `>=` `!=` sont respectivement les versions accessibles au clavier de $\leq$, $\geq$ et $\neq$.

5 Exemples

Rien de bien compliqué :

```
>>> type(46)
<type 'int'>
>>> 5 * 7
35
>>> print(5 * 7)
35
>>> 5 // 2
2
>>> 5 % 2
1
>>> 5 ** 3
125
```

et les comparaisons :

```
>>> 5 < 3
False
>>> 5 == 3 + 2
True
>>> 5 = 3 + 2
```

```
  File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> 5 <= 5
True
```

La structure des nombres réels

1 Le type

On dit aussi « les nombres flottants » et le nom du type est `float`.

2 L'ensemble des valeurs

`float` représente l'ensemble des nombres réels $\mathbb{R}$, avec cependant une précision limitée par la machine, mais les erreurs seront négligeables dans le cadre de ce cours.

3 Les opérations

Les opérations les plus courantes sont :

Opération	Entrée	Sortie
`+ - * / **`	`float`×`float`	`float`
`int`	`float`	`int`
`float`	`int`	`float`
`== < > <= >= !=`	`float`×`float`	`bool`

4 La sémantique des opérations

Ce sont les opérations habituelles, avec cette fois une division exacte et donc pas de modulo puisqu'il n'y a pas de reste.

5 Exemples

```
>>> type(46.8)
<type 'float'>
>>> 5.0 / 2.0
2.5
>>> 7.5 * 2
15.0
>>> int(7.5 * 2)
15
>>> int(7.75)
7
>>> float(3)
3.0
```

Noter la différence entre `/` et `//`.

La structure des chaînes de caractères

1 Le type

Le type des chaînes de caractères est appelé `str` en Python (et `string` dans presque tous les autres langages de programmation).

2 L'ensemble des valeurs

Il existe 256 « *caractères* » qui couvrent à peu près tout ce que l'on peut taper au clavier en une fois, Une *chaîne de caractères*, comme son nom l'indique, est une suite de caractères (de longueur aussi grande que nécessaire, avec les limitations habituelles d'occupation mémoire qui ne seront pas atteintes dans le cadre de ce cours).

Une chaîne de caractères doit être délimitée par des guillemets comme dans `"toto"`.

3 Les opérations

Les opérations les plus simples sont :

Opération	Entrée	Sortie
`+` `%` ...	`str`×`str`	`str`
`len`	`str`	`int`
`==` `<` `>` `<=` `>=` `!=`	`str`×`str`	`bool`

Plus généralement, l'opération `%` peut avoir plusieurs types d'entrées ; voir ci-dessous.

Il existe d'autres opérations sur les chaînes de caractères ; charge à vous de compléter cette fiche lorsque nous rencontrerons de nouvelles opérations dans les cours ou les TD, en fonction des besoins.

4 La sémantique des opérations

Le `+` dénote la concaténation (mise bout à bout) des chaînes de caractères ; `len` est la longueur (c'est-à-dire le nombre de caractères) d'une chaîne ; les comparaisons sont les comparaisons alphabétiques inspirées de celles d'un dictionnaire français ou anglais, sachant que les chiffres et ponctuations viennent avant les lettres, et que les majuscules viennent avant les minuscules.

« `%` » est l'opération de *substitution*. Si la chaîne de caractères s_0 contient « `%s` » et si s_1 est une autre chaîne de caractères, alors s_0 `%` s_1 est la chaîne obtenue en remplaçant dans s_0 les deux caractères `%s` par la chaîne s_1.

S'il y a plusieurs « `%s` », il faut mettre entre parenthèses autant de chaînes (`(`s_1`,...,`s_n`)`) après l'opération `%`.

S'il y a un « `%d` » ou un « `%i` » au lieu de « `%s` » il faut mettre un entier relatif à la place d'une chaîne (`d` comme décimal, `i` comme integer et `s` comme string).

S'il y a un « `%f` » (`f` comme float) il faut mettre un nombre réel. On peut imposer le nombre de chiffres après la virgule : « `%.4f` » par exemple limite la précision à 4 chiffres après la virgule.

L'opération de substitution `%` admet d'autres types de substitutions ; celles décrites dans cette fiche sont seulement les plus utiles.

5 Exemples

```
>>> "toto" + "tutu"
'tototutu'
>>> len("toto")
4
>>> "toto" < "tutu"
True
>>> "ab" < "aab"
False
```

```
>>> "a b" == "ab"
False
```

Noter l'absence d'espace entre `"toto"` et `"tutu"` après concaténation.

Et pour la substitution :

```
>>> "bonjour %s ; il faut beau aujourd'hui." % "Pierre Dupond"
"bonjour Pierre Dupond ; il faut beau aujourd'hui."
>>> nom = "Dupond"
>>> prenom="Pierre"
>>> genre = "homme"
>>> "Ce %s s'appelle %s et c'est un %s" % (prenom,nom,genre)
"Ce Pierre s'appelle Dupond et c'est un homme"
>>> age = 41
>>> "%s %s a %i ans." % (prenom,nom,age)
'Pierre Dupond a 41 ans.'
>>> "%s %s mesure %fm." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.850000m.'
>>> "%s %s mesure %.2fm." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.85m.'
```