TP de Python numéro 3 : Listes

*Semaine du **10 octobre**.*

Ce TP est d'une importance majeure pour toute l'année.

I. Notion de liste, commandes liées aux listes

1. Listes en Python

Définition 1. Une *liste* en python est un objet de type `list`, consistant en la **donnée successive de variables** python. Une liste est donnée par la syntaxe suivante :

- Une liste commence par un crochet ouvrant `[` et se termine par un crochet fermant `]`.
- Les contenus des variables successives données par la liste sont, entre ces crochets, séparés par une virgule.

L'ordre de ces données est pris en compte.
Deux listes sont donc égales si et seulement si elles contiennent les mêmes valeurs, dans le même ordre.

Exemple 2.

1. La commande `L=[2,3,1]` définit une liste `L` contenant successivement les valeurs 2,3 et 1.
2. Le type des objets présents dans une liste peut changer d'un objet à l'autre. Par exemple, la liste `[1,"Bonjour", True, 3]` est valide, et contient des entiers, du texte et du booléen.
3. Effectuer et comprendre, dans l'invite de commande :
```
L=[1,2,3]
G=[1,2,3,3]
H=[1,3,2]
L==G
L==H
```

Remarque. La liste *vide* est la liste `[]` ne contenant aucun élément.

2. Indexation des éléments d'une liste

Indices, longueur, accès aux éléments d'une liste

Soit `L=[`l_0`,`l_1`,...,`l_{n-1}`]` une liste.

1. Pour tout $i \in [\![0, n-1]\!]$, on dit que *le i-ième élément de* `L` est l_i. On dit également que l_i est l'élément d'*indice* i de la liste `L` ("indice" se dit "index" en anglais).
2. Le nombre n d'éléments de `L` est appelé la *longueur* de `L`. La liste vide est de longueur 0. La commande `len(L)` renvoie la longueur de la liste `L`.
3. On remarque que les éléments de `L` sont indéxés (ou numérotés) de 0 à `len(L)-1`.
4. Ce dernier point remarqué, pour tout entier positif `i` :
   (a) la commande `L[i]` renvoie le `i`-ième élément de `L` si $0 \leq$ `i` $\leq$ `len(L)` $- 1$ (autrement dit, si `i` est bien l'indice d'un élément de `L`),
   (b) la commande `L[-i]` renvoie le `i`-ième élément de `L` obtenu en comptant à partir de la fin de `L`, si celui-ci est bien défini. Par exemple, `L[-1]` renvoie le dernier élément de `L`, `L[-2]` son avant dernier élément, etc.
   (c) Dans tous les autres cas, `L[i]` et `L[-i]` renvoient une erreur à bien connaitre : "IndexError: list index out of range". Cette erreur signifie que vous tentez d'accéder à un élément de la liste `L` qui n'existe pas.

Exemple 3. Histoire sans paroles si `L= [3,4,1,2]`:

L[0]	L[1]	L[2]	L[3]
↓	↓	↓	↓
[3,	4,	1,	2]
↑	↑	↑	↑
L[-4]	L[-3]	L[-2]	L[-1]

Exercice 4. Prédire l'effet des lignes suivantes, effectuées dans l'ordre indiqué et dans l'invite de commande, puis vérifier vos prédictions.

```
1. L=[1,2,"Bonjour"]
2. len(L)
3. L[1]
4. L[2]
5. L[3]
6. L[-1]
7. L[-2]
8. L[-3]
9. L[0]
10. L[4]
11. len([5])
12. len([])
```

Remarque. Par extension, on pourra utiliser des entiers négatifs pour décrire l'indice d'un élément de `L` dans ce cours.

Exercice 5. Vous pouvez déjà faire ces deux premières fonctions. On aura plus tard des moyens beaucoup plus efficaces pour les réaliser.

1. Écrire le code d'une fonction `Python` d'entête `def testAppartenance(L,a):` prenant en entrée une liste `L` et un objet python `a`, et renvoyant `True` si `a` est un élément de `L`, et `False` sinon.
2. Écrire le code d'une fonction `Python` d'entête `def depasse(L,a):` prenant en entrée une liste `L` de nombres et un nombre `a`, et renvoyant `True` si un élément de `L` est strictement supérieur à `a`, et `False` sinon.

3. Modifications d'une liste

a) Modification d'une entrée

Les éléments d'une liste sont aussi parfois appelés ses entrées.

Modifier une entrée

Soient `L` une liste, n sa longueur, et `i` un entier de $[\![-n, n-1]\!]$ et `a` une donnée `Python`.
Alors, la commande
`L[i]=a`
modifie la liste `L` en remplaçant l'élément en `i`-ième position par `a`.
Autrement dit, les entrées `L[i]` de `L` se manipulent comme (en fait, *sont*) des variables, que l'on peut réaffecter.

Exemple 6. Prédire le résultat des commandes `print` ci-dessous, puis vérifier avec l'invite de commande.

```
L=[1,3,2,4]
L[2]=5
print(L)
L[-1]=100
print(L)
L[1]=L[-1]
print(L)
L[0],L[1]=10,12
print(L)
```

Remarque. La commande `L[i]=a` renvoie une erreur si `i` n'est pas un indice pertinent.

b) Ajout d'un élément en fin de liste

Ajout d'un élément en fin de liste

Soit `L` une liste et `a` une donnée. Alors, la commande
`L.append(a)`
modifie la liste `L` en ajoutant l'élément `a` à la fin de celle-ci.

