TP Python 3 - Listes

Listes

Les listes (objets de type `list`) sont des séquences d'objets séparés par des virgules et placés entre crochets. Les objets d'une liste ne sont pas nécessairement du même type.

```
>>> L = [17, 'abc', [1, 2, 3]]
```

Les listes sont indexées à partir de 0**.** On accède (en temps constant) à l'élément d'indice `i` d'une liste `L` par la syntaxe `L[i]`. Ainsi `L[0]` est le premier élément de `L`, `L[1]` le deuxième, etc.

```
>>> L[0]
17
>>> L[1]
'abc'
>>> L[2]
[1, 2, 3]
>>> L[3]          # on sort de la liste
IndexError: list index out of range
>>> L[2][1]       # c'est le deuxième élément de L[2]
2
```

Noter également que `L[-1]` désigne le dernier élément de `L`, `L[-2]` l'avant-dernier, etc.

```
>>> L[-1]
[1, 2, 3]
```

Les listes sont modifiables : `L[i] = x` remplace l'élément d'indice `i` de `L` par `x`.

```
>>> L[0] = 42
>>> L
[42, 'abc', [1, 2, 3]]
```

La liste vide est notée `[]`.

Exercice 1 Créer la liste `liste = [2, 0, 5, [7, 6, 17], 4]`, remplacer le 0 par 8 puis remplacer le 17 par 22.

Fonctions et méthodes associées aux listes

La méthode `pop` permet d'enlever (et de récupérer) le dernier élément d'une liste.

```
>>> L = [5, 7, 6]
>>> x = L.pop()
>>> x
6
>>> L
[5, 7]
```

La méthode `append` permet d'ajouter un élément à la fin d'une liste.

```
>>> L.append(12)
>>> L
[5, 7, 12]
```

On créera souvent des listes en ajoutant des éléments à une liste initialement vide.

```
>>> multiples = []
>>> for i in range(10):
...     multiples.append(7*i)
>>> multiples
[0, 7, 14, 21, 28, 35, 42, 49, 56, 63]
```

Exercice 2 Construire la liste des carrés des entiers compris entre 1 et 1000 :

1) Avec une boucle `for`.

2) Avec une boucle `while`.

La fonction `len` renvoie le nombre d'éléments d'une liste.

```
>>> len([1, 2, 3])
3
```

On peut concaténer deux listes, i.e. placer la deuxième à la suite de la première, avec l'opérateur `+`. On peut concaténer plusieurs copies d'une liste avec l'opérateur `*`.

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1] * 5
[1, 1, 1, 1, 1]
```

On peut tester avec `in` et `not in` si un élément appartient ou non à une liste.

```
>>> 2 in [1, 2, 3]
True
>>> 5 not in [1, 2, 3]
True
```

Pour trier une liste, on peut utiliser la fonction `sorted` (qui renvoie une nouvelle liste triée) ou la méthode `sort` (qui modifie la liste).

```
>>> L = [2, 1, 4, 3]
>>> sorted(L)
[1, 2, 3, 4]
>>> L             # L n'a pas été modifiée
[2, 1, 4, 3]
>>> L.sort()      # ne renvoie rien...
>>> L             # ... mais L a été modifiée
[1, 2, 3, 4]
```

Itération sur les listes

Pour itérer sur une liste, on peut utiliser un `range` :

```
>>> L = [5, 13, 4]
>>> for i in range(3):
...     print(L[i])
5
13
4
```

Mais il est plus efficace d'itérer directement sur la liste :

```
>>> for x in L:
...     print(x)
5
13
4
```

Si on a besoin de l'indice on utilisera la première méthode, sinon la seconde est préférable.

Exemples de fonctions agissant sur des listes

La fonction suivante renvoie la somme des éléments de la liste qu'elle reçoit en argument.

```
def somme(L):  # avec un range
    '''Renvoie la somme des éléments
    de L'''
    s = 0
    for i in range(len(L)):
        s = s + L[i]
    return s
```

```
def somme(L):  # en itérant sur L
    '''Renvoie la somme des éléments
    de L'''
    s = 0
    for x in L:
        s = s + x
    return s
```

```
>>> somme([4, 3, -2, 1])
6
```

Exercice 3 Écrire une fonction `produit` qui, recevant une liste de nombres, renvoie le produit de ses éléments.

```
>>> produit([1, 2, 3, 4, 6])
144
```

Exercice 4 Écrire une fonction `pairs_et_impairs` qui, recevant une liste d'entiers `L`, renvoie un couple de deux listes, la première contenant les éléments pairs de `L` et la seconde ses éléments impairs.

