Chapitre # (ALGO) 2 Listes, Tuples & Chaînes de caractères

Les tentatives de création de machines pensantes nous seront d'une grande aide pour découvrir comment nous pensons nous-mêmes.

— **Alan Turing**

Résumé & Plan

L'objectif de ce TP est d'étudier diverses structures de données séquentielles, *i.e.* où les éléments sont repérés par des entiers (listes, tuples et chaînes).

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices.
- Traitez les exercices présents dans la liste donnée au début de séance.
- Exercices ◐ / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

Les listes, tuples et les chaînes de caractères sont des structures de données basiques en programmation. Les manipulations de ces types d'objets sont très proches : on se repère dans ces structures de la même façon, au moyen d'entiers. Et c'est pour cette raison que nous les traitons de manière simultanée. Les listes sont à utiliser dans beaucoup de contextes généraux, les chaînes sont à privilégier si l'on a besoin de faire du traitement de texte (suppression d'accents, mise en minuscule/majuscule *etc.*), les tuples sont des listes non modifiables, on les utilise assez peu en tant que tel.

1. LISTES ET TUPLES

1.1. Généralités

- Une *liste* est une suite finie d'éléments pouvant être de types différents. Ces éléments sont modifiables (ou mutables), contrairement aux tuples que nous verrons juste après. On peut de plus ajouter ou enlever des éléments à une liste, ce qui permet de représenter des structures de données évoluant au cours du temps (création d'une liste initiale, puis modifications successives). On se repère dans les éléments d'une liste par des entiers relatifs (positif lorsque l'on compte directement, négatif en comptage à rebours).
- Les *tuples* (« n-uplets » en français) correspondent aux listes à la différence qu'ils sont non modifiables : nous retiendrons uniquement cette différence avec les listes.

(Définir une liste ou un tuple en Python) Dans la pratique, on définit un tuple à l'aide de parenthèses *a contrario* des listes qui utilisent des crochets. Pour une liste vide, on tape

```
>>> L = []    # ou encore :
>>> L = list()
```

On peut aussi directement taper la valeur des éléments :

```
>>> L = [1, 7, 4.0]
```

Il est possible que chaque élément d'une liste soit de type différent (précédemment des entiers et flottants), et même des listes.

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]] # une liste
>>> T = ("ATGC", 1, 3.0, [1, 2, 3]) # un tuple
```

(Longueur) Une liste possède une *longueur*, qui correspond au nombre d'éléments d'une liste, elle est donnée par la fonction `len` en Python. Par exemple :

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]]
```

```
>>> len(L)
4
```

L'objet `"ATGC"` est ce qu'on appelle une « chaîne de caractères ». Peu importe ce que c'est pour le moment, nous l'étudierons dans la prochaine section.

(Repérage & Changement de valeur) On accède à un élément d'une liste en utilisant son *indice* dans ladite liste, numéroté **à partir de zéro**. Par exemple :

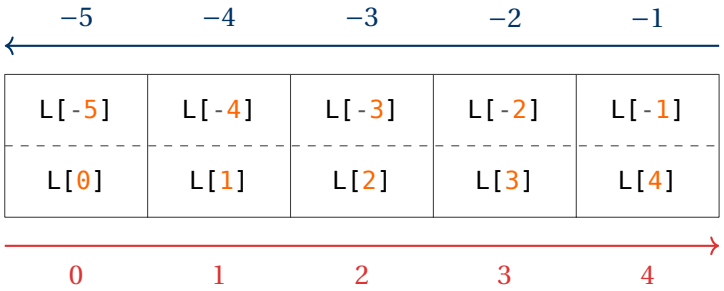
```
>>> L[0]
'ATGC'
>>> L[1]
1
>>> L[2]
3.0
>>> L[3]
[1, 2, 3]
>>> len(L[3]) # ici, le dernier élément est une liste donc \
↪ possède lui aussi une longueur
3
>>> len(L[len(L)-1]) # le dernier élément a pour indice \
↪ len(L) -1 : ne pas oublier le -1 car on compte à partir de \
↪ 0.
3
>>> # on peut aussi compter à rebours :
>>> L[-1]
[1, 2, 3]
>>> L[-2]
3.0
>>> L[-3]
1
```

Lorsque l'on sort des indices admissibles d'une liste, on obtient une erreur.

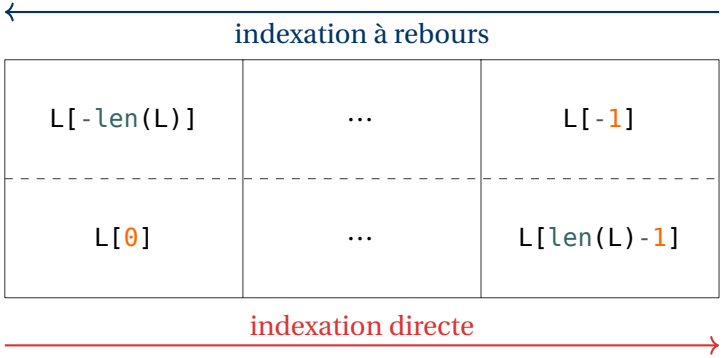
```
>>> L[5] # oups !
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list index out of range
```

Enfin, il est possible de modifier les éléments d'une liste, par exemple en faisant :

```
>>> L[1] = 2
>>> L
['ATGC', 2, 3.0, [1, 2, 3]]
>>> L[1] += 1 # on ajoute 1 au deuxième élément
>>> L
['ATGC', 3, 3.0, [1, 2, 3]]
```



INDEXATION DES ÉLÉMENTS D'UNE LISTE DE LONGUEUR 5



INDEXATION DES ÉLÉMENTS D'UNE LISTE DQUELCONQUE

(Différence entre les deux : la mutabilité des éléments)

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]]
>>> L[2]
3.0
>>> L[2] = 15 # ça marche !
>>> L
['ATGC', 1, 15, [1, 2, 3]]
>>>
>>> t = ("ATGC", 1, 3.0, [1, 2, 3])
>>> t[2]
3.0
>>> t[2] = 15 # oups !
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Exercice 1 | **Égalité de valeurs extrêmes** Solution Écrire une fonction d'en-tête `est_indent(L)` qui renvoie **`True`** si le premier et le dernier élément d'une liste `L` sont identiques, et **`False`** sinon. Par exemple, `est_ident([1, 2, 1])` retourne **`True`**, alors que `est_ident([1, 2, 3])` retourne **`False`**.

