10 Plus sur les chaînes de caractères

10.1 Préambule

Nous avons déjà abordé les chaînes de caractères dans les chapitres 2 *Variables* et 3 *Affichage*. Ici nous allons un peu plus loin, notamment avec les méthodes associées aux chaînes de caractères.

10.2 Chaînes de caractères et listes

Les chaînes de caractères peuvent être considérées comme des listes (de caractères) un peu particulières :

```
>>> animaux = "girafe tigre"
>>> animaux
'girafe tigre'
>>> len(animaux)
12
>>> animaux[3]
'a'
```

Nous pouvons donc utiliser certaines propriétés des listes comme les tranches :

```
>>> animaux = "girafe tigre"
>>> animaux[0:4]
'gira'
>>> animaux[9:]
'gre'
>>> animaux[:-2]
'girafe tig'
>>> animaux[1:-2:2]
'iaetg'
```

Mais *a contrario* des listes, les chaînes de caractères présentent toutefois une différence notable, ce sont **des listes non modifiables**. Une fois une chaîne de caractères définie, vous ne pouvez plus modifier un de ses éléments. Le cas échéant, Python renvoie un message d'erreur :

```
>>> animaux = "girafe tigre"
>>> animaux[4]
'f'
>>> animaux[4] = "F"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Par conséquent, si vous voulez modifier une chaîne de caractères, vous devez en construire une nouvelle. Pour cela, n'oubliez pas que les opérateurs de concaténation (`+`) et de duplication (`*`) (introduits dans le chapitre 2 *Variables*) peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne de caractères (voir plus bas).

10.3 Caractères spéciaux

Il existe certains caractères spéciaux comme `\n` que nous avons déjà vu (pour le retour à la ligne). Le caractère `\t` produit une tabulation. Si vous voulez écrire des guillemets simples ou doubles et que ceux-ci ne soient pas confondus avec les guillemets de déclaration de la chaîne de caractères, vous pouvez utiliser `\'` ou `\"`.

```
>>> print("Un retour à la ligne\npuis une tabulation\t puis un guillemet\"")
Un retour à la ligne
puis une tabulation     puis un guillemet"
>>> print('J\'affiche un guillemet simple')
J'affiche un guillemet simple
```

Vous pouvez aussi utiliser astucieusement des guillemets doubles ou simples pour déclarer votre chaîne de caractères :

```
>>> print("Un brin d'ADN")
Un brin d'ADN
>>> print('Python est un "super" langage de programmation')
Python est un "super" langage de programmation
```

Quand on souhaite écrire un texte sur plusieurs lignes, il est très commode d'utiliser les guillemets triples qui conservent le formatage (notamment les retours à la ligne) :

```
>>> x = """souris
... chat
... abeille"""
>>> x
'souris\nchat\nabeille'
>>> print(x)
souris
chat
abeille
```

Attention, les caractères spéciaux n'apparaissent intérprétés que lorsqu'ils sont utilisés avec la fonction `print()`. Par exemple, le `\n` n'apparait comme un saut de ligne que lorsqu'il est dans une chaîne de caractères passée à la fonction `print()` :

```
>>> "bla\nbla"
'bla\nbla'
```

```
>>> print("bla\nbla")
bla
bla
```

10.4 Préfixe de chaîne de caractères

Nous avons vu au chapitre 3 la notion de *f-string*. Il s'agit d'un mécanisme pour formater du texte au sein d'une chaîne de caractères. Par exemple :

```
>>> var = "f-string"
>>> f"voici une belle {var}"
'voici une belle f-string'
```

Que signifie le `f` que l'on accole aux guillements de la chaîne de caractères ? Celui-ci est appelé « préfixe de chaîne de caractères » ou *stringprefix*.

> **Remarque**
>
> Un *stringprefix* modifie la manière dont Python va interpréter la dite *string*. Celui-ci doit être systématiquement « collé » à la chaîne de caractères, c'est-à-dire pas d'espace entre les deux.

Il existe différents *stringprefixes* en Python, nous vous montrons ici les deux qui nous apparaissent les plus importants.

- Le préfixe `r` mis pour *raw string* qui force la non-interprétation des caractères spéciaux :

