11 Plus sur les listes

Nous avons vu les listes dès le chapitre 4 et les avons largement utilisées depuis le début de ce cours. 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 ou les listes de compréhension.

11.1 Méthodes associées aux listes

Comme pour les chaînes de caractères, les listes possèdent de nombreuses **méthodes** qui leur sont propres et qui peuvent se révéler très pratiques. On rappelle qu'une méthode est une fonction qui agit sur l'objet auquel elle est attachée par un point.

11.1.1 `.append()`

La méthode `.append()`, que l'on a déjà vu au chapitre 4 *Listes*, ajoute un élément à la fin d'une liste :

```
>>> a = [1, 2, 3]
>>> a.append(5)
>>> a
[1, 2, 3, 5]
```

qui est équivalent à :

```
>>> a = [1, 2, 3]
>>> a = a + [5]
>>> a
[1, 2, 3, 5]
```

Conseil : préférez la version avec `.append()` qui est plus compacte et facile à lire.

11.1.2 `.insert()`

La méthode `.insert()` insère un objet dans une liste à un indice déterminé :

```
>>> a = [1, 2, 3]
>>> a.insert(2, -15)
>>> a
[1, 2, -15, 3]
```

11.1.3 `del`

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

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> a
[1, 3]
```

Remarque

Contrairement aux méthodes associées aux listes présentées dans cette rubrique, `del` est une instruction générale de Python, utilisable pour d'autres objets que des listes. Celle-ci ne prend pas de parenthèse.

11.1.4 `.remove()`

La méthode `.remove()` supprime un élément d'une liste à partir de sa valeur :

```
>>> a = [1, 2, 3]
>>> a.remove(3)
>>> a
[1, 2]
```

S'il y a plusieurs fois la même valeur dans la liste, seule la première est retirée. Il faut appeler la méthode `.remove()` autant de fois que nécessaire pour retirer toutes les occurences d'un même élément :

```
>>> a = [1, 2, 3, 4, 3]
>>> a.remove(3)
>>> a
[1, 2, 4, 3]
>>> a.remove(3)
>>> a
[1, 2, 4]
```

11.1.5 `.sort()`

La méthode `.sort()` trie les éléments d'une liste du plus petit au plus grand:

```
>>> a = [3, 1, 2]
>>> a.sort()
>>> a
[1, 2, 3]
```

L'argument `reverse=True` spécifie le tri inverse, c'est-à-dire du plus grand au plus petit élément :

```
>>> a = [3, 1, 2]
>>> a.sort(reverse=True)
>>> a
[3, 2, 1]
```

11.1.6 `sorted()`

La fonction `sorted()` trie également une liste. Contrairement à la méthode précédente `.sort()`, cette fonction renvoie la liste triée et ne modifie pas la liste initiale :

```
>>> a = [3, 1, 2]
>>> sorted(a)
[1, 2, 3]
>>> a
[3, 1, 2]
```

La fonction `sorted()` supporte aussi l'argument `reverse=True` :

```
>>> a = [3, 1, 2]
>>> sorted(a, reverse=True)
[3, 2, 1]
>>> a
[3, 1, 2]
```

11.1.7 `.reverse()`

La méthode `.reverse()` inverse une liste :

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

11.1.8 `.count()`

La méthode `.count()` compte le nombre d'éléments (passés en argument) dans une liste :

```
>>> a = [1, 2, 4, 3, 1, 1]
>>> a.count(1)
3
>>> a.count(4)
1
>>> a.count(23)
0
```

11.1.9 Particularités des méthodes associées aux listes

- De nombreuses méthodes mentionnées précédemment (`.append()` , `.sort()` , etc.) modifient la liste mais ne renvoient rien, c'est-à-dire qu'elles ne renvoient pas d'objet récupérable dans une variable. Il s'agit d'un exemple d'utilisation de méthode (donc de fonction particulière) qui fait une action mais qui ne renvoie rien. Pensez-y dans vos utilisations futures des listes : même si `var = liste.reverse()` est une instruction Python valide, elle n'a aucun intérêt, préférez-lui `liste.reverse()` .

Remarque

Pour exprimer la même idée, la documentation parle de modification de la liste « sur place » (*in place* en anglais) :

```
>>> liste = [1, 2, 3]
>>> help(liste.reverse)
Help on built-in function reverse:

reverse() method of builtins.list instance
    Reverse *IN PLACE*.
```

Cela signifie que la liste est modifiée « sur place », c'est-à-dire **dans la méthode** au moment où elle s'exécute. La liste étant modifiée « en dur » dans la méthode, cette dernière ne renvoie donc rien. L'explication du mécanisme sous-jacent vous sera donnée dans la rubrique 12.4 *Portée des listes* du chapitre 12 *Plus sur les fonctions*.

- Certaines méthodes ou instructions des listes décalent les indices d'une liste (par exemple `.insert()` , `del` , etc.).
- Enfin, pour obtenir une liste exhaustive des méthodes disponibles pour les listes, utilisez la fonction `dir(ma_liste)` (`ma_liste` étant une liste).

11.2 Construction d'une liste par itération

La méthode `.append()` est très pratique car on peut l'utiliser pour construire une liste au fur et à mesure des itérations d'une boucle.

Pour cela, il est commode de définir préalablement une liste vide de la forme `ma_liste = []` . Voici un exemple où une chaîne de caractères est convertie en liste :