(Suppression, Ajout/échanges d'éléments & Concaténation) On peut facilement supprimer un élément d'une liste en connaissant son indice (à l'aide de **`del`**), ou alors supprimer le premier (en partant de la gauche) élément possédant une certaine valeur à l'aide de `remove`. Par exemple,

```
>>> L = ["ATGC", 1, 3.0, [1, 2, 3]]
>>> del L[1] # suppression du deuxième élément
>>> L # la liste est modifiée directement
['ATGC', 3.0, [1, 2, 3]]
>>> L.append("ATGC") # analogue à L += ["ATGC"]
>>> L
['ATGC', 3.0, [1, 2, 3], 'ATGC']
>>> L.remove("ATGC")
>>> L # supprime le premier "ATGC" en partant de la gauche
[3.0, [1, 2, 3], 'ATGC']
>>> L[0], L[1] = L[1], L[0] # échange des deux premiers \
↪ éléments
>>> L
[[1, 2, 3], 3.0, 'ATGC']
```

Exemple 1 **(Fonction à effet de bord)** On considère la fonction ci-après. Que fait-elle ?

```
def mystere(L):
    L.append(L[-1]**2)
```

Notez bien comment observer son effet :

```
>>> L = [1, 2, 3]
>>> mystere(L)
>>> L
[1, 2, 3, 9]
```

Exercice 2 | **Échange en place de deux éléments** Solution Écrire une fonction d'en-tête `echangeEnPlace(L, i, j)` qui prend en arguments une liste `L` et deux indices `i` et `j` et qui échange les éléments d'indices `i` et `j` dans la liste. Cette fonction devra modifier la liste sans rien renvoyer.

Remarque 1 La fonction de l'exercice précédente est une fonction à effet de bord (comme défini dans le Chapitre (ALGO) 1) : la fonction modifie la liste `L` passée en argument.

(Parcourir une liste) On dispose de deux options pour cela : soit au moyen d'un entier correspondant à l'indice de l'élément, soit directement en parcourant les éléments de la liste.

```
>>> for i in range(len(L)):
...     print(L[i])
...     # à utiliser quand \
    ↪ vous avez besoin de \
    ↪ travailler avec \
    ↪ l'indice i ensuite
...
1
2
3
9
```

```
>>> for x in L:
...     print(x)
...     # à utiliser quand \
    ↪ seul la valeur de \
    ↪ l'élément vous importe
...
1
2
3
9
```

Attention

Bien mettre **`for i in range(len(L))`** car on rappelle que par défaut un range s'arrête strictement avant la borne de droite. Ainsi, `i` parcourra `0, ..., len(L)-1` ce qui est exactement la liste des indices de ses éléments.

(Concaténation) On peut aussi fusionner deux listes au moyen de l'opérateur `+`, ou encore de la méthode `extend` ou encore de manière « artisanale » à l'aide d'une boucle **`for`**.

```
>>> L
[1, 2, 3, 9]
>>> M = [1, 2, 3]
>>> N = L + M
>>> N
[1, 2, 3, 9, 1, 2, 3]
```

```
[1, 2, 3, 9]
>>> M = [1, 2, 3]
>>> L.extend(M)
>>> L # modifie directement \
↪ la liste initiale
[1, 2, 3, 9, 1, 2, 3]
```

```
>>> L
```

On peut aussi utiliser, comme annoncé, plusieurs `append` au moyen d'une boucle **`for`**.

```
>>> L
[1, 2, 3, 9, 1, 2, 3]
>>> M = [1, 2, 3]
>>> for x in M:
...     L.append(x)
...
>>> L
[1, 2, 3, 9, 1, 2, 3, 1, 2, 3]
```

MODES DE DÉFINITION D'UNE LISTE. Comme pour les ensembles, il est possible de définir une liste de plusieurs manières.

(Construire une liste par énumération) Une liste peut être définie par énumération (c'est le mode utilisé précédemment).

```
>>> L = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut également partir d'une liste vide, et la remplir à l'aide d'une boucle `for`.

```
>>> L_bis = []
>>> for i in range(10):
...     L_bis.append(i**2)
...
>>> L_bis
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(Construire une liste par compréhension) On peut également la définir en compréhension :

-

```
>>> M = [i**2 for i in range(10)]
>>> M
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

On peut aussi filtrer les éléments selon une condition, indiquée dans la compréhension. Par exemple, si on ne souhaite que les carrés d'entiers pairs, on tapera

```
>>> N = [i**2 for i in range(10) if i%2 == 0]
>>> N
[0, 4, 16, 36, 64]
```

Et pour finir, comme pour les boucles `for`, si l'indice du `range` n'est pas important, on peut utiliser le symbole underscore. Par exemple, pour créer une liste de zéros de taille 10, on tape

```
>>> zeros = [0 for _ in range(10)]
>>> zeros
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

CAS PARTICULIER DES LISTES D'ENTIERS. Pour créer rapidement des listes d'entiers, on se sert de `range`. On rappelle que cet objet a été défini dans le Chapitre (ALGO) 1.