```
>>> s = "Voici un retour à la ligne\nEt là une autre ligne"
>>> s
'Voici un retour à la ligne\nEt là une autre ligne'
>>> print(s)
Voici un retour à la ligne
Et là une autre ligne
>>> s = r"Voici un retour à la ligne\nEt là une autre ligne"
>>> s
'Voici un retour à la ligne\\nEt là une autre ligne'
>>> print(s)
Voici un retour à la ligne\nEt là une autre ligne
```

L'ajout du `r` va forcer Python à ne pas interpréter le `\n` comme un retour à la ligne, mais comme un *backslash* littéral suivi d'un *n*. Quand on demande à l'interpréteur d'afficher cette chaîne de caractères, celui-ci met deux *backslashes* pour signifier qu'il s'agit d'un *backslash* littéral (le premier échappe le second). Finalement, l'utilisation de la syntaxe `r"Voici un retour à la ligne\nEt là une autre ligne"` renvoie une chaîne de caractères normale, puisqu'on voit ensuite que le `r` à disparu lorsqu'on demande à Python d'afficher le contenu de la variable `s`. Comme dans `var = 2 + 2`, d'abord Python évalue `2 + 2` et c'est ce résultat qui

est affecté à la variable `var`. Enfin, on notera que seule l'utilisation du `print()` mène à l'interprétation des caractères spéciaux comme `\n`, comme expliqué dans la rubrique précédente.

Les caractères spéciaux non interprétés dans les *raw strings* sont de manière générale tout ce dont le *backslash* modifie la signification, par exemple un `\n`, un `\t`, etc.

- Le préfixe `f` mis pour *formatted string* qui met en place l'écriture formattée comme vue au chapitre 3 *Affichage* :

```
>>> animal = "renard"
>>> animal2 = "poulain"
>>> s = f"Le {animal} est un animal gentil\nLe {animal2} aussi"
>>> s
'Le renard est un animal gentil\nLe poulain aussi'
>>> print(s)
Le renard est un animal gentil
Le poulain aussi
>>> s = "Le {animal} est un animal gentil\nLe {animal2} aussi"
>>> s
'Le {animal} est un animal gentil\nLe {animal2} aussi'
>>> print(s)
Le {animal} est un animal gentil
Le {animal2} aussi
```

La *f-string* remplace le contenu des variables situées entre les accolades et interprète le `\n` comme un retour à la ligne. Pour rappel, consultez le chapitre 3 si vous souhaitez plus de détails sur le fonctionnement des *f-strings*.

Conseils

Il existe de nombreux autres détails concernant les préfixes qui vont au delà de ce cours. Pour en savoir plus, vous pouvez consulter la documentations officielle.

10.5 Méthodes associées aux chaînes de caractères

Voici quelques méthodes spécifiques aux objets de type `str` :

```
>>> x = "girafe"
>>> x.upper()
'GIRAFE'
>>> x
'girafe'
>>> 'TIGRE'.lower()
'tigre'
```

Les méthodes `.lower()` et `.upper()` renvoient un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces méthodes n'altère pas la chaîne de caractères de départ mais renvoie une chaîne de caractères transformée.

Pour mettre en majuscule la première lettre seulement, vous pouvez faire :

```
>>> x[0].upper() + x[1:]
'Girafe'
```

ou plus simplement utiliser la méthode adéquate :

```
>>> x.capitalize()
'Girafe'
```

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la méthode `.split()` :

```
>>> animaux = "girafe tigre singe souris"
>>> animaux.split()
['girafe', 'tigre', 'singe', 'souris']
>>> for animal in animaux.split():
...     print(animal)
...
girafe
tigre
singe
souris
```

La méthode `.split()` découpe une chaîne de caractères en plusieurs éléments appelés *champs*, en utilisant comme séparateur n'importe quelle combinaison « d'espace(s) blanc(s) ».

Définition

Un espace blanc (*whitespace* en anglais) correspond aux caractères qui sont invisibles à l'œil, mais qui occupent de l'espace dans un texte. Les espaces blancs les plus classiques sont l'espace, la tabulation et le retour à la ligne.

Il est possible de modifier le séparateur de champs, par exemple :