```
>>> seq = "CAAAGGTAACGC"
>>> seq_list = []
>>> seq_list
[]
>>> for base in seq:
```

```
...     seq_list.append(base)
...
>>> seq_list
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Remarquez que dans cet exemple, vous pouvez directement utiliser la fonction `list()` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, etc.) et qui renvoie une liste :

```
>>> seq = "CAAAGGTAACGC"
>>> list(seq)
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Cette méthode est certes plus simple, mais il arrive parfois qu'on doive utiliser des boucles tout de même, comme lorsqu'on lit un fichier. On rappelle que l'instruction `list(seq)` convertit un objet de type chaîne de caractères en un objet de type liste (il s'agit donc d'une opération de *casting*). De même que `list(range(10))` convertit un objet de type `range` en un objet de type `list`.

11.3 Test d'appartenance

L'opérateur `in` teste si un élément fait partie d'une liste.

```
liste = [1, 3, 5, 7, 9]
>>> 3 in liste
True
>>> 4 in liste
False
>>> 3 not in liste
False
>>> 4 not in liste
True
```

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

11.4 Copie de listes

Il est très important de savoir que l'affectation d'une liste (à partir d'une liste préexistante) crée en réalité une **référence** et non une **copie** :

```
>>> x = [1, 2, 3]
>>> y = x
>>> y
[1, 2, 3]
>>> x[1] = -15
>>> x
[1, -15, 3]
```

```
>>> y
[1, -15, 3]
```

Vous voyez que la modification de `x` modifie `y` aussi ! Pour comprendre ce qui se passe nous allons de nouveau utiliser le site *Python Tutor* avec cet exemple (Figure 1) :

Figure 1. Copie de liste.

Techniquement, Python utilise des pointeurs (comme dans le langage de programmation C) vers les mêmes objets. *Python Tutor* l'illustre avec des flèches qui partent des variables `x` et `y` et qui pointent vers la même liste. Donc, si on modifie la liste `x`, la liste `y` est modifiée de la même manière. Rappelez-vous de ceci dans vos futurs programmes car cela pourrait avoir des effets désastreux !

Pour éviter ce problème, il va falloir créer une copie explicite de la liste initiale. Observez cet exemple :

```
>>> x = [1, 2, 3]
>>> y = x[:]
>>> x[1] = -15
>>> y
[1, 2, 3]
```

L'instruction `x[:]` a créé une copie « à la volée » de la liste `x`. Vous pouvez utiliser aussi la fonction `list()` qui renvoie explicitement une liste:

```
>>> x = [1, 2, 3]
>>> y = list(x)
>>> x[1] = -15
>>> y
[1, 2, 3]
```

Si on regarde à nouveau dans *Python Tutor* (Figure 12), on voit clairement que l'utilisation d'une tranche `[:]` ou de la fonction `list()` crée des copies explicites. Chaque flèche pointe vers une liste différente, indépendante des autres.

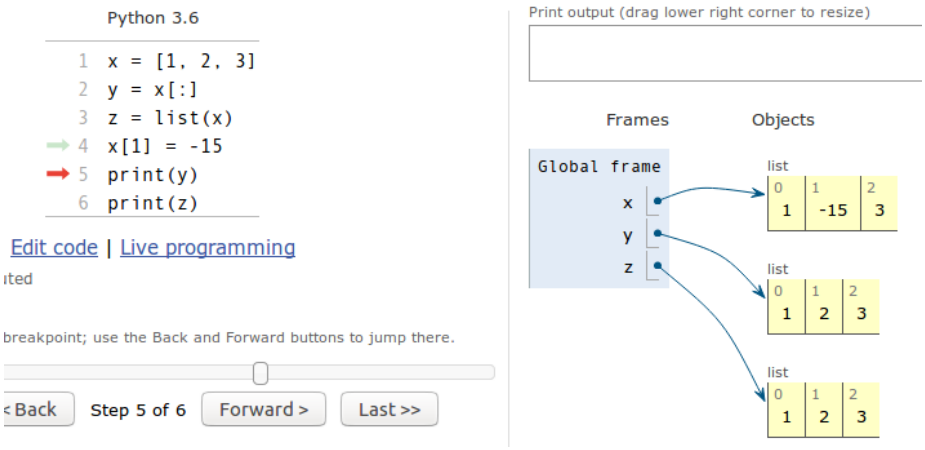

Figure 2. Copie de liste avec une tranche [:] et la fonction list().

Attention, les deux astuces précédentes ne fonctionnent que pour les listes à une dimension, autrement dit les listes qui ne contiennent pas elles-mêmes d'autres listes. Voyez par exemple :

```
>>> x = [[1, 2], [3, 4]]
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> x[1][1] = 55
>>> x
[[1, 2], [3, 55]]
>>> y
[[1, 2], [3, 55]]
```

et

