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 (par exemple *via* google) 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 Premiers calculs

Exercice 1 : Tapez les exemples vus en cours. À cette occasion, on découvre entre autres :

- que les accents ne sont pas bien gérés, mieux vaut les éviter
- 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
- *etc.*

Exercice 2 : 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.

3 Définition de la syntaxe des expressions booléennes

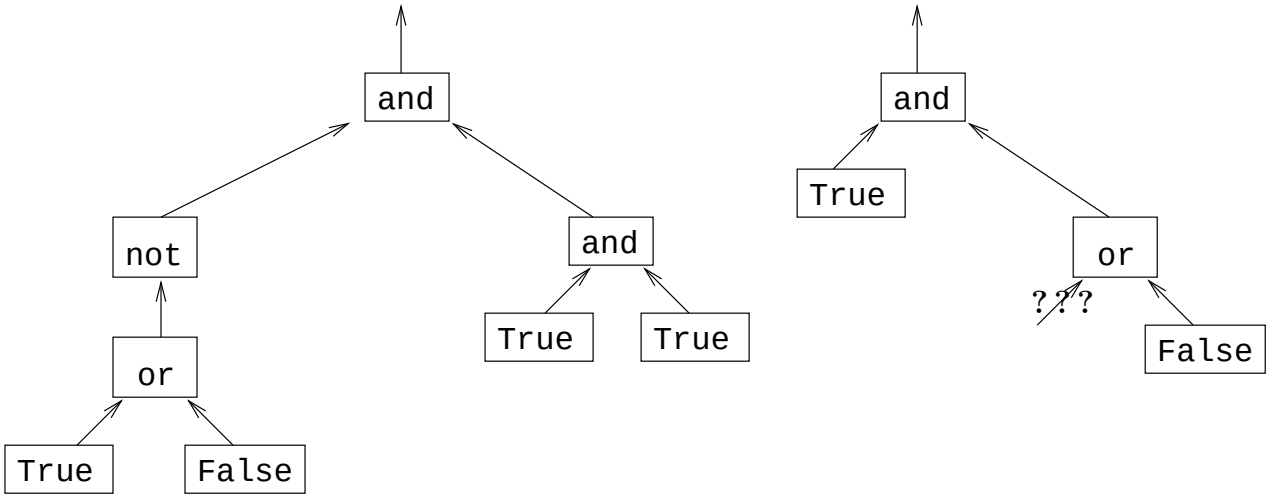
Un programme dans un langage donné est une suite de symboles, construite en respectant certaines règles. La *syntaxe* explique comment sont construits les mots et les phrases valides du langage (les expressions), la *sémantique* en donne le sens.

Ainsi, une expression booléenne est :

- une constante booléenne (`True` ou `False`)
- ou une expression unaire de la forme « `not` *exp* » où *exp* est une expression booléenne plus simple
- ou une expression binaire de la forme « *exp1* `and` *exp2* » où *exp1* et *exp2* sont des expressions booléennes plus simples
- ou une expression binaire de la forme « *exp1* `or` *exp2* » où *exp1* et *exp2* sont des expressions booléennes plus simples

Exemples d'expressions correctes avec ces règles : « `True` », « `True and False` », « `(True and False) or True` ». Expressions incorrectes : « `True and` », « `True and or False` » ou « `True not False` ». Elles comportent des *erreurs syntaxiques* parce qu'elles ne peuvent pas être construites en utilisant uniquement les règles précédentes.

Pour respecter ces règles, il faut pouvoir fabriquer un *arbre syntaxique* tel que, en chaque nœud de l'arbre, l'opération booléenne qui sera calculée en dernier en constitue la *racine*, et les expréssions qu'elle relient en sont les *sous-arbres*. Par exemple l'expression « `not(True or False) and (True and True)` » est correcte alors que « `True and or False` » ne l'est pas. En effet, elles conduisent respectivement aux formes d'arbre syntaxique suivantes :



Exercice 3 : Tracez les arbres syntaxiques des expressions précédentes. Vérifier que l'arbre est bien formé pour les expressions correctes, mal formé ou impossible à compléter sinon.