```
>>> animaux = "girafe:tigre:singe::souris"
>>> animaux.split(":")
['girafe', 'tigre', 'singe', '', 'souris']
```

Attention, dans cet exemple, le séparateur est un seul caractères « `:` » (et non pas une combinaison de un ou plusieurs `:`) conduisant ainsi à une chaîne vide entre `singe` et `souris`.

Il est également intéressant d'indiquer à `.split()` le nombre de fois qu'on souhaite découper la chaîne de caractères avec l'argument `maxsplit` :

```
>>> animaux = "girafe tigre singe souris"
>>> animaux.split(maxsplit=1)
['girafe', 'tigre singe souris']
>>> animaux.split(maxsplit=2)
['girafe', 'tigre', 'singe souris']
```

La méthode `.find()`, quant à elle, recherche une chaîne de caractères passée en argument :

```
>>> animal = "girafe"
>>> animal.find("i")
1
>>> animal.find("afe")
3
>>> animal.find("z")
-1
>>> animal.find("tig")
-1
```

Si l'élément recherché est trouvé, alors l'indice du début de l'élément dans la chaîne de caractères est renvoyé. Si l'élément n'est pas trouvé, alors la valeur `-1` est renvoyée.

Si l'élément recherché est trouvé plusieurs fois, seul l'indice de la première occurrence est renvoyé :

```
>>> animaux = "girafe tigre"
>>> animaux.find("i")
1
```

On trouve aussi la méthode `.replace()` qui substitue une chaîne de caractères par une autre :

```
>>> animaux = "girafe tigre"
>>> animaux.replace("tigre", "singe")
'girafe singe'
>>> animaux.replace("i", "o")
'gorafe togre'
```

La méthode `.count()` compte le nombre d'occurrences d'une chaîne de caractères passée en argument :

```
>>> animaux = "girafe tigre"
>>> animaux.count("i")
2
>>> animaux.count("z")
0
>>> animaux.count("tigre")
1
```

La méthode `.startswith()` vérifie si une chaîne de caractères commence par une autre chaîne de caractères :

```
>>> chaine = "Bonjour monsieur le capitaine !"
>>> chaine.startswith("Bonjour")
True
>>> chaine.startswith("Au revoir")
False
```

Cette méthode est particulièrement utile lorsqu'on lit un fichier et que l'on veut récupérer certaines lignes commençant par un mot-clé. Par exemple dans un fichier PDB, les lignes contenant les coordonnées des atomes commencent par le mot-clé `ATOM`.

Enfin, la méthode `.strip()` permet de « nettoyer les bords » d'une chaîne de caractères :

```
>>> chaine = "  Comment enlever les espaces au début et à la fin ?       "
>>> chaine.strip()
'Comment enlever les espaces au début et à la fin ?'
```

La méthode `.strip()` enlève les espaces situés sur les bords de la chaîne de caractère mais pas ceux situés entre des caractères visibles. En réalité, cette méthode enlève n'importe quel combinaison « d'espace(s) blanc(s) » sur les bords, par exemple :

```
>>> chaine = "  \tfonctionne avec les tabulations et les retours à la ligne\n"
>>> chaine.strip()
'fonctionne avec les tabulations et les retours à la ligne'
```

La méthode `.strip()` est très pratique quand on lit un fichier et qu'on veut se débarrasser des retours à la ligne.

10.6 Extraction de valeurs numériques d'une chaîne de caractères

Une tâche courante en Python est de lire une chaîne de caractères (provenant par exemple d'un fichier), d'extraire des valeurs de cette chaîne de caractères pour ensuite les manipuler.

On considère par exemple la chaîne de caractères `val` :