(Produire des listes d'entiers avec `range`) L'instruction `list(range(n))` produit une liste d'entiers consécutifs entre 0 et $n-1$:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

De manière générale, l'instruction `list(range(a, b))` produit une liste d'entiers consécutifs entre `a` et `b-1` si `b-1` est supérieur ou égal à `a` et une liste vide dans le cas contraire. Dans le premier cas, on peut également spécifier un pas d'avancement : `list(range(a, b, h))` retourne la liste des entiers de la forme `a + h*k` tant que `a + h*k` est inférieur strictement à `b`.

```
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
```

Attention

Il faut bien faire attention au fait que `range(...)` **n'est pas** une liste mais un *itérateur*, *i.e.* un objet de référence pour décrire une boucle `for`. Il faut donc d'abord le convertir en liste pour afficher ses valeurs. En effet,

```
>>> range(1, 12)
range(1, 12)
>>> list(range(1, 12))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Exercice 3 | Années à coupe du monde Solution Construire de plusieurs manières la liste des années à coupe du monde de foot entre 1998 et 2100. *À l'aide d'un `range`, puis une boucle `while` par exemple*

Exercice 4 | Tables de multiplication Solution Dans tout l'exercice, on répondra aux questions en une seule commande avec les instructions `range()` et `list()`.

1. Affichez la table de multiplication par 9, *i.e.* la liste des entiers $0, 9, 18, \ldots, 90$.
2. Combien y a-t-il de nombres pairs dans l'intervalle `[2, 10000]` ?

DÉCOUPAGES (OU *SLICING*) DE LISTES. On peut aussi, à partir d'une liste complète, en extraire des tranches. Voyons comment. Soit `L` une liste de longueur `n`, et `p, q` deux entiers tels que $0 \leqslant p, q \leqslant n$.

- `L[p:q]` contient tous les éléments de `L` dont l'indice est dans $[\![p\,,\,q-1]\!]$.
- `L[:q]` est la liste constituée des `q` premiers éléments de `L`, c'est-à-dire ceux entre les indices `0` et `q-1`.[1]
- `L[p:]` est la liste `L` privée de ses `p` premiers éléments.

Découpage

```
>>> L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L[0:3]
[0, 1, 2]
>>> L[3:6]
[3, 4, 5]
>>> L[3:4]
[3]
>>> L[3:3]
[]
>>> L[3:2]
[]
>>> L[3:]
[3, 4, 5, 6, 7, 8, 9]
>>> L[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> L[:] # on extrait tout, pas très utile à part pour copier \
↪ des listes (voir la section sur les copies)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Remarque 2 On remarquera que `L[2:3]` est une liste, alors que `L[2]` est, dans l'exemple précédent, un entier.

Exercice 5 | Jours de la semaine Solution

1. Constituez une liste `Semaine` semaine contenant les noms des 7 jours de la semaine.
2. À partir de cette liste, comment récupérer une liste contenant seulement les 5 premiers jours de la semaine? ceux du week-end d'autre part?
3. Proposez deux instructions permettant d'accéder au dernier jour de la semaine.

1. Tout comme pour les `range`, la borne de droite est exclue.

LISTES DE LITES. On a parfois besoin de créer/manipuler des listes dont les éléments sont eux-mêmes des listes, il faut donc s'habituer à manipuler les syntaxes faisant intervenir plusieurs crochets `[]` à la suite. Plutôt que de longs discours, voyons cela avec des exercices.

Exemple 2 On considère `L = [[3,1], [7], [1,9,8,0]]`.

1. Que vaut `L[1]`?
2. Que vaut `L[0][1]`?
3. Que vaut `L[2][3:4][0]`?
4. Que vaut `len(L)`?
5. Que vaut `L` après avoir exécuté `L.append(9.75)`?
6. Que vaut `L` après avoir exécuté `del L[0][1]`?

Exercice 6 | Liste de listes des saisons Solution On suppose définie dans Python la liste ci-après :

```
saisons = ['Janvier', 'Fevrier', 'Mars',  'Avril', 'Mai', \
↪ 'Juin', 'Juillet', 'Aout', 'Septembre', 'Octobre', 'Novembre', \
↪ 'Decembre']
```

1. En déduire 4 instructions (de slicing) permettant de créer 4 listes correspondant à chaque saison. Par exemple, la variable `hiver` devra contenir `['Janvier', 'Fevrier', 'Mars']`, et de-même pour les autres saisons.
2. Créez ensuite une liste `saisons_blocs` égale à `[hiver, printemps, ete, automne]`. Prévoyez ce que renvoient les instructions suivantes, puis vérifiz-le dans l'interpréteur :

1. `saisons[2]`
2. `saisons[1][0]`
3. `saisons[1:2]`
4. `saisons[:][1]`.

Exercice 7 | **Séparation indices pairs / impairs** Solution Écrire une fonction d'entête `split_list(L)` qui prend en entrée une liste d'entiers et qui renvoie une liste de listes, dont la première est la liste constituée des entiers pairs et la seconde de la liste des entiers impairs de la liste d'origine. *Par exemple,* `split_list([1, 3, 4, 2, 5, 2])` *doit renvoyer* `[[4, 2, 2], [1, 3, 5]]`.

Dans toute la suite, nous ne travaillerons plus qu'avec des listes.

1.2. Copies de listes

Les copies de listes sont sources de pièges, et cela nécessite bien une section dédiée. Regardons un premier exemple.

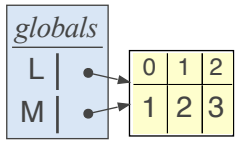
```
>>> L = [1,2,3]
>>> M = L
>>> M
[1, 2, 3]
>>> L[1] = -15
>>> L
[1, -15, 3]
>>> M
[1, -15, 3]
```

On voit que `M` a également été modifiée. Ceci est dû au fait suivant : les éléments de `M` sont liés à ceux de `M`, l'instruction `M = L` ne réalise donc pas, par défaut, une copie en dure des éléments de `L`.

Ceci est dû au fait, qu'en Python, une liste est un *tableau d'adresses*, donc en faisant `M = L`, on crée une liste d'adresses qui sont les mêmes que celles de `L`. Ainsi, une modification de `L` entrainera une modification de `M`. Voyons ce qui se passe côté machine :

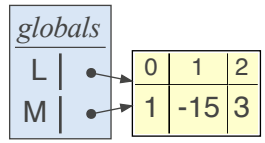
On crée ici une liste `L`, en faisant `M = L` on voit que les deux objets pointent vers la même liste (tableau de droite).

```
L = [1,2,3]
M = L
```



Ainsi, lorsque l'on modifie `L`, la liste `M` sera elle aussi modifiée.

```
L[1] = -15
```



Remarque 3 **(Intérêt de l'adressage)** La taille (comptée en bits) d'une adresse est beaucoup plus petite que celle des données. Il est alors beaucoup plus rentable, par exemple dans les problèmes de tri, de travailler sur les adresses que sur les objets.