```
>>> pairs_et_impairs([1, 2, 4, 7, 3, 2, 1])
([2, 4, 2], [1, 7, 3, 1])
```

La fonction suivante renvoie le plus grand élément de la liste (supposée non vide) qu'elle reçoit en argument :

```
def maximum(L):  # avec un range
    '''Renvoie le maximum de la liste
    non vide L'''
    m = L[0]
    for i in range(1, len(L)):
        if L[i] > m:
            m = L[i]
    return m
```

```
def maximum(L):  # en itérant sur L
    '''Renvoie le maximum de la liste
    non vide L'''
    m = L[0]
    for x in L:
        if x > m:
            m = x
    return m
```

```
>>> maximum([2, 6, 4, 8, 2])
8
```

Exercice 5

1) Modifier la fonction précédente pour qu'elle renvoie le plus petit élément de `L`.

2) Si la liste `L` est de longueur n, combien la fonction `maximum` effectue-t-elle de comparaisons et d'affectations dans le meilleur des cas ? Et dans le pire des cas ?

La fonction suivante renvoie l'indice du plus grand élément de la liste qu'elle reçoit en argument. Noter qu'ici on est obligé d'itérer sur un range car on a besoin de l'indice.

```
def indice_du_maximum(L):
    '''Renvoie l'indice du plus grand élément de la liste non vide L'''
    imax = 0
    for i in range(1, len(L)):
        if L[i] > L[imax]:
            imax = i
    return imax
```

```
>>> indice_du_maximum([2, 6, 4, 8, 2])
3
```

Exercice 6 Écrire une fonction `meilleur_eleve` qui, recevant une liste de listes contenant des noms d'élèves et leurs notes à un devoir, renvoie le nom de l'élève ayant obtenu la meilleure note.

```
>>> notes = [['Antoine', 12], ['Albert', 8], ['Anatole', 15], ['Alphonse', 9]]
>>> meilleur_eleve(notes)
'Anatole'
```

La fonction suivante renvoie la liste qu'elle reçoit en argument à l'envers.

```
def renverser(L):
    '''Renvoie une copie renversée de la liste L.'''
    res = []
    n = len(L)
    for i in range(n):
        res.append(L[n-i-1])   # attention aux indices
    return res
```

```
>>> renverser([3, 4, 8, 6])
[6, 8, 4, 3]
```

Slicing

Le *slicing* (découpage en tranches en français) permet de récupérer ou de modifier un morceau d'une liste : `L[i:j]` représente la sous-liste de `L` formée des éléments dont les indices sont compris entre `i` et `j-1`. On peut spécifier un pas en écrivant `L[i:j:k]`.

```
>>> L = [2, 8, 1, 3, 5]
>>> L[2:4]            # éléments d'indices 2 et 3
[1, 3]
>>> L[2:]             # éléments d'indices 2 et plus
[1, 3, 5]
>>> L[:2]             # éléments d'indices strictement inférieurs à 2
[2, 8]
>>> L[:]              # toute la liste
[2, 8, 1, 3, 5]
>>> L[1:5:2]          # éléments d'indices 1 et 3
[8, 3]
>>> L[::-1]           # toute la liste mais à l'envers
[5, 3, 1, 8, 2]
```

Définition par compréhension

En Python, on peut définir des listes par *compréhension* :

```
>>> [i ** 2 for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

On peut ajouter une condition :

```
>>> [i ** 2 for i in range(1, 11) if i % 2 == 0]
[4, 16, 36, 64, 100]
```

Exercice 7 Construire avec cette méthode la liste des multiples de 7 compris entre 1 et 100.

Copie

La copie d'une liste en Python est une opération délicate :

```
>>> L1 = [1, 2, 3]  # on crée une liste L1
>>> L2 = L1         # on crée une copie de L1
>>> L2
[1, 2, 3]           # ça marche...
>>> L1[0] = 4       # ... mais si on modifie un élément de L1...
>>> L1
[4, 2, 3]
>>> L2
[4, 2, 3]           # ... L2 est modifiée aussi
```

Explication : quand on crée la liste `L1`, on crée en mémoire l'objet `[1, 2, 3]`, et on lui associe le nom `L1`. Quand on écrit `L2 = L1`, on ne crée pas de nouvel objet : on associe également à l'objet `[1, 2, 3]` le nom `L2` . On dit que `L1` et `L2` pointent vers le même objet. Ainsi quand celui-ci est modifié, il l'est à la fois pour `L1` et pour `L2`.
Il y a une solution simple pour remédier au problème :

```
>>> L1 = [1, 2, 3]
>>> L2 = L1[:]     # on crée un nouvel objet en mémoire
>>> L2
[1, 2, 3]
>>> L1[0] = 4
>>> L1
[4, 2, 3]
>>> L2
[1, 2, 3]          # L2 n'est pas modifiée
```

Exercices

Exercice 8

Le but de l'exercice est de programmer quelques fonctions élémentaires sur les listes. On s'interdira donc d'utiliser les fonctions Python prédéfinies : on n'utilisera que des boucles, des tests, la fonction `len` et la méthode `append`. L'opérateur de concaténation `+`, le slicing et les listes définies par compréhension sont également interdits. Par ailleurs, on veillera à ce que la ou les listes données en argument ne soient pas modifiées.