```
>>> val = "3.4 17.2 atom"
```

On souhaite extraire les valeurs `3.4` et `17.2` pour ensuite les additionner.

Dans un premier temps, on découpe la chaîne de caractères avec la méthode `.split()` :

```
>>> val2 = val.split()
>>> val2
['3.4', '17.2', 'atom']
```

On obtient alors une liste de chaînes de caractères. On transforme ensuite les deux premiers éléments de cette liste en *floats* (avec la fonction `float()`) pour pouvoir les additionner :

```
>>> float(val2[0]) + float(val2[1])
20.599999999999998
```

Remarque

Retenez bien l'utilisation des instructions précédentes pour extraire des valeurs numériques d'une chaîne de caractères. Elles sont régulièrement employées pour analyser des données extraites d'un fichier.

10.7 Conversion d'une liste de chaînes de caractères en une chaîne de caractères

On a vu dans le chapitre 2 *Variables* la conversion d'un type simple (entier, *float* et chaîne de caractères) en un autre avec les fonctions `int()`, `float()` et `str()`. La conversion d'une liste de chaînes de caractères en une chaîne de caractères est particulière puisqu'elle fait appelle à la méthode `.join()`.

```
>>> seq = ["A", "T", "G", "A", "T"]
>>> seq
['A', 'T', 'G', 'A', 'T']
>>> "-".join(seq)
'A-T-G-A-T'
>>> " ".join(seq)
'A T G A T'
>>> "".join(seq)
'ATGAT'
```

Les éléments de la liste initiale sont concaténés les uns à la suite des autres et intercalés par un séparateur qui peut être n'importe quelle chaîne de caractères. Ici, on a utilisé un tiret, un espace et rien (une chaîne de caractères vide).

Attention, la méthode `.join()` ne s'applique qu'à une liste de chaînes de caractères.

```
>>> maliste = ["A", 5, "G"]
>>> " ".join(maliste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected string, int found
```

On espère qu'après ce petit tour d'horizon vous serez convaincu de la richesse des méthodes associées aux chaînes de caractères. Pour avoir une liste exhaustive de l'ensemble des méthodes associées à une variable particulière, vous pouvez utiliser la fonction `dir()`.

```
>>> animaux = "girafe tigre"
>>> dir(animaux)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '_
_getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '_
_init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mo
d__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__'
, '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'for
mat_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'i
sidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'is
title', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Pour l'instant, vous pouvez ignorer les méthodes qui commencent et qui se terminent par deux tirets bas (*underscores*) `__`.

Vous pouvez également accéder à l'aide et à la documentation d'une méthode particulière avec `help()`, par exemple pour la méthode `.split()` :

```
>>> help(animaux.split)
Help on built-in function split:

split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator.
(END)
```

Attention à ne pas mettre les parenthèses à la suite du nom de la méthode. L'instruction correcte est `help(animaux.split)` et non pas `help(animaux.split())`.

10.8 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

10.8.1 Parcours d'une liste de chaînes de caractères

Soit la liste `['girafe', 'tigre', 'singe', 'souris']`. Avec une boucle, affichez chaque élément ainsi que sa taille (nombre de caractères).

10.8.2 Lecture d'une séquence à partir d'un fichier FASTA

Le fichier `UBI4_SCerevisiae.fasta` contient une séquence d'ADN au format FASTA.

Créez une fonction `lit_fasta()` qui prend comme argument le nom d'un fichier FASTA sous la forme d'une chaîne de caractères, lit la séquence dans le fichier FASTA et la renvoie sous la forme d'une chaîne de caractères.

Utilisez ensuite cette fonction pour récupérer la séquence d'ADN dans la variable `sequence` puis pour afficher les informations suivantes :

- le nom du fichier FASTA,
- la longueur de la séquence (c'est-à-dire le nombre de bases qu'elle contient),
- un message vérifiant que le nombre de base est (ou non) un multiple de 3,
- le nombre de codons (on rappelle qu'un codon est un bloc de 3 bases),
- les 10 premières bases,
- les 10 dernières bases.

