10 Plus sur les listes

10.1 Propriétés des listes

Jusqu'à maintenant, nous avons toujours utilisé des listes qui étaient déjà remplies. Nous savons comment modifier un de ses éléments, mais il est aussi parfois très pratique d'en remanier la taille (*e. g.* ajouter, insérer ou supprimer un ou plusieurs éléments, etc.). Les listes possèdent à cet effet des méthodes qui leurs sont propres. Observez les exemples suivants :

append() pour ajouter un élément à la fin d'une liste.

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

qui est équivalent à

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

insert() pour insèrer un object dans une liste avec un indice déterminé.

```
>>> x.insert(2,-15)
>>> x
[1, 2, -15, 3, 5]
```

del pour supprimer un élément d'une liste à une indice déterminé.

```
>>> del x[1]
>>> x
[1, -15, 3, 5]
```

remove() pour supprimer un élément d'une liste à partir de sa valeur

```
>>> x.remove(5)
>>> x
[1, -15, 3]
```

sort() pour trier une liste.

```
>>> x.sort()
>>> x
[-15, 1, 3]
```

reverse() pour inverser une liste.

```
>>> x.reverse()
>>> x
[3, 1, -15]
```

count() pour compter le nombre d'éléments (passé en argument) dans une liste.

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

Remarque 1 : attention, une liste remaniée n'est pas renvoyée ! Pensez-y dans vos utilisations futures des listes.

Remarque 2 : attention, certaines fonctions ci-dessus décalent les indices d'une liste (par exemple `insert()`, `del` etc).

La méthode `append()` est particulièrement pratique car elle permet de 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 `maliste = []`. 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 vous pouvez directement utiliser la fonction `list()` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, tuples, 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 que l'on doive utiliser les boucles tout de même, comme lorqu'on lit un fichier.

10.2 Test d'appartenance

L'inscription `in` permet de tester 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 élement n'est pas dans une liste.

10.3 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
>>> y
[1, -15, 3]
```

Vous voyez que la modification de `x` modifie `y` aussi. Rappelez-vous de ceci dans vos futurs programmes car cela pourrait avoir des effets désastreux ! Techniquement, Python utilise des pointeurs (comme dans le langage C) vers les mêmes objets et ne crée pas de copie à moins que vous n'en ayez fait la demande explicitement. Regardez cet exemple :

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

Dans l'exemple précédent, `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]
```

Attention, les deux techniques précédentes ne fonctionnent que pour les listes à une dimension, autrement dit les listes qui ne contienent pas elles-mêmes d'autres listes.

```
>>> x = [[1,2],[3,4]]
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> y[1][1] = 55
>>> y
[[1, 2], [3, 55]]
>>> x
[[1, 2], [3, 55]]
>>> y = list(x)
>>> y[1][1] = 77
>>> y
[[1, 2], [3, 77]]
>>> x
[[1, 2], [3, 77]]
```

La méthode de copie qui **marche 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)
>>> y[1][1] = 99
>>> y
[[1, 2], [3, 99]]
>>> x
[[1, 2], [3, 4]]
```

10.4 Exercices

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

1. 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 `min()`,`append()` et `remove()` vous seront utiles).
2. Générez aléatoirement une séquence nucléique de 20 bases en utilisant une liste et la méthode `append()`.
3. Transformez la séquence nucléique `TCTGTTAACCATCCACTTCG` en sa séquence complémentaire inverse. N'oubliez pas que la séquence complémentaire doit être inversée, pensez aux méthodes des listes !
4. Soit la liste de nombres `[5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]`. Enlevez les doublons de cette liste, triez-là et affichez-là.
5. Générez aléatoirement une séquence nucléique de 50 bases contenant 10 % de A, 50 % de G, 30 % de T et 10 % de C.
6. Exercice +++. **Triangle de Pascal**
   Voici le début du triangle de Pascal :

   ```
   1
   11
   121
   1331
   14641
   ...
   ```

   Comprenez comment une ligne est construite à partir de la précente. À partir de l'ordre 1 (ligne 2, `11`), générez l'ordre suivant (`121`). Vous pouvez utiliser une liste préalablement générée avec `range()`. Généralisez à l'aide d'une boucle. Écrivez dans un fichier `pascal.out` les lignes du triangle de Pascal de l'ordre 1 jusqu'à l'ordre 10.

11 Dictionnaires et tuples

11.1 Dictionnaires

Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets, c-à-d qu'il n'y a pas de notion d'ordre (*i. e.* pas d'indice). On accède aux **valeurs** d'un dictionnaire par des **clés**. Ceci semble un peu confus ? Regardez l'exemple suivant :