Un opérateur binaire peut être placé, comme ici, en position *infixe* c'est-à-dire, entre ses arguments. Pour certains opérateurs ou dans certains langages de programmation, il peut aussi être placé en position *préfixe* (devant ses arguments : « `or` (exp,exp) » ou plus rarement *postfixe* (derrière ses arguments : « exp exp `or` »). Dans le langage Python, `and` et `or` sont infixes mais les fonctions que vous programmerez vous-mêmes seront préfixes.

4 Priorité et parenthésage à gauche

Nous allons maintenant travailler avec des expressions arithmétiques que nous pouvons définir de la façon suivante. Une expression arithmétique est :

- un nombre entier relatif
- ou une expression binaire de la forme « *exp1* `+` *exp2* » où *exp1* et *exp2* sont des expressions arithmétiques
- ou une expression binaire de la forme « *exp1* `-` *exp2* » où *exp1* et *exp2* sont des expressions arithmétiques
- ou une expression binaire de la forme « *exp1* `*` *exp2* » où *exp1* et *exp2* sont des expressions arithmétiques
- ou une expression binaire de la forme « *exp1* `/` *exp2* » où *exp1* et *exp2* sont des expressions arithmétiques
- ou une expression binaire de la forme « *exp1* `%` *exp2* » où *exp1* et *exp2* sont des expressions arithmétiques
- ou enfin une expression binaire de la forme « *exp1* `**` *exp2* » où *exp1* et *exp2* sont des expressions arithmétiques

Exercice 4 : Tracez les arbres syntaxiques des expressions suivantes : « `(1 + (5 % 2)) - 4` », « `(5 ** 2) - (3 / 2)` » et « `(4 ** / 4) +` ».
Vérifiez pour chacune si l'arbre est bien formé (expressions correctes) ou mal formé et dans ce cas indiquez les malformations.

L'expression $2*3+4$ peut *a priori* être interprétée comme $A =$ « `(2 * 3) +4` » ou comme $B =$ « `2 * (3 +4)` ».

Exercice 5 : De combien de façon peut-on *a priori* évaluer l' expression `2*8/4+5` ?

Il y a donc des ambiguïtés si l'on ne met pas toutes les parenthèses. Une solution consisterait à ajouter des parenthèses pour lever l'ambiguïté mais, pour limiter l'usage de parenthèses dans la notation infixe, on peut définir un ordre d'utilisation, appelé *priorité* sur les opérateurs.

Une opération *op1* est prioritaire devant *op2* si *op1* « choisit » ses arguments avant *op2*. Par exemple, « * » est prioritaire devant « + », $2*3+4$ se lit donc $(2*3)+4$. Si l'on souhaite un groupement différent des arguments, il faut alors utiliser des parenthèses : $2*2+4$ vaut 8 alors que $2*(2+4)$ vaut 12. Si deux opérations ont même priorité, leurs arguments sont lus de gauche à droite. Par exemple, « * » et « / » ont même priorité donc $2*3/4$ signifie $(2*3)/4$.

En Python :

- les opérations sur les nombres ont la priorité, avec de plus `* / %` et `**` prioritaires sur $+$ et $-$
- les comparaisons viennent ensuite
- enfin des opérations booléennes
- les opérateurs de même priorité sont parenthèsés à gauche.

Exercice 6 : Inventez une demi-douzaine d'expressions dont le résultat, une fois calculé sous Python, permet de mettre en évidence ces priorités.

5 Premières fonctions et procédures en Python

Exercice 7 : 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 8 : 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 9 : Quels résultats donneraient respectivement les expressions `1 + carre(2)` et `1 + double(2)` ?