La sortie produite par le script devrait ressembler à ça :

```
UBI4_SCerevisiae.fasta
La séquence contient WWW bases
La longueur de la séquence est un multiple de 3 nucléotides
La séquence possède XXX codons
10 premières bases : YYYYYYYYYY
10 dernières bases : ZZZZZZZZZZ
```

où `WWW` et `XXX` sont des entiers et `YYYYYYYYYY` et `ZZZZZZZZZZ` sont des bases.

Conseil : vous trouverez des explications sur le format FASTA et des exemples de code dans l'annexe A *Quelques formats de données rencontrés en biologie*.

10.8.3 Fréquence des bases dans une séquence d'ADN

Soit la séquence d'ADN `ATATACGGATCGGCTGTTGCCTGCGTAGTAGCGT`. On souhaite calculer la fréquence de chaque base A, T, C et G dans cette séquence et afficher le résultat à l'écran.

Créez pour cela une fonction `calc_composition()` à laquelle vous passez en argument votre séquence d'ADN sous forme d'une chaîne de caractères et qui renvoie une liste de quatre *floats* indiquant respectivement la fréquence en bases `A`, `T`, `G` et `C`.

10.8.4 Conversion des acides aminés du code à trois lettres au code à une lettre

Créez une fonction `convert_3_lettres_1_lettre()` qui prend en argument une chaîne de caractères avec des acides aminés en code à trois lettres et renvoie une chaîne de caractères avec les acides aminés en code à 1 lettre.

Utilisez cette fonction pour convertir la séquence protéique `ALA GLY GLU ARG TRP TYR SER GLY ALA TRP`.

Rappel de la nomenclature des acides aminés :

Acide aminé	**Code 3-lettres**	**Code 1-lettre**
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartate	Asp	D
Cystéine	Cys	C
Glutamate	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Méthionine	Met	M
Phénylalanine	Phe	F

Acide aminé	Code 3-lettres	Code 1-lettre
Proline	Pro	P
Sérine	Ser	S
Thréonine	Thr	T
Tryptophane	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

10.8.5 Distance de Hamming

La distance de Hamming mesure la différence entre deux séquences de même taille en comptant le nombre de positions qui, pour chaque séquence, ne correspondent pas au même acide aminé.

Créez la fonction `dist_hamming()` qui prend en argument deux chaînes de caractères et qui renvoie la distance de Hamming (sous la forme d'un entier) entre ces deux chaînes de caractères.

Calculez la distance de Hamming entre les séquences
`AGWPSGGASAGLAIL` et `IGWPSAGASAGLWIL`
puis entre les séquences
`ATTCATACGTTACGATT` et `ATACTTACGTAACCATT`.

10.8.6 Palindrome

Un palindrome est un mot ou une phrase dont l'ordre des lettres reste le même si on le lit de gauche à droite ou de droite à gauche. Par exemple, « ressasser » et « engage le jeu que je le gagne » sont des palindromes.

Créez la fonction `test_palindrome()` qui prend en argument une chaîne de caractères et qui affiche `xxx est un palindrome` si la chaîne de caractères `xxx` passée en argument est un palindrome ou `xxx n'est pas un palindrome` sinon. Pensez à vous débarrasser au préalable des majuscules et des espaces.

Testez ensuite si les expressions suivantes sont des palindromes :

- `radar`
- `never odd or even`
- `karine alla en Iran`
- `un roc si biscornu`

10.8.7 Mot composable

Un mot est composable à partir d'une séquence de lettres si la séquence contient toutes les lettres du mot. Chaque lettre de la séquence ne peut être utilisée qu'une seule fois. Par exemple, « coucou » est composable à partir de « uocuoceokzefhu ».

Écrivez la fonction `test_composable()` qui prend en argument un mot (sous la forme d'une chaîne de caractères) et une séquence de lettres (aussi comme une chaîne de caractères) et qui affiche `Le mot xxx est composable à partir de yyy` si le mot (`xxx`) est composable à partir de la séquence de lettres (`yyy`) ou `Le mot xxx n'est pas composable à partir de yyy` sinon.

