Feuille d'exercices n° 3
Programmation Python 2

Exercice 3.1

Une chaîne d'ADN est *AT-riche* si la proportion de bases A et T est supérieure à 0.65.

1. Écrire la fonction `at_rich()` prenant une chaîne d'ADN en entrée et retournant vrai si la chaîne est *AT-riche*. On considérera que les bases peuvent être données en minuscules ou majuscules ;
2. Écrire le programme demandant une chaîne d'ADN à l'utilisateur et affichant le message « Elle est AT-riche » ou « Elle n'est pas AT-riche » suivant la circonstance.

Exercice 3.2

1. Écrire une fonction `bissextile()` prenant en entrée une année comprise entre 1583 et 2017 et retournant un booléen indiquant si elle est bissextile ou pas ;
2. Écrire un programme Python demandant en boucle à l'utilisateur une année et affichant si elle est bissextile ou non.

Exercice 3.3

1. Écrire une fonction `EcoRI()` prenant en entrée une chaîne d'ADN et retournant vrai si elle contient un site de restriction *EcoRI* (*GAATTC*) ;
2. Écrire un programme pour tester la fonction `EcoRI()`.

Exercice 3.4

On veut calculer le GC-ratio d'une séquence de base A, T, C, G, défini par :

$$\text{GC-ratio} = \frac{\#G + \#C}{\#A + \#T + \#G + \#C}$$

Écrire le programme Python prenant en paramètre un nom de fichier au format FASTA contenant une séquence de bases et retournant le GC-ratio de la séquence. On prendra soin de ne pas considérer les lignes du fichier commençant par la caractère « > ». On affichera aussi un message d'aide si le programme n'est pas appelé avec le bon nombre de paramètres et l'on retournera alors le code d'erreur 1.

Exercice 3.5

1. Écrire une fonction `common()` prenant en entrée deux listes d'entiers `L1` et `L2` et retournant une liste des éléments de `L1` se trouvant dans `L2` ;
2. Écrire un programme générant aléatoirement des listes `L1` et `L2` pour tester la fonction `common()`.

Exercice 3.6

Prenons un entier positif n et additionnons le carré de chacun des chiffres le constituant. Recommençons cette procédure jusqu'à ce que l'on obtienne 1 ou que l'on boucle sur un nombre déjà vu. Si l'on boucle sur 1, le nombre n est dit *joyeux*, sinon, il est *malheureux*.

1. Écrire la fonction `joyeux()` prenant un entier en entrée et retournant s'il est joyeux ou pas ;

2. Afficher les k premiers nombres joyeux, pour k fourni en paramètre au programme.

Exercice 3.7 (D'après Python for Biologists)

On appelle « *base ambigüe* » une base n'étant aucune des quatre bases *A*, *T*, *C* et *G*.

1. Écrire la fonction `first_ambiguous()` prenant en entrée une chaîne d'ADN et retournant la valeur et la position de la première base ambigüe ;
2. Écrire la fonction `all_ambiguous()` prenant en entrée une chaîne d'ADN et retournant la valeur et la position de toutes les bases ambigües ;
3. Écrire la fonction `polyAtail()` prenant en entrée une chaîne d'ARN et retournant la longueur de sa queue poly(A) (ou 0 s'il n'y a pas de queue poly(A)) ;
4. Tester les différentes fonctions créées dans un fichier `dna_re.py` de façon à ce que les tests ne soient exécutés que lorsque le programme est lancé sur la ligne de commande ;
5. Lancez l'interpréteur Python et chargez le module `dna_re.py`, puis cherchez la première base ambigüe dans la chaîne ADN se trouvant dans le fichier dna.dat (disponible sur madoc).

Exercice 3.8

1. En utilisant un dictionnaire comme dans l'exercice 3.7, écrire la fonction `complement()` prenant en entrée une chaîne d'ADN et retournant son complémentaire ;
2. Si la chaîne considérée contient des bases ambigües (voir exercice 3.7), la fonction `complement()` échouera. Écrire une fonction `robust_complement()` prenant en entrée une chaîne d'ADN et calculant son complémentaire. Si la chaîne contient des bases ambigües, la fonction retournera la chaîne complémentaire jusqu'à la première base ambigüe.Pour cela, on s'assurera de rattraper l'exception générée en cas de présence de base ambigüe. On importera le fichier `dna_re.py` créé dans l'exercice 3.7 pour utiliser la fonction `first_ambiguous()` qui y est définie.