```
>>> y = list(x)
>>> x[1][1] = 77
>>> x
[[1, 2], [3, 77]]
>>> y
[[1, 2], [3, 77]]
```

La méthode de copie qui **fonctionne à tous les coups** consiste à appeler la fonction `deepcopy()` du module *copy*.

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

11.5 Liste de compréhension

Conseil : pour les débutants, vous pouvez passer cette rubrique.

En Python, la notion de liste de compréhension (ou compréhension de listes) représente une manière originale et très puissante de générer des listes. La syntaxe de base consiste au moins en une boucle `for` au sein de crochets précédés d'une variable (qui peut être la variable d'itération ou pas):

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [2 for i in range(10)]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

Pour plus de détails, consultez à ce sujet le site de Python et celui de Wikipédia.

Voici quelques exemples illustrant la puissance des listes de compréhension.

11.5.1 Nombres pairs compris entre 0 et 30

```
>>> print([i for i in range(31) if i % 2 == 0])
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

11.5.2 Jeu sur la casse des mots d'une phrase

```
>>> message = "C'est sympa la BioInfo"
>>> msg_lst = message.split()
>>> print([[m.upper(), len(m)] for m in msg_lst])
[["C'EST", 5], ['SYMPA', 5], ['LA', 2], ['BIOINFO', 7]]
```

11.5.3 Formatage d'une séquence avec 60 caractères par ligne

Exemple d'une séquence constituée de 150 alanines :

```
# Exemple d'une séquence de 150 alanines.
>>> seq = "A" * 150
>>> width = 60
>>> seq_split = [seq[i:i+width] for i in range(0, len(seq), width)]
>>> print("\n".join(seq_split))
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
```

11.5.4 Formatage FASTA d'une séquence (avec la ligne de commentaire)

Exemple d'une séquence constituée de 150 alanines :

```
>>> com = "Séquence de 150 alanines"
>>> seq = "A" * 150
>>> width = 60
>>> seq_split = [seq[i:i+width] for i in range(0, len(seq), width)]
>>> print(">"+com+"\n"+"\n".join(seq_split))
>séquence de 150 alanines
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
```

11.5.5 Sélection des carbones alpha dans un fichier pdb

Exemple avec la structure de la barstar :

```
>>> with open("1bta.pdb", "r") as f_pdb:
...     CA_lines = [line for line in f_pdb if line.startswith("ATOM")
                                            and line[12:16].strip() ==
"CA"]
...
>>> print(len(CA_lines))
89
```

11.6 Exercices

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

11.6.1 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 fonction `sort()`. Les fonctions et méthodes `min()`, `.append()` et `.remove()` vous seront utiles.

11.6.2 Séquence d'ADN aléatoire

Créez une fonction `seq_alea()` qui prend comme argument un entier positif `taille` représentant le nombre de bases de la séquence et qui renvoie une séquence d'ADN aléatoire sous forme d'une liste de bases. Utilisez la méthode `.append()` pour ajouter les différentes bases à la liste et la fonction `random.choice()` du module *random* pour choisir une base parmi les 4 possibles.

Utilisez cette fonction pour générer aléatoirement une séquence d'ADN de 15 bases.

11.6.3 Séquence d'ADN complémentaire inverse

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.

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`.

11.6.4 Doublons

Soit la liste de nombres `liste = [5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]`.

À partir de `liste`, créez une nouvelle liste sans les doublons, triez-la et affichez-la.

11.6.5 Séquence d'ADN aléatoire 2

Créez une fonction `seq_alea_2()` qui prend comme argument un entier et quatre *floats* représentant respectivement la longueur de la séquence et les pourcentages de chacune des 4 bases A, T, G et C. La fonction générera aléatoirement une séquence d'ADN qui prend en compte les contraintes fournies en arguments et renverra la séquence sous forme d'une liste.

Utilisez cette fonction pour générer aléatoirement une séquence d'ADN de 50 bases contenant 10 % de A, 30 % de T, 50 % de G et 10 % de C.

Conseil : la fonction `random.shuffle()` du module *random* vous sera utile.

11.6.6 Le nombre mystère

Trouvez le nombre mystère qui répond aux conditions suivantes :

- *Il est composé de 3 chiffres*.
- *Il est strictement inférieur à 300*.
- *Il est pair*.
- *Deux de ses chiffres sont identiques*.
- *La somme de ses chiffres est égale à 7*.

On vous propose d'employer une méthode dite « *brute force* », c'est-à-dire d'utiliser une boucle et à chaque itération de tester les différentes conditions.

11.6.7 Triangle de Pascal (exercice +++)

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
[...]
```

Déduisez comment une ligne est construite à partir de la précédente. Par exemple, à partir de la ligne 2 (`1 1`), construisez la ligne suivante (ligne 3 : `1 2 1`) et ainsi de suite.

Implémentez cette construction en Python. Généralisez à l'aide d'une boucle.

Écrivez dans un fichier `pascal.out` les 10 premières lignes du triangle de Pascal.