Testez cette fonction avec les mots et les séquences suivantes :

Mot	**Séquence**
python	aophrtkny
python	aeiouyhpq
coucou	uocuoceokzezh
fonction	nhwfnitvkloco

10.8.8 Alphabet et pangramme

Les codes ASCII des lettres minuscules de l'alphabet vont de 97 (lettre « a ») à 122 (lettre « z »). La fonction `chr()` prend en argument un code ASCII sous la forme d'un entier et renvoie le caractère correspondant (sous la forme d'une chaîne de caractères). Ainsi `chr(97)` renvoie `'a'`, `chr(98)` renvoie `'b'` et ainsi de suite.

Créez la fonction `get_alphabet()` qui utilise une boucle et la fonction `chr()` et qui renvoie une chaîne de caractères contenant toutes les lettres de l'alphabet.

Un pangramme est une phrase comportant au moins une fois chaque lettre de l'alphabet. Par exemple, « Portez ce vieux whisky au juge blond qui fume » est un pangramme.

Créez la fonction `pangramme()` qui utilise la fonction `get_alphabet()` précédente, qui prend en argument une chaîne de caractères (`xxx`) et qui renvoie `xxx est un pangramme` si cette chaîne de caractères est un pangramme ou `xxx n'est pas un pangramme` sinon. Pensez à vous débarrasser des majuscules le cas échéant.

Testez ensuite si les expressions suivantes sont des pangrammes :

- Portez ce vieux whisky au juge blond qui fume
- Monsieur Jack vous dactylographiez bien mieux que votre ami Wolf
- Buvez de ce whisky que le patron juge fameux

10.8.9 Lecture d'une séquence à partir d'un fichier GenBank (exercice +++)

On cherche à récupérer la séquence d'ADN du chromosome I de la levure *Saccharomyces cerevisiae* contenu dans le fichier au format GenBank `NC_001133.gbk`.

Le format GenBank est présenté en détails dans l'annexe A *Quelques formats de données rencontrés en biologie*. Pour cet exercice, vous devez savoir que la séquence démarre après la ligne commençant par le mot `ORIGIN` et se termine avant la ligne commençant par les caractères `//` :

```
ORIGIN
        1 ccacaccaca cccacacacc cacacaccac accacacacc acaccacacc
cacacacaca
       61 catcctaaca ctaccctaac acagccctaa tctaaccctg gccaacctgt
ctctcaactt
[...]
   230101 tgttagtgtt agtattaggg tgtggtgtgt gggtgtggtg tgggtgtggg
tgtgggtgtg
   230161 ggtgtgggtg tgggtgtggt gtggtgtgtg ggtgtggtgt gggtgtggtg tgtgtggg
//
```

Pour extraire la séquence d'ADN, nous vous proposons d'utiliser un algorithme de « drapeau », c'est-à-dire une variable qui sera à `True` lorsqu'on lira les lignes contenant la séquence et à `False` pour les autres lignes.

Créez une fonction `lit_genbank()` qui prend comme argument le nom d'un fichier GenBank sous la forme d'une chaîne de caractères, lit la séquence dans le fichier GenBank et la renvoie sous la forme d'une chaîne de caractères.

Utilisez ensuite cette fonction pour récupérer la séquence d'ADN dans la variable `sequence` dans le programme principal. Le script affichera :

```
NC_001133.gbk
La séquence contient XXX bases
10 premières bases : YYYYYYYYYY
10 dernières bases : ZZZZZZZZZZ
```

où `XXX` est un entier et `YYYYYYYYYY` et `ZZZZZZZZZZ` sont des bases.

Vous avez toutes les informations pour effectuer cet exercice. Si toutefois vous coincez sur la mise en place du drapeau, voici l'algorithme en pseudo-code pour vous aider :

```
drapeau <- Faux
seq <- chaîne de caractères vide
Lire toutes les lignes du fichier:
    si la ligne contient //:
        drapeau <- Faux
    si drapeau est Vrai:
        on ajoute à seq la ligne (sans espace, chiffre et retour à la ligne)
    si la ligne contient ORIGIN:
        drapeau <- Vrai
```

10.8.10 Affichage des carbones alpha d'une structure de protéine

Téléchargez le fichier `1bta.pdb` qui correspond à la structure tridimensionnelle de la protéine barstar sur le site de la *Protein Data Bank* (PDB).