COMMENT EFFECTUER UNE « VRAIE » COPIE ? Pour recopier une liste `X` **contenant des entiers** dans une autre liste `Y`, on écrira plutôt

```
X = Y[:] # un découpage total crée une copie
#ou Y=list(X)
```

Et là ça fonctionne :

```
>>> X = [1,2,3]
>>> Y = X[:]
>>> Y
[1, 2, 3]
>>> X[1] = -15
>>> X
[1, -15, 3]
>>> Y
[1, 2, 3]
```

Remarque 4 **(Pour des entiers ou flottants, tout va bien)** Le phénomène précédent ne se produit en revanche **pas** pour des entiers. En effet :

```
>>> a = 1
>>> b = a
>>> a = 2
>>> b
1
```

En revanche, la méthode précédente ne fonctionne toujours pas pour les listes de listes (donc les listes qui contiennent au moins une liste dans une des coordonnées). Voici une méthode fonctionnant toujours, à l'aide du module `copy`.

```
>>> import copy
>>> X = [[1, 2], [3, 4]]
>>> Y = copy.deepcopy(X)
>>> X[1][1] = 99
>>> X
[[1, 2], [3, 99]]
>>> Y
[[1, 2], [3, 4]]
```

Exercice 8 | **Échange non en place de deux éléments** Solution Écrire une fonction d'en-tête `echangeNonEnPlace(L, i, j)` qui prend en arguments une liste `L` et deux indices `i` et `j` et qui renvoie une copie de la liste `L` dans laquelle les éléments d'indices `i` et `j` sont échangés. Cette fonction ne devra pas modifier la liste `L` initiale.

1.3. Résumé non exhaustif des opérations sur les listes

Étant donnée une liste `L`, on dispose des opérations suivantes :

MANIPULATIONS DE LISTES

Commande	Effet
`len(L)`	longueur
`L[0]`	premier élément
`L[-1]`	dernier élément
`L[i:j]`	liste extraite des éléments d'indices entre i (inclus) et j (exclus)
`L[i:]`	liste extraite à partir de l'indice i (inclus)
`L.append(v)`	ajoute l'élément `v` à la fin de la liste
`L.remove(v)`	supprimer le premier élément `v` apparaissant dans la liste,
	retourne une erreur s'il n'est pas présent
`L.extend(s)`	ajoute la liste `s` à la fin de la liste
`L.insert(i, v)`	insert l'objet `v` à l'indice `i`
`del L[i]`	supprime l'élément d'indice `i`

MANIPULATIONS DE LISTES

`del L[i:j]`	supprime les éléments entre les indices `i` et `j` si `j > i`
`L.pop()`	supprime le dernier élément et retourne l'élément supprimé

On peut aussi citer d'autres fonctions et méthodes, qui serviront parfois dans certains exercices (si on vous l'y autorise) :

- `L.reverse()`, retourne la liste et modifie `L`,
- `L.sort()`, trie la liste dans l'ordre croissant et modifie `L`,
- `M = sorted(L)`, trie la liste par ordre croissant en créeant une copie de `L`.

Exercice 9 | **Suppression des doublons sur un exemple** Solution Soit la liste de nombres `L = [5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]`. À partir de commandes du tableau précédent (à indiquer directement dans la console), agir sur `L` pour qu'elle soit sans doublon : c'est-à-dire pour que `L = [5, 1, 2, 3, 4]`.

1.4. Algorithmes classiques sur les listes

On s'interdira dans cette partie d'utiliser une fonction déjà existante de Python.

Attention

Cette sous-section est presque la plus importante de l'année en Informatique. Toutes ces briques de base algorithmiques seront très largement réutilisées tout au long de votre CPGE.

Exercice 10 | **Somme des éléments d'une liste** Solution Écrire une fonction itérative d'en-tête `somme(L)` qui renvoie la somme des termes de la liste. *On s'interdira bien entendu d'utiliser la fonction `sum` existante sur les listes.*

Exercice 11 | **Appartenance d'un élément dans une liste – Méthode par balayage** Solution

1. Écrire une fonction `appartient(x, L)` qui détermine si la liste L contient l'élément `x`. Notez que cette fonction existe déjà : `x in L` renvoie le même résultat.
2. Écrire une fonction `indice(x, L)` qui renvoie le premier indice où l'élément `x` apparait dans la liste L, et `None` si L ne contient pas `x`. Notez que cette fonction existe déjà : `L.index(x)` renvoie le même résultat.
3. **[Suppression des doublons]** Écrire une fonction d'en-tête `suppr_doublon(L)` qui retourne une autre liste contenant les éléments de `L` mais apparaissant une unique fois. *Par exemple, `suppr_doublon([1,2,2,3,5,4,5])` retournera `[1, 2, 3, 5, 4]`.*

Exercice 12 | Autour du maximum d'une liste Solution

1. On veut écrire une fonction d'en-tête `maximum(L)` renvoyant la valeur du plus grand élément de la liste `L` (celle-ci étant supposée non vide, et ne contenir que des entiers). On donne l'algorithme sur lequel repose la fonction :
   - affecter le premier élément de la liste `L` à une variable `maxi`,
   - pour chacun des éléments `x` de `L`,
     - si `x` est plus grand que `maxi`, affecter la valeur de `x` à `maxi`.