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

1 Fonctions et procédures en Python

Exercice 1 : 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 2 : 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 3 : 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 4 : 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!"`.

2 Chaînes de caractères

Exercice 5 : Écrivez la 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.

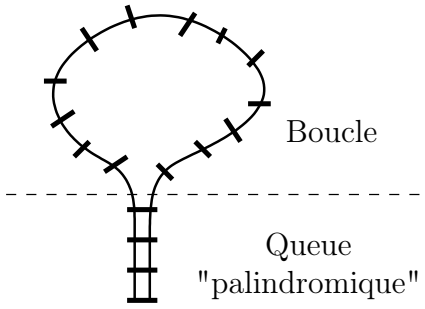
Exercice 6 : Écrivez la fonction `palindrome` qui prend en entrée une chaîne de caractères et fournit en sortie un booléen qui dit si c'est un palindrome.
On se contentera pour ce TD d'une première version simple qui consiste à comparer la chaîne avec son inverse (obtenu *via* l'exercice précédent).

Chaînes de caractères et palindromes

Exercice 1 : Écrivez la fonction `palindrome` qui prend en entrée une chaîne de caractères et fournit en sortie un booléen qui dit si c'est un palindrome.

- Relisez ou refaites la première version simple qui consiste à comparer la chaîne avec son inverse (calculé grâce à la fonction `inverse`).
- 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 2 : 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 **??** *(ce n'est pas vraiment facile... :-)).*

Exercice 3 : 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 4 : *(exercice subsidiaire)*

Écrivez une fonction `boucle2` qui étend la fonction `boucle` de l'exercice **??** 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 deux gènes G1 et G2, 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
```

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: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGG (50%)
2: ATATATATATATATATATATGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGCGC (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`.

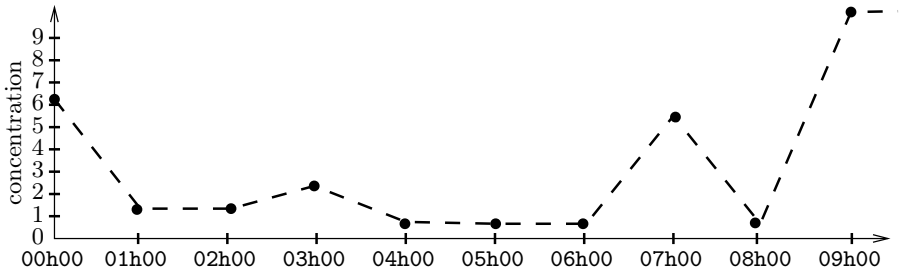
Exercice 3 : Écrivez la fonction `pourcent` qui prend en entrée un brin `b` (c'est-à-dire une chaîne de caractères contenant des `ATGC`) et retourne son pourcentage (entier) en `GC`.

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).

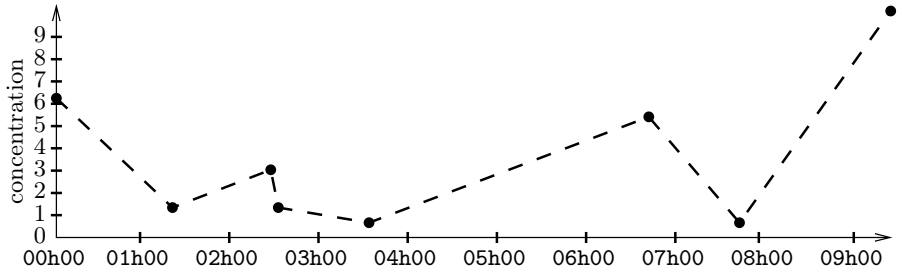
Exercice 5 : Écrivez la fonction `moyenne` qui fournit la moyenne (entière) d'une liste d'entiers.

Fonctions sur les dictionnaires et les listes

On veut exploiter des mesures de concentrations de protéines au sein d'une population cellulaire. Ces mesures varient au cours du temps, de sorte que pour chaque protéine, on peut reconstruire un « profil d'expression », c'est-à-dire une courbe de concentrations, du début de l'expérience (temps `00h00`) à la fin. Par exemple :



Malheureusement, les conditions expérimentales ne permettent pas de mesurer toutes les protéines aux mêmes temps, ni même à intervalles réguliers, de sorte que les courbes brutes expérimentales ressemblent plutôt à ça :



et l'objectif de ce TD et du suivant est principalement de « normaliser » les profils d'expression afin de les rendre comparables, en particulier avec une « mesure » toutes les heures entières, calculée par interpolation des mesures expérimentales.

On considère tout au long de ce TD (et du suivant) des dictionnaires représentant des mesures de type `float`, faites à divers temps comptés en heures et minutes à partir d'un top de départ `"00h00"` et sur une durée totale ne pouvant pas dépasser `"23h30"`. Par exemple :