```
>>> ani1 = {}
>>> ani1['nom'] = 'girafe'
>>> ani1['taille'] = 5.0
>>> ani1['poids'] = 1100
>>> ani1
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
>>> ani1['taille']
5.0
```

En premier, on définit un dictionnaire vide avec les symboles `{}` (tout comme on peut le faire pour les listes avec `[]`). Ensuite, on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs (une par clé). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste). Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style `dictionnaire['cle']`.

Méthodes keys() et values()

Les méthodes `keys()` et `values()` renvoient, comme vous vous en doutez, les clés et les valeurs d'un dictionnaire (sous forme de liste) :

```
>>> ani1.keys()
['nom', 'poids', 'taille']
>>> ani1.values()
['girafe', 1100, 5.0]
```

On peut aussi initialiser toutes les clés d'un dictionnaire en une seule opération :

```
>>> ani2 = {'nom':'singe', 'poids':70, 'taille':1.75}
```

Liste de dictionnaires

En créant une liste de dictionnaires possédant les mêmes clés, on obtient une structure qui ressemble à une base de données :

```
>>> animaux = [ani1, ani2]
>>> animaux
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe', 'poids': 70,
>>>
>>> for ani in animaux:
...     print ani['nom']
...
girafe
singe
```

Existence d'une clef

Enfin, pour vérifier si une clé existe, vous pouvez utiliser la propriété `has_key()` :

```
>>> if ani2.has_key('poids'):
...     print "La clef 'poids' existe pour ani2"
...
La clef 'poids' existe pour ani2
```

Python permet même de simplifier encore les choses :

```
>>> if "poids" in ani2:
...     print "La clef 'poids' existe pour ani2"
...
La clef 'poids' existe pour ani2
```

Vous voyez que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes.

11.2 Tuples

Les **tuples** correspondent aux listes à la différence qu'ils sont **non modifiables**. On a vu à la section précédente que les listes pouvaient être modifiées par des références ; les tuples vous permettent de vous affranchir de ce problème. Pratiquement, ils utilisent les parenthèses au lieu des crochets :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x[2]
3
>>> x[0:2]
(1, 2)
>>> x[2] = 15
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

L'affectation et l'indiçage fonctionne comme avec les listes, mais si l'on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un autre tuple :

```
>>> x = (1,2,3)
>>> x + (2,)
(1, 2, 3, 2)
```

Remarquez que pour utiliser un tuple d'un seul élément, vous devez utiliser une syntaxe avec une virgule `(element,)`, ceci pour éviter une ambiguïté avec une simple expression. Autre particularité des tuples, il est possible d'en créer de nouveaux sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x = 1,2,3
>>> x
(1, 2, 3)
```

Toutefois, nous vous conseillons d'utiliser systématiquement les parenthèses afin d'éviter les confusions.

Enfin, on peut utiliser la fonction `tuple(sequence)` qui fonctionne exactement comme la fonction `list`, c-à-d qu'elle prend en argument un objet séquentiel et renvoie le tuple correspondant :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple("ATGCCGCGAT")
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
```

Remarque : les listes, dictionnaires, tuples sont des objets qui peuvent contenir des collections d'autres objets. On peut donc construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc.

11.3 Exercices

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

1. En utilisant un dictionnaire et la fonction `has_key()`, déterminez le nombre d'occurences de chaque acide aminé dans la séquence `AGWPSGGASAGLAILWGASAIMPGALW`. Le dictionnaire ne doit contenir que les acides aminés présents dans la séquence.
2. Soit la séquence nucléotidique suivante :

   `ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG`

   En utilisant un dictionnaire, faites un programme qui répertorie tous les mots de 2 lettres qui existent dans la sequence (AA, AC, AG, AT, etc.) ainsi que leur nombre d'occurences puis qui les affiche à l'écran.
3. Faites de même avec des mots de 3 et 4 lettres.
4. En vous basant sur les scripts précédents, extrayez les mots de 2 lettres et leur occurence sur le génome du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* NC_001133.fna. Attention, le génome complet est fourni au format fasta.
5. Créez un script `extract-words.py` qui prend en arguments un fichier genbank suivi d'un entier compris entre 1 et 4. Ce script doit extraire du fichier genbank tous les mots (ainsi que leur nombre d'occurences) du nombre de lettres passées en option.
6. Appliquez ce script sur le génome d'*Escherichia coli* : NC_000913.fna (au format fasta). Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*E. Coli* ? Comment pourrait-on en améliorer la rapidité ?
7. À partir du fichier PDB 1BTA, construisez un dictionnaire qui contient 4 clés se référant au premier carbone alpha : le numéro du résidu, puis les coordonnées x, y et z.
8. Sur le même modèle que ci-dessus, créez une liste de dictionnaires pour chacun des carbones alpha de la protéine.
9. À l'aide de cette liste, calculez les coordonnées x, y et z du barycentre de ces carbones alpha.