   Écrivez cette fonction en utilisant un parcours de la liste à l'aide de l'instruction `for x in L`. Testez la fonction sur des listes présentant des situations diverses (maximum en première position, en dernière position, ailleurs que sur les bords, liste à un seul élément, élément maximal apparaissant plusieurs fois dans la liste).
2. On peut noter que l'emploi de `for x in L` entraîne un test inutile (celui du premier élément de `L`). Cependant la syntaxe associée est très simple et bien adaptée à la situation. Modifier la fonction précédente en utilisant un slicing sur `L` pour remédier à ce problème.
3. Écrivez sur le même principe une fonction d'en-tête `minimum(L)` renvoyant cette fois la valeur du plus petit élément de la liste `L`, et testez-la sur plusieurs listes bien choisies.
4. Écrivez une fonction d'en-tête `maximum_premind(L)` renvoyant le plus petit indice `k` tel que `L[k]` soit le plus grand élément de la liste `L`, et utilisant une boucle `for`. Par exemple, pour la liste `L = [2, 4, 1, 5, 3]`, la fonction renverra 3 puisque le plus grand élément qui vaut 5 apparaît à l'indice 3. Quel type de parcours de liste est-il nécessaire d'utiliser ici ? Comment renvoyer le dernier indice ?
5. Écrire une fonction `maximum_occur(L)` qui recherche le maximum de la liste `L`, et renvoie la liste des indices où le maximum est atteint. Même question avec le minimum. Comment améliorer la fonction `maximum_occur` pour qu'elle ne nécessite qu'une seule boucle `for` ? On écrira alors une fonction `maximum_occur_bis`.

Exercice 13 | Renverser une liste Solution Écrire une fonction d'en-tête `renverse(L)`, et qui étant donnée une liste `L`, retourne la même liste mais avec des éléments écrits dans l'autre sens. *Notez que cette opération existe déjà sous forme de méthode, qui modifie `L` directement :*

```
>>> L = [1, 2, 3]
>>> L.reverse()
>>> L
[3, 2, 1]
```

Exercice 14 | Décaler une liste Solution Écrire une fonction d'en-tête `decale(L)`, et qui étant donnée une liste `L`, retourne la même liste mais avec les éléments décalés une fois vers la droite. Par exemple, `decale([1, 2, 3])` retournera `[3, 1, 2]`.

Exercice 15 | Appliquer une fonction sur une liste Solution L'exercice suivant peut être fait très facilement avec la bibliothèque `nummpy`, mais nous allons nous en passer.

1. Créer une fonction d'en-tête `Applique_Fonction(L, f)` qui étant données une liste `L` et une fonction `f`, retourne la liste où la fonction `f` est appliquée à chaque élément.
2. **[Application]** Créer une fonction `Calc_Somme_Carre` prenant en argument un entier `n` et retournant $\sum_{i=1}^{n} i^2$. *On pourra se servir de la fonction `sum` de Python*

POUR ALLER PLUS LOIN. On propose ici quelques exercices supplémentaires une fois tout le reste du TP terminé.

Exercice 16 | Max/Min en un coup Solution Proposer une fonction d'en-tête `min_max(L)` qui retourne le minimum et le maximum de `L`, à l'aide d'une seule boucle `for`.

Exercice 17 | Insérer dans une liste Solution Écrire une fonction `inserer` qui prend en entrée une liste, un élément `x` et un indice `i`, et qui insère l'élément `x` dans la liste à la position `i`. *Notez que cette fonction existe déjà, on peut utiliser la méthode `insert` sur les listes. Attention : on demande une fonction qui **modifie la liste `L` donnée en entrée**.*

Exercice 18 | Deux maximums d'une liste Solution Écrire une fonction d'en-tête `deux_maximum(L)`, qui à une liste `L` retourne :

- l'unique élément de `L` si elle est de longueur,
- sinon, retourne les deux plus grands éléments (au sens large).

Indication : On cherchera à imiter la fonction `maximum` codée dans un précédent exercice

2. CHAÎNES DE CARACTÈRES

2.1. Généralités

Les chaînes de caractères (type `string` sont des listes de caractères (lettres de l'alphabet, chiffres, symboles). On les note entre guillemets [2] ou apostrophes :

2. Mieux vaut privilégier ceux-ci afin d'éviter les complications dues aux accents

```
>>> ch1 = "bcpst1" # <- utiliser plutôt cette version, moins \
↪ sensible aux accents
>>> ch2 = 'bcpst1'
```

Nous présenterons très peu de généralités sur les chaînes, car elles se manipulent essentiellement comme des listes.

- On accède à chacun des caractères comme pour une liste :
```
>>> ch1[0]
'b'
```
- et on dispose des fonctions de concaténation (`ch2 + ch2`).
```
>>> ch3 = ch1+ch2 # création d'une nouvelle chaîne par \
↪ concaténation
>>> ch3
'bcpst1bcpst1'
```
- Une chaîne possède une longueur, et attention : les espaces sont comptés comme un caractère.
```
>>> s = "les BCPST1 aiment l'informatique"
>>> len(s)
32
```

Attention **On ne peut pas modifier une chaîne existante !**

Il est fondamentale de retenir qu'on ne peut changer une chaîne existante. Comme pour les `tuple`, c'est un type non mutable. Voyez l'exemple ci-après.

```
>>> ch1 = "bcpst1"
>>> ch1[5] = "2"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> ch1_modif = ch1[0:5] + "2" # Il faut créer une nouvelle \
↪ chaîne
>>> ch1 = ch1[0:5] + "2" # On peut aussi "écraser" l'ancienne \
↪ chaîne
>>> ch1
'bcpst2'
```

Ainsi, pour manipuler des chaînes, on peut transiter par des listes que l'on retransforme en chaîne. [3]

3. Mais peut-être que le type `string` n'était alors pas le bon type à utiliser dans ce contexte ...

Passage de chaînes aux listes

```
>>> ch1 = "bcpst1"
>>> Lch1 = list(ch1)
>>> Lch1 # là on peut modifier L
['b', 'c', 'p', 's', 't', '1']
>>> Lch1[5] = "2"
>>> "".join(Lch1) # permet de revenir à la chaîne initiale
'bcpst2'
```

Notez que l'on peut, comme pour les listes :

- effectuer des découpages de chaînes (avec la même syntaxe),
- parcourir une chaîne avec une boucle **`for`**.
- En revanche, on ne **peut pas** créer de chaînes par compréhension.