```
{ "00h00":6.3 , "01h21":1.4 , "02h32":3.0 , "02h40":1.4 , "03h35":0.7 , "06h45":5.6 , "07h50":0.7 , "09h12":9.9 }
```

Ces mesures sont supposées être les niveaux de concentration d'une protéine tout au long d'une expérience. La chaîne de caractère représentant les temps sont normalisées en 5 caractères comme dans l'exemple, de sorte que deux temps peuvent être comparés simplement en tant que chaînes de caractères.

NOTE : sauvegardez vos fonctions pour pouvoir les réutiliser aux prochains TD.

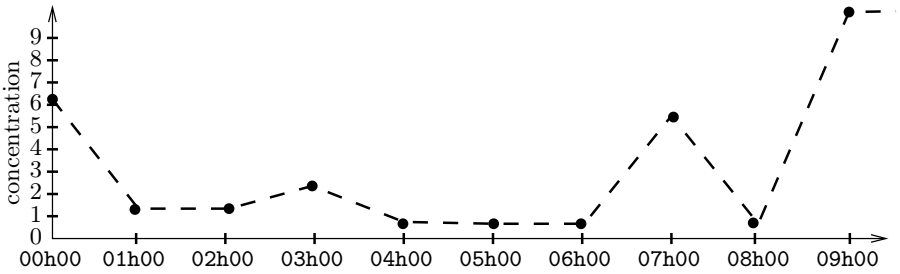
Exercice 1 : Le premier problème rencontré est que les dictionnaires « mélangent » les champs et les affichent donc dans n'importe quel ordre. Commencez par le vérifier en définissant une variable globale `profilBrut` contenant le dictionnaire précédent puis écrivez une fonction `horloge` qui prend en 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 2 : Écrivez une procédure `afficheProfil` qui à partir d'un dictionnaire représentant un profil, affiche à l'écran son contenu dans l'ordre croissant des temps.

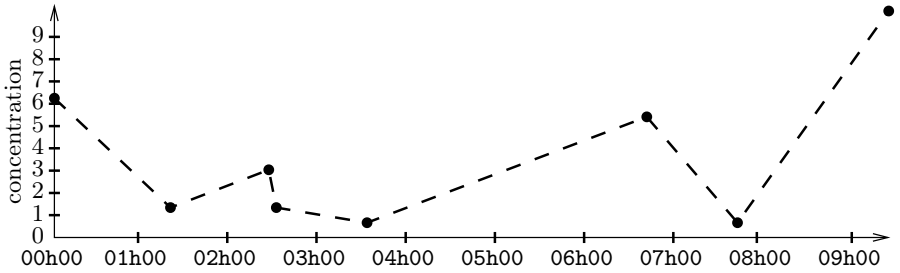
Exercice 3 : Écrivez une fonction `tempsmini` qui prend en entrée un dictionnaire et fournit en sortie la liste des temps où la protéine a atteint son minimum de concentration. Pour cet exercice, on admettra sans le vérifier que l'heure `"00h00"` appartient toujours au dictionnaire.

Fonctions sur les dictionnaires et les listes (suite)

On veut exploiter des mesures de concentrations de protéines au sein d'une population cellulaire. Ces mesures varient au cours du temps, de sorte que pour chaque protéine, on peut reconstruire un « profil d'expression », c'est-à-dire une courbe de concentrations, du début de l'expérience (temps `00h00`) à la fin. Par exemple :



Malheureusement, les conditions expérimentales ne permettent pas de mesurer toutes les protéines aux mêmes temps, ni même à intervalles réguliers, de sorte que les courbes brutes expérimentales ressemblent plutôt à ça :



et l'objectif de ce TD est de « normaliser » les profils d'expression afin de les rendre comparables, en particulier avec une « mesure » toutes les heures entières, calculée par une moyenne de mesures expérimentales.

On considère tout au long de ce TD (et du suivant) des dictionnaires représentant des mesures de type `float`, faites à divers temps comptés en heures et minutes à partir d'un top de départ `"00h00"` et sur une durée totale ne pouvant pas dépasser `"23h30"`. Par exemple :