Créez la fonction `trouve_calpha()` qui prend en argument le nom d'un fichier PDB (sous la forme d'une chaîne de caractères), qui sélectionne uniquement les lignes contenant des carbones alpha, qui stocke ces lignes dans une liste et les renvoie sous la forme d'une liste de chaînes de caractères.

Utilisez la fonction `trouve_calpha()` pour afficher à l'écran les carbones alpha des deux premiers résidus (acides aminés).

Conseil : vous trouverez des explications sur le format PDB et des exemples de code pour lire ce type de fichier en Python dans l'annexe A *Quelques formats de données rencontrés en biologie*.

10.8.11 Calcul des distances entre les carbones alpha consécutifs d'une structure de protéine (exercice +++)

En utilisant la fonction `trouve_calpha()` précédente, calculez la distance interatomique entre les carbones alpha des deux premiers résidus (avec deux chiffres après la virgule).

Rappel : la distance euclidienne *d* entre deux points A et B de coordonnées cartésiennes respectives (x_A, y_A, z_A) et (x_B, y_B, z_B) se calcule comme suit :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

Créez ensuite la fonction `calcule_distance()` qui prend en argument la liste renvoyée par la fonction `trouve_calpha()`, qui calcule les distances interatomiques entre carbones alpha consécutifs et affiche ces distances sous la forme :

`numero_calpha_1 numero_calpha_2 distance`

Les numéros des carbones alpha seront affichés sur 2 caractères. La distance sera affichée avec deux chiffres après la virgule. Voici un exemple avec les premiers carbones alpha :

```
 1  2 3.80
 2  3 3.80
 3  4 3.83
 4  5 3.82
```

Modifiez maintenant la fonction `calcule_distance()` pour qu'elle affiche à la fin la moyenne des distances.

La distance inter-carbone alpha dans les protéines est très stable et de l'ordre de 3,8 angströms. Observez avec attention les valeurs que vous avez calculées pour la protéine barstar. Repérez une valeur surprenante. Essayez de l'expliquer.

Conseil : vous trouverez des explications sur le format PDB et des exemples de code pour lire ce type de fichier en Python dans l'annexe A *Quelques formats de données rencontrés en biologie*.

10.8.12 Compteur de gènes dans un fichier GenBank

Dans cet exercice, on souhaite compter le nombre de gènes du fichier GenBank NC_001133.gbk (chromosome I de la levure Saccharomyces cerevisiae) et afficher la longueur de chaque gène. Pour cela, il faudra récupérer les lignes décrivant la position des gènes. Voici par exemple les cinq premières lignes concernées dans le fichier NC_001133.gbk:

```
     gene            complement(<1807..>2169)
     gene            <2480..>2707
     gene            complement(<7235..>9016)
     gene            complement(<11565..>11951)
     gene            <12046..>12426
[...]
```

Lorsque la ligne contient le mot `complement` le gène est situé sur le brin complémentaire, sinon il est situé sur le brin direct. Votre code devra récupérer le premier et le second nombre indiquant respectivement la position du début et de fin du gène. Attention à bien les convertir en entier afin de pouvoir calculer la longueur du gène. Notez que les caractères `>` et `<` doivent être ignorés, et les `..` servent à séparer la position de début et de fin.

On souhaite obtenir une sortie de la forme :

```
gène   1 complémentaire ->   362 bases
gène   2 direct         ->   227 bases
gène   3 complémentaire ->  1781 bases
[...]
gène  99 direct         ->   611 bases
gène 100 direct         ->   485 bases
gène 101 direct         ->  1403 bases
```

Conseil : vous trouverez des explications sur le format GenBank dans l'annexe A *Quelques formats de données rencontrés en biologie*.