Remarque 5 **(Opérations spécifiques au traitement de textes)** On présente ici quelques opérations très spécifiques aux chaînes, il ne faut pas les retenir mais seulement savoir qu'elles existent.

Séparer les éléments d'une phrase en fonction des espaces

```
>>> s = "les BCPST1 aiment l'informatique"
>>> L = s.split()
>>> L
['les', 'BCPST1', 'aiment', "l'informatique"]
```

Mettre un texte en majuscule ou minuscule

```
>>> s = "BcPsT MontAIGNE, les pro de L'Info"
>>> s.lower()
"bcpst montaigne, les pro de l'info"
>>> s.upper()
"BCPST MONTAIGNE, LES PRO DE L'INFO"
```

Il en existe encore beaucoup d'autres (suppression des espaces, des accents par exemple) mais inutiles de toutes les connaitre. S'il y en besoin, l'exercice vous les donnera.

2.2. Résumé non exhaustif des opérations sur les chaînes

Etant donnée une chaîne `s`, on dispose des opérations suivantes (très proches des précédentes), avec en plus des méthodes/fonctions sur la mise en forme de carac-

tères propres aux chaînes (passage en majuscule, en minuscule, séparation des caractères pour les mettre dans une liste, *etc.*).

Manipulations de chaînes	
Commande	**Effet**
`len(s)`	longueur
`s[0]`	premier élément
`s[-1]`	dernier élément
`s[i:j]`	chaîne extraite des éléments d'indices entre i (inclus) et j (exclus)
`s[i:]`	chaîne extraite à partir de l'indice i (inclus)
`s.join`	concatène à la chaîne de départ la nouvelle
`L = s.split()`	affecte à `L` la chaîne `s` dont les mots ont été séparés (et toute la chaîne si elle n'a pas de trou)

2.3. Algorithmes classiques sur les chaînes

Comme pour les listes, il y a des algorithmes très classiques sur les chaînes à bien connaitre. Les chaînes forment un moyen très commode pour traiter des problèmes de génétique, une séquence d'ADN pouvant être codée très simplement à l'aide d'une chaîne ne comportant que les lettres `A, T, G, C`.

Exercice 19 | Rechercher un caractère dans une chaîne – Méthode par balayage Solution

Écrire une fonction `cherche_carac(x, ch)` qui détermine si la chaîne `ch` contient l'élément `x`.

Exercice 20 | Recherche de mots dans une chaîne Solution

1. Écrire une fonction `recherche_Mot(m, t)` qui recherche si le mot `m` est présent dans le texte `t`, et qui renvoie la position de la première occurence du mot s'il est présent, et `False` sinon (cela inclut le cas où `m` est de longueur supérieure à `t`).
2. **[Application]** On dit qu'une chaîne de caractère code une séquence `t` d'ADN si elle est composée uniquement des lettres `'A','T','G','C'` et qu'un *codon stop* est un triplet du type `TAA, TAG` ou `TGC`.
   - **2.1)** Écrire une fonction d'en-tête `est_adn(t)` et qui renvoie `True` si `t` est une séquence d'ADN.
   - **2.2)** Écrire une fonction d'en-tête `codonstop(t)` qui renvoie `True` si la séquence `t` contient un codon stop et `False` sinon.

POUR ALLER PLUS LOIN. Vous trouverez ici quelques exercices supplémentaires à destination des plus rapides, si vous n'êtes pas dans ce cas de figure.

Exercice 21 | Nombre de caractères communs entre deux chaînes Solution

Écrire une fonction d'en-tête `compte_commun(c1, c2)` prenant en argument deux chaînes `c1, c2` et qui retourne le nombre de caractère en commun.

Exercice 22 | Décalages dans une chaîne, prémisses du codage de VIGENÈRE Solution

On supposera effectuée en début d'exercice la déclaration suivante :

```
alphabet = [chr(97+k) for k in range(26)]
```

1. Écrire une fonction `codage(c)` qui étant donnée une chaîne `c` ne comportant que des lettres non accentuées, retourne leur indice dans l'alphabet entre `0` et `25`.
2. Écrire une fonction `decodage(L)` qui étant donnée une liste `L` ne comportant que des entiers entre 0 et 25, retourne la chaîne associée.
3. Écrire une fonction `decalage(c, n)` qui étant donnée une chaîne `c` ne comportant que des lettres non accentuées, retourne une autre chaîne dont les lettres ont été décalées de `n` modulo 26.

3. SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé

```
def est_ident(L):
    if L[0] == L[-1]:
        return True
    else:
        return False
```

Ou de manière plus condensée :

```
def est_ident(L):
    return L[0] == L[-1]
```

Solution (exercice 2) Énoncé

```
def echangeEnPlace(L, i, j):
    L[i], L[j] = L[j], L[i]
```

```
>>> L = [1, 2, 3, 4]
>>> echangeEnPlace(L, 0, 2)
>>> L # on affiche L pour voir l'effet
[3, 2, 1, 4]
```

Solution (exercice 3) Énoncé Je rappelle qu'une coupe du monde a lieu tous les 4 ans.

1ère solution : à l'aide d'un `range`.

```
>>> L = list(range(1998,2101,4))
>>> L
[1998, 2002, 2006, 2010, 2014, 2018, 2022, 2026, 2030, 2034, \
↪ 2038, 2042, 2046, 2050, 2054, 2058, 2062, 2066, 2070, \
↪ 2074, 2078, 2082, 2086, 2090, 2094, 2098]
```

2ème solution : par complétion à l'aide d'un `while`

```
>>> M = []
>>> k = 0
>>> while 1998+4*k <= 2100:
...     M.append(1998+4*k)
...     k += 1
...
>>> M
[1998, 2002, 2006, 2010, 2014, 2018, 2022, 2026, 2030, 2034, \
↪ 2038, 2042, 2046, 2050, 2054, 2058, 2062, 2066, 2070, \
↪ 2074, 2078, 2082, 2086, 2090, 2094, 2098]
```

Solution (exercice 4) Énoncé Il s'agit de faire afficher les entiers $9k$ avec $k \in [\![1, 9]\!]$.

```
>>> list(range(0, 91, 9))
[0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

Cela peut également se faire par compréhension. Pour la seconde question, la commande suivante répond au problème.