```
{ "00h00":6.3 , "01h21":1.4 , "02h32":3.0 , "02h40":1.4 , "03h35":0.7 , "06h45":5.6 , "07h50":0.7 , "09h12":9.9 }
```

Ces mesures sont supposées être les niveaux de concentration d'une protéine tout au long d'une expérience. La chaîne de caractère représentant les temps sont normalisées en 5 caractères comme dans l'exemple, de sorte que deux temps peuvent être comparés simplement en tant que chaînes de caractères.

NOTE : sauvegardez vos fonctions pour pouvoir les réutiliser aux prochains TD.

Exercice 1 : Écrivez une fonction `horaire` qui prend en entrée deux nombres entiers `h` et `m`, qui fournit un message d'erreur si `h` n'est pas une heure valide (c'est-à-dire si elle n'est pas comprise entre 0 et 23) ou si `m` 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"`.

Exercice 2 : Écrivez une fonction `normalise` qui prend en entrée un dictionnaire et le « lisse » en donnant une unique mesure 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 mesure donnée en `"00h00"` est la moyenne des mesures comprises entre ces deux temps.
- Ensuite, pour chaque heure `"Xh00"`, on associe la moyenne des valeurs du dictionnaire de départ entre les temps `"(X-1)h30"` et `"Xh30"`, ou bien la moyenne précédente (celle de `"(X-1)h00"`) s'il n'y a aucune mesure entre ces deux temps.
- Le dictionnaire retourné par la fonction `normalise` contiendra donc 24 mesures lissées (de `"00h00"` à `"23h00"`).

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.

Fonctions sur les dictionnaires et sur les fichiers

On considère encore tout au long de ce TD des dictionnaires représentant des mesures réelles de (type `float`) faites à divers « temps » comptés en heures et minutes à partir d'un top de départ `"00h00"` et sur une durée totale ne pouvant pas dépasser `"23h30"`.
Par exemple : `{ "01h00":0.5 , "02h45":1.55 , "05h00":2.5 , "07h35":18.5 , "10h00":0.2 , "15h11":4.0 }`
Ces mesures sont supposées être les niveaux de concentration d'une protéine tout au long d'une expérience. La chaîne de caractère représentant les temps sont normalisées comme dans l'exemple, de sorte que deux temps peuvent être comparés simplement comme des chaînes de caractères.

NOTE : sauvegardez vos fonctions pour pouvoir les réutiliser.

Exercice 1 : Écrivez la fonction `chargement` qui prend en entrée le nom d'un fichier dont le contenu a une forme normalisée comme l'exemple suivant :

```
01h00 = 0.5
02h45 = 1.55
05h00 = 2.5
07h35 = 18.5
10h00 = 0.2
15h11 = 4.0
```

et retourne le dictionnaire correspondant. Remarquez à nouveau que, même si le fichier est trié par temps croissants, le dictionnaire ne l'est plus.
Ce fichier d'exemple est téléchargeable *via* la page web habituelle `www.i3s.unice.fr/~bernot/Enseignement/GB3_Python1`

Exercice 2 : Comprendre ce que fait la procédure très sommaire suivante, puis l'utiliser.

