Les listes : compléments

Dans ce chapitre nous allons plus loin avec les méthodes associées aux listes, ainsi que d'autres caractéristiques très puissantes telles que les tests d'appartenance qu'on a déjà parfois pu rencontrer.

1. Méthodes associées aux listes

Les listes possèdent de nombreuses méthodes qui leur sont propres et qui peuvent se révéler très pratiques.

Ici, on peut voir une méthode comme une fonction qui agit sur l'objet auquel elle est attachée, syntaxiquement, par un point.

Le but de ce qui suit est de découvrir quelques méthodes utiles et les utiliser.

(a) .append() (déjà rencontré)

La méthode **.append()** , que l'on a déjà vu au TP 3, ajoute un élément à la fin d'une liste :

```
Entrée[5]:
1 L = [1,2,3]
2 L.append(8)
3 L
Sortie[5]: [1, 2, 3, 8]
```

Remarque

i. ce qui est équivalent à :

```
Entrée[3]:
1 L = [1,2,3]
2 L = L+[8]
3 L
Sortie[3]: [1, 2, 3, 8]
```

ou

```
Entrée[4]:
1 L = [1,2,3]
2 L += [8]
3 L
Sortie[4]: [1, 2, 3, 8]
```

ii. On préfère généralement la version avec **.append()** qui est considérée plus compacte et facile à lire.

(b) .insert()

```
Entrée[7]:
1 L = [43,27,4,7,65,76,987]
2 L.insert(3,11)
3 L
Sortie[7]: [43, 27, 4, 11, 7, 65, 76, 987]
```

- Quel est est l'objet inséré ? Avec quel indice ?
- Quel est le rôle de la méthode `.insert()` ?

(c) del

L'instruction **del** supprime un élément d'une liste à un indice déterminé :

```
Entrée[9]:
1 L = [43,27,4,7,65,76,987]
2 del L[3]
3 L
Sortie[9]: [43, 27, 4, 65, 76, 987]
```

- Quel est le rôle de l'instruction `del` ?

Remarque

L'instruction **del** est une instruction générale de Python, utilisable pour d'autres objets que des listes. Celle-ci ne prend pas de parenthèses.

(d) .remove()

```
Entrée[11]:
1 L = [1,2,3,4,8,4]
2 L.remove(4)
3 L
Sortie[11]: [1, 2, 3, 8, 4]
```

```
Entrée[10]:
1 L = [1,2,3,4,8,4]
2 L.remove(3)
3 L
Sortie[10]: [1, 2, 4, 8, 4]
```

- Quel est le rôle de la méthode **`.remove()`** ?

(e) .sort()

```
Entrée[12]:
1 L = [1,2,543,6,5,4,65]
2 L.sort()
3 L
Sortie[12]: [1, 2, 4, 5, 6, 65, 543]
```

```
Entrée[13]:
1 L=['girafe','tigre','singe','abeille']
2 L.sort()
3 L
Sortie[13]: ['abeille', 'girafe', 'singe', 'tigre']
```

- Quel est le rôle de la méthode **`.sort()`** ?

(f) .reverse()

```
Entrée[14]:
1 L=[97,8,69,3,6,86,308]
2 L.reverse()
3 L

Sortie[14]: [308, 86, 6, 3, 69, 8, 97]
```

- Quel est le rôle de la méthode **`.reverse()`** ?

(g) .count()

```
Entrée[16]:
1 L=[8,97,8,69,3,6,86,30,8]
2 L.count(8)

Sortie[16]: 3
```

- Quel est le rôle de la méthode **`.count()`** ?

(h) Remarque importante

De nombreuses méthodes ci-dessus (.append(), .sort(), etc.) modifient la liste « en direct »mais ne renvoient rien, c'est-à-dire qu'elles ne renvoient pas d'objet récupérable dans une variable. Il s'agit donc de fonction particulière qui fait une action mais qui ne renvoie rien.

Pensez-y dans vos utilisations futures des listes.

2. Test d'appartenance (déjà rencontré)

L'opérateur **in** teste si un élément fait partie d'une liste et renvoie un booléen : True ou False

```
Entrée[17]:
1 L=[8,97,8,69,3,6,86,30,8]
2 5 in L
Sortie[17]: False
```

La variation avec **not** permet, *a contrario*, de vérifier qu'un élément n'est pas dans une liste :

```
Entrée[18]:
1 L=[8,97,8,69,3,6,86,30,8]
2 5 not in L
Sortie[18]: True
```

3. Remarques importante

- **Prédire la sortie de ces deux programmes.**

```
3 L=[1,2,3,4,5,6]
4 for i in L:
5     L.remove(i)
6 print(L)
```

```
3 L=[1,2,3,4,5,6]
4 for i in range(len(L)):
5     L.remove(L[i])
6 print(L)
```

- **Remarque importante à propos des copies de listes**

```
1 L=[1,2,3]
2 Liste = L
3 L.append(4)
4 print(Liste)
```

Prédire la valeur de **Liste**.

4. Applications

(a) Tri de liste

Soit la liste de nombres [8, 3, 12.5, 45, 25.5, 52, 1].

Triez les nombres de cette liste par ordre croissant, **sans** utiliser la méthode **`.sort()`**.

Les fonctions et méthodes **`min()`**, **`.append()`** et **`.remove()`** peuvent être utilisées.

(b) Séquence d'ADN complémentaire inverse

Rappel : la séquence complémentaire s'obtient en remplaçant A par T, T par A, C par G et G par C.

i. Créez une fonction **comp_inv()** qui prend comme argument une séquence d'ADN sous la forme d'une chaîne de caractères, qui renvoie la séquence complémentaire inverse sous la forme d'une autre chaîne de caractères et qui utilise des méthodes associées aux listes.

ii. Utilisez cette fonction pour transformer la séquence d'ADN TCTGTTAACCATCCACTTCG en sa séquence complémentaire inverse.

Rappel : la séquence complémentaire inverse doit être « inversée ». Par exemple, la séquence complémentaire inverse de la séquence ATCG est CGAT.

(c) Doublons

Créez une fonction **`supp_tri`** qui :

- prend pour argument une liste L
- renvoie la liste L sans les doublons et triée

Exemple

```
In [1]: L=[5,1,1,2,5,6,3,4,4,4,2]

In [2]: supp_tri(L)
Out[2]: [1, 2, 3, 4, 5, 6]
```

(d) Le nombre mystère

Trouvez le nombre mystère qui répond aux conditions suivantes :

i. *Il est composé de 3 chiffres*.
ii. *Il est pair*.
iii. *Deux de ses chiffres sont identiques*.
iv. *La somme de ses chiffres est égale à 7*.

Pour pouvoir appliquer les méthodes du TP, il est nécessaire de convertir les entiers en liste.
Vous définirez une fonction **`int_to_list`** qui pend pour argument un entier **`n`** et qui renvoie la liste de ses chiffres. Ensuite, vous suivrez une méthode dite « *brute force* », c'est-à-dire : vous utiliserez une boucle afin de tester les entiers possibles et à chaque itération tester les différentes conditions.

(e) Triangle de Pascal

Voici le début du triangle de Pascal :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

i. Comment une ligne est construite à partir de la précédente ?

ii. Codez cette construction en Python.

iii. Généralisez à l'aide d'une boucle.