1) Écrire une fonction `indice` qui, recevant un objet `x` et une liste `L`, renvoie l'indice de la première occurrence de `x` dans `L`. Si `x` n'est pas dans `L`, la fonction doit renvoyer `False`.

```
>>> indice(2, [1, 3, 5, 2, 3, 2, 1])
3
>>> indice(4, [1, 3, 5, 2, 3, 2, 1])
False
```

2) Écrire une fonction `compter` qui, recevant un objet `x` et une liste `L`, renvoie le nombre de fois que `x` apparaît dans `L`.

```
>>> compter(5, [2, 5, 5, 1, 3, 5])
3
```

3) Écrire une fonction `concatener` qui, recevant deux listes `L1` et `L2`, renvoie la concaténation de `L1` et `L2`.

```
>>> concatener([3, 1, 5, 2], [6, 1, 2])
[3, 1, 5, 2, 6, 1, 2]
```

4) Écrire une fonction `remplacer` qui, recevant une liste `L` et deux objets `x1` et `x2`, renvoie une copie de `L` où on a remplacé toutes les occurrences de `x1` par `x2`.

```
>>> remplacer([4, 5, 3, 4, 2, 1], 4, 1)
[1, 5, 3, 1, 2, 1]
```

5) Écrire une fonction `decouper` qui, recevant une liste `L` et deux entiers `i` et `j`, renvoie la liste des éléments de `L` dont les indices sont compris entre `i` et `j` (inclus).

```
>>> decouper([1, 7, 5, 2, 4, 6], 1, 3)
[7, 5, 2]
```

6) Écrire une fonction `echanger` qui, recevant une liste `L` et deux entiers `i` et `j`, renvoie une liste où les éléments d'indices `i` et `j` de `L` ont été intervertis.

```
>>> echanger([1, 2, 3, 4], 1, 3)
[1, 4, 3, 2]
```

7) Écrire une fonction `enlever` qui, recevant un objet `x` et une liste `L`, renvoie une liste dans laquelle toutes les occurrences de `x` ont été retirées.

```
>>> enlever(1, [1, 2, 3, 1, 4])
[2, 3, 4]
```

Exercice 9 Écrire une fonction `est_triee` qui, recevant une liste, renvoie `True` si celle-ci est triée et `False` sinon. On n'utilisera ni `sorted` ni `sort`.

```
>>> est_triee([1, 4, 5, 9, 12])
True
>>> est_triee([1, 4, 5, 3, 12])
False
```

Exercice 10 Écrire une fonction `supprimer_doublons` qui, recevant une liste `L`, renvoie une liste dans laquelle chaque élément de `L` ne se trouve qu'en un seul exemplaire.

```
>>> supprimer_doublons([1, 2, 1, 3, 1, 2, 4])
[1, 2, 3, 4]
```

Exercice 11

1) Écrire une fonction `chiffres` qui, recevant un entier, renvoie la liste de ses chiffres.

```
>>> chiffres(12358)
[1, 2, 3, 5, 8]
```

2) Trouver tous les entiers naturels égaux au cube de la somme de leurs chiffres.

3) Trouver tous les entiers naturels égaux à la somme des cubes de leurs chiffres.

Exercice 12 Écrire une fonction `nombre_manquant` qui, recevant une liste de n entiers distincts compris entre 0 et n, renvoie le nombre manquant.

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

Exercice 13 Écrire une fonction `decomposer` qui, recevant un entier naturel non nul n, renvoie sa décomposition en produit de facteurs premiers : si $n = p_1^{a_1} p_2^{a_2} \dots p_r^{a_r}$ où $p_1 < p_2 < \dots < p_r$ sont premiers et les a_i sont strictement positifs, la fonction doit renvoyer la liste `[[p1, a1], [p2, a2], ..., [pr, ar]]`. Par exemple, on a $60 = 2^2 3^1 5^1$, donc `decomposer(60)` doit renvoyer `[[2, 2], [3, 1], [5, 1]]`.