```
>>> def mkprofil (fich) :
...     t = "00h00"
...     f = open(fich,"w")
...     while len(t) == 5 :
...         v = input("Valeur au temps %s = " % t)
...         f.write("%s = %s\n" % (t,v))
...         t = input("Temps suivant = ")
...     f.close()
```

Critiquez l'ergonomie de cette procédure. Comment pourrait-on l'améliorer, en supposant par exemple que les mesures se font toutes les heures en partant du temps `00h00` ?

Exercice 3 : Écrivez la fonction `multichargement` qui prend en entrée le nom d'un fichier dont le contenu a une forme parfaitement normalisée comme la suivante :

```
01h00 = 0.5
02h45 = 1.55
05h00 = 2.5
07h35 = 18.5
10h00 = 0.2
15h11 = 4.0

00h00 = 8.2
05h28 = 14.5
06h14 = 0.5
10h00 = 0.4
11h55 = 0.3
24h00 = 0.2
48h30 = 0.1

01h00 = 10.5
02h45 = 11.55
05h00 = 12.5
07h35 = 28.2
10h00 = 10.2
15h11 = 34.5
```

et retourne la liste de dictionnaires correspondante. Par exemple ce fichier contient 3 profils d'expression : ils sont séparés par une ligne vide.

Exercice 4 : Ecrivez une procédure `mkmultiprofil` en s'inspirant d'un `mkprofil` amélioré. On supposera que tous les profils codés dans le fichier créé partagent exactement les mêmes horaires.

Fonctions sur les dictionnaires, encore...

On considère encore tout au long de ce TD des dictionnaires représentant des mesures réelles de (type `float`) faites à divers « temps » comptés en heures et minutes à partir d'un top de départ `"00h00"` et sur une durée totale ne pouvant pas dépasser `"99h59"`. Par exemple :

```
{ "00h00":0.9 , "01h00":0.5 , "02h45":1.55 , "05h00":2.5 , "07h35":18.5 , "10h00":0.2 , "15h11":4.0 }
```

Ces mesures sont supposées être les niveaux de concentration d'une protéine tout au long d'une expérience, ce qu'on appelle aussi un « profil d'expression » de la protéine. Les chaînes de caractères représentant les temps sont normalisées comme dans l'exemple, de sorte que deux temps peuvent être comparés simplement comme des chaînes de caractères.

Exercice 1 : Écrivez la fonction `covariantes` qui prend en entrées deux dictionnaires représentant des mesures effectuées (*simultanément*) sur deux protéines différentes et retourne un booléen disant si elles augmentent et diminuent en même temps (sans pour autant avoir les mêmes mesures).

Indications :
- Cette fonction appelle entre autres la fonction `horloge` des TD précédents.
- Si par exemple `h1` et `h2` sont deux horaires *consécutifs* des profils d'expression `d1` et `d2`, alors les deux protéines seront covariantes sur cet intervalle de temps si et seulement si : $(\texttt{d1}[\texttt{h2}] - \texttt{d1}[\texttt{h1}]) \times (\texttt{d2}[\texttt{h2}] - \texttt{d2}[\texttt{h1}])$ est positif ou nul (astuce mathématique pour assurer que les deux sens de variation sont de même signe).

Exercice 2 : Écrivez la fonction `cluster` qui prend en entrées :
- d'une part, un dictionnaire `ref` pour une protéine,
- et d'autre part, une liste de dictionnaires `l` représentant une liste de protéines mesurées simultanément avec la protéine de référence `ref`,

et fournit en sortie la liste des protéines covariantes avec `ref`.