```
>>> L = list(range(2,10001,2))
>>> len(L)
5000
```

Solution (exercice 5) Énoncé

```
>>> Semaine = ["lundi", "mardi", "mercredi", "jeudi", \
↪ "vendredi", "samedi", "dimanche"]
>>> Semaine[0:5]
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> Semaine[5:]
['samedi', 'dimanche']
>>> Semaine[-1] # ou bien
'dimanche'
>>> Semaine[6]
'dimanche'
```

Solution (exercice 6) Énoncé

```
>>> mois = ['Janvier', 'Fevrier', 'Mars',  'Avril', 'Mai', \
↪ 'Juin', 'Juillet', 'Aout', 'Septembre', 'Octobre', \
↪ 'Novembre', 'Decembre']
>>> hiver = mois[:3]
>>> hiver
['Janvier', 'Fevrier', 'Mars']
>>> printemps = mois[3:6]
>>> printemps
['Avril', 'Mai', 'Juin']
>>> ete = mois[6:9]
>>> ete
['Juillet', 'Aout', 'Septembre']
>>> automne = mois[9:]
```

```
>>> automne
['Octobre', 'Novembre', 'Decembre']
>>>
>>> saisons = [hiver, printemps, ete, automne]
>>> saisons[2]
['Juillet', 'Aout', 'Septembre']
>>> saisons[1][0]
'Avril'
>>> saisons[1:2]
[['Avril', 'Mai', 'Juin']]
>>> saisons[:]
[['Janvier', 'Fevrier', 'Mars'], ['Avril', 'Mai', 'Juin'], \
↪ ['Juillet', 'Aout', 'Septembre'], ['Octobre', 'Novembre', \
↪ 'Decembre']]
>>> saisons[:][1]
['Avril', 'Mai', 'Juin']
```

Solution (exercice 7) Énoncé

```
def split_list(L):
    L_imp = []
    L_pai = []
    for x in L:
        if x % 2 == 0:
            L_pai.append(x)
        else:
            L_imp.append(x)
    return [L_imp, L_pai]
```

Solution (exercice 8) Énoncé

```
def echangeEnPlace(L, i, j):
    M = copy.deepcopy(L)
    M[i], M[j] = M[j], M[i]
    return M
>>> L = [1, 2, 3]
>>> echangeEnPlace(L, 0, 2)
[3, 2, 1]
>>> L # elle est préservée
[1, 2, 3]
```

Solution (exercice 9) Énoncé

```
>>> L = [5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]
>>> L.remove(5)
>>> L
[1, 1, 2, 5, 6, 3, 4, 4, 4, 2]
>>> L.remove(1)
>>> L
[1, 2, 5, 6, 3, 4, 4, 4, 2]
>>> L.remove(2)
>>> L
[1, 5, 6, 3, 4, 4, 4, 2]
>>> L.remove(4)
>>> L
[1, 5, 6, 3, 4, 4, 2]
>>> L.remove(4)
```

Solution (exercice 10) Énoncé

```
def somme(L):
    """
    Retourne la somme des éléments de L
    """
    S = 0
    for x in L:
        S += x
    return S
```

Solution (exercice 11) Énoncé

```
def appartient(x, L):
    """
    retourne True si x est dans L
    """
    for y in L:
        if y == x:
            return True
    return False
```

```
def indice(x,L):
    for i in range(len(L)):
        if L[i] == x:
```

```
            return i
```

```
def suppr_doublon(L):
    L_prim = []
    for x in L:
        if appartient(x, L_prim) == False:
            L_prim.append(x)
    return L_prim
```

```
>>> L = [1, 2, 2, 3, 5, 4, 5]
>>> suppr_doublon(L)
[1, 2, 3, 5, 4]
```

Solution (exercice 12) Énoncé

1.
```
def maximum(L):
    """
    Retourne la valeur du maximum de L
    """
    maxi = L[0]
    for x in L:
        if x > maxi:
            maxi = x
    return maxi
```

2.
```
def maximum(L):
    """
    Retourne la valeur du maximum de L
    """
    maxi = L[0]
    for x in L[1:]:
        if x > maxi:
            maxi = x
    return maxi
```

3.
```
def minimum(L):
    """
    Retourne la valeur du minimum de L
    """
    mini = L[0]
    for x in L[1:]:
        if x < mini:
            mini = x
    return mini
```

4. Un parcours en indice est ici nécessaire.
```
def maximum_premind(L):
    """
    Retourne le maximum de L, et renvoie le premier indice \
    ↪ où il apparaît
    """
    maxi = L[0]
    ind_maxi = 0
    for k in range(1, len(L)):
        if L[k] > maxi:
            maxi = L[k]
            ind_maxi = k
    return maxi, ind_maxi
```
Pour avoir le dernier indice, on remplace la condition `L[k] > maxi` par `L[k] >= maxi`.

5.
```
def maximum_occur(L):
    """
    Retourne le maximum de L, et renvoie la liste des \
    ↪ occurences où il apparaît
    """
    # Recherche du maximum
    maxi = L[0]
    for k in range(1, len(L)):
        if L[k] > maxi:
            maxi = L[k]
    # Recherche des indices
    ind_maxi = []
    for k in range(len(L)):
        if L[k] == maxi:
            ind_maxi.append(k)
    return maxi, ind_maxi

def maximum_occur_bis(L):
    """
    Retourne le maximum de L, et renvoie la liste des \
    ↪ occurences où il apparaît.
    Un seul parcours de la liste ici.
    """
    maxi = L[0]
```

```
    ind_maxi = [0]
    for k in range(1, len(L)):
        if L[k] == maxi:
            ind_maxi.append(k)
        if L[k] > maxi:
            maxi = L[k]
            # découverte d'un nouveau potentiel max, on \
            ↪ vide la liste
            ind_maxi = [k]
    return maxi, ind_maxi
```

Solution (exercice 13) Énoncé

```
def renverse(L):
    """
    retourne la liste des éléments de L écrits dans l'autre \
    ↪ sens
    """
    L_renv = []
    for i in range(len(L)):
        L_renv.append(L[-i-1])
    return L_renv
```

Autres solutions possibles : utiliser un `range` à rebours, ou encore une liste par compréhension.

Solution (exercice 14) Énoncé Nous proposons plusieurs possibilités.

```
def decalage(L):
    """
    Décale les éléments de L vers la droite, ajoute zéro au \
    ↪ début
    (par modif. d'une copie)
    """
    tete = L[0]
    del L[0]
    L.append(tete)

def decalage_bis(L):
    """
    Décale les éléments de L vers la droite, ajoute zéro au \
    ↪ début
    (par boucle for sur une copie)
    """
    M = [0 for _ in L]
    for k in range (len(L)):
        M[k-1] = L[k] # pour k=0 c'est le dernier élément M[-1]
    return M

def decalage_bisbis(L):
    """
    Décale les éléments de L vers la droite, ajoute zéro au \
    ↪ début
    (par concaténation)
    """
    return L[1:len(L)] + [L[0]]
```

Solution (exercice 15) Énoncé

```
def Applique_Fonction(L, f):
    """
    retourne une liste où f a été appliquée à chaque élément \
    ↪ de L
    """
    return [f(x) for x in L]

def Calc_Somme_Carre(n):
    """
    retourne la somme des n premiers carrés
    """
    L = list(range(0, n+1))
    return somme(L, lambda x:x**2)
```

Plutôt que d'"utiliser l'instruction **lambda**, on aurait pu définir une fonction `f` à l'aide d'un **return** classique.

Solution (exercice 16) Énoncé

```
def min_max(L):
    """
    Cherche le minimum et le maximum de L
    """
    maxi = L[0]
    mini = L[0]
    for k in range(1, len(L)):
        if L[k] > maxi:
```

```
            maxi = L[k]
        elif L[k] < mini:
            mini = L[k]
    return mini, maxi
```

Cette fonction permet notamment de retourner facilement l'étendue d'une série statistique.

Solution (exercice 17) Énoncé

```
def inserer(L, x, i):
    L.append(x)
    # on fait descendre ensuite x au bon endroit par \
    ↪ permutations successives
    for k in range(i, len(L)):
        L[k], L[-1] = L[-1], L[k]
>>> L = [1, 2, 3, 4]
>>> inserer(L, -1, 2)
>>> L
[1, 2, -1, 3, 4]
```

Solution (exercice 18) Énoncé

```
def deux_maximum(L):
    """
    retourne les deux plus grands éléments, où seulement le \
    ↪ plus grand si L est de taille 1
    """
    if len(L) == 1:
        return L[0]
    else:
        if L[0] < L[1]:
            premier_max = L[1]
            deuxieme_max = L[0]
        else:
            premier_max = L[0]
            deuxieme_max = L[1]

        for x in L[2:]:
            if x > premier_max:
                deuxieme_max = premier_max
                premier_max = x
            elif x > deuxieme_max:
                deuxieme_max = x
        return deuxieme_max
>>> L = [1]
>>> deux_maximum(L)
1
>>> L = [2, 1, 2, 0]
>>> deux_maximum(L)
2
```

Solution (exercice 19) Énoncé

```
def cherche_carac(x, ch):
    for y in ch:
        if y == x:
            return True
    return False
```

Solution (exercice 20) Énoncé

```
def recherche_mot(m, t):
    l = len(m)
    if len(m) > len(t):
        return False
    else:
        for i in range(len(t)-l+1):
            if m == t[i:i+l]:
                return True
    return False

def est_adn(t):
    Lettres = ["A", "T", "G", "C"]
    for lettre in t:
        if lettre not in Lettres:
            return False
    return True

def codonstop(t):
    return recherche_mot("TAA",t) != False or \
    ↪ recherche_mot("TAG",t) != False or \
    ↪ recherche_mot("TGC",t) != False
>>> recherche_mot("noel", "joyeux noel")
```

```
True
>>> est_adn("noel")
False
>>> est_adn("TGCA")
True
>>> codonstop("TAGC")
True
>>> codonstop("TGTG")
False
```

Solution (exercice 21) Énoncé La version naïve ci-après ne fonctionne pas.

```
def compte_commun_naif(c1, c2):
    S = 0
    for c in c1:
        if c in c2:
            S += 1
    return S
```

En effet, si une lettre dans `c1` apparaît plusieurs fois dans `c1` et est présente dans `c2`, alors le compteur sera mis à jour plusieurs fois. On va donc devoir tenir du fait qu'une lettre a déjà été rencontré ou non.

```
def compte_commun(c1, c2):
    S = 0
    L = [] # lettres déjà rencontrées dans c1 et présentes \
    ↪ dans c2
    for c in c1:
        if c in c2 and c not in L:
            S += 1
            L.append(c)
    return S
>>> compte_commun_naif("ebe", "abcdef")
3
>>> compte_commun("ebe", "abcdef")
2
```

Solution (exercice 22) Énoncé

On peut commencer par créer une liste correspondant à l'alphabet, puis on retournera l'indice de chaque lettre dans cette liste.

```
def codage(c):
    code = []
    for x in c:
        i = 0
        while x != alphabet[i]:
            i += 1
        code.append(i)
    return code
>>> codage("bonjour")
[1, 14, 13, 9, 14, 20, 17]
```

```
def decodage(L):
    M = [alphabet[i] for i in L]
    return "".join(M)
>>> decodage(codage("bonjour"))
'bonjour'
```

```
def decalage(c, n):
    L = codage(c)
    L_decal = [(i + n)%26 for i in L]
    return decodage(L_decal)
```