Exemple 7. Les lignes suivantes affichent `[1,2,3]`.

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

c) Suppression d'une entrée

Supprimer une entrée

Soit `L` une liste, et `i` l'indice d'un élément de `L` (pouvant être négatif). Alors, la commande
`del L[i]`
modifie la liste `L` en retirant son élément d'indice `i`.

Exemple 8. A la fin des lignes suivantes :

```
L=[1,3,4,10]
del L[2]
```

la liste `L` est `[1,3,10]`.

Remarque. Les **occurrences** d'un élément dans une liste sont ses éventuelles apparitions.

Exercice 9.

1. Écrire le code d'une fonction d'entête `def Occurrences(L,a):` prenant en entrée une liste `L` et une donnée `a`, et renvoyant en sortie le nombre (éventuellement nul) de fois que `a` est présent dans la liste `L`.
2. Écrire le code d'une fonction d'entête `def ListeOccurrences(L,a):` prenant en entrée une liste `L` et une donnée `a`, et renvoyant en sortie la liste des indices de toutes les occurrences de `a` dans la liste `L`.

   On pourra, dans cette fonction :

   - *initialiser une variable* `G` *comme étant la liste vide à l'aide de* `G=[]` *puis,*
   - *remplir* `G` *par les indices des occurrences de* `a` *dans* `L`.
3. Coder une fonction d'entête `def Supprime(L,a):` prenant en entrée une liste `L` et une donnée `a`, et modifiant `L` en retirant toutes les occurrences de `a`.

   On pourra utiliser la fonction de la question précédente. Remarquez que dans ce cahier des charges, la fonction ne renvoie rien (elle ne fait que modifier son argument).

En fait...

Test d'occurrence, nombre d'occurrences

Soit `L` une liste et `a` une donnée. Alors :

- la commande

  `a in L`

  renvoie `True` si `a` est élément de `L`, et `False` sinon.
- La commande

  `L.count(a)`

  renvoie le nombre d'occurrences de `a` dans la liste `L` (c'est-à-dire, le nombre de fois que `a` est présent dans `L`).

Exemple 10. Que renvoie la fonction `mystere` suivante, prenant en entrée une liste de nombres `L` non vide ?

```
def mystere(L):
        if L[0]**2 in L:
                return(True)
        if L[-1]**2 in L:
                return(True)
        return(False)
```

4. Autres opérations sur les listes

a) Sous-listes

Accès à une sous-liste

Soit `L` une liste, et `i` et `j` des entiers tels que `i` $\leq$ `j`. Alors :

1. La commande `L[i:j]` renvoie la sous liste obtenue en conservant uniquement les entrées dont l'indice est compris entre `i` et `j-1`.
2. La commande `L[:j]` renvoie la sous liste obtenue en conservant uniquement les entrées dont l'indice est inférieur ou égal à `j-1`.
3. La commande `L[i:]` renvoie la sous liste obtenue en conservant uniquement les entrées dont l'indice est supérieur à `i`.

Dans chaque cas, si aucun élément ne vérifie la condition demandée, la commande donnée renvoie la liste vide.

Remarque. On peut utiliser des entiers négatifs, mais il faut se familiariser à quelques subtilités.

Exemple 11. Prédire et vérifier l'exécution des commandes suivantes, dans l'ordre indiqué :

1. `L=[9,7,5,3,1]`
2. `print(L[1:4])`
3. `print(L[2:5])`
4. `print(L[-4:-2])`
5. `print(L[2:])`
6. `print(L[:-2])`
7. `print(L[1:-2])`

b) Copie de liste

La nécessite de la commande suivante est plus subtile. Pour la comprendre, regardez l'exemple suivant.

Exemple 12. Exécutez une par une les lignes suivantes dans l'invite de commande. Que remarquez vous?

```
L=[6,2,3,1,3]
G=L
print(G)
G[0]=0
print(G)
print(L)
```

Remarque. (*Spoil* l'exemple précédent) Dans l'exemple précédent, on remarque que si on effectue la commande `G=L`, on définit une liste `G` étant égale à `L`, mais avec une subtilité supplémentaire : les deux listes sont liées (en fait, `G` et `L` désignent la même case mémoire de l'ordinateur). Autrement dit, toute modification de l'une affectera l'autre.

Des fois, on veut copier une liste en une liste indépendante.

Copie de liste

Soit `L` une liste. Alors, la commande
`M=L[:]`
définit une nouvelle liste `M` identique à `L` qui lui est indépendante.
Cette opération est aussi donnée par la commande :
`M=L.copy()`

Exemple 13. Reprendre l'exemple précédent en remplaçant `"G=L"` par `"G=L[:]"`.

c) Concaténation de listes

La concaténation de deux listes est la liste obtenue en mettant ces deux listes bout-à-bout.

Concaténation

1. Soient `L` et `M` deux listes. Alors, la commande
   `L+M`
   renvoie la liste obtenue en concaténant `L` et `M`.
2. Soit `L` une liste et `n` un entier positif. Alors, la commande
   `L*n`
   renvoie la liste obtenue en concaténant `L` `n` fois avec elle-même.

Exemple 14. Prédire et vérifier le résultat renvoyé par `[3,2,1]+[4]`, puis par `[1,3]*4`.

5. Listes et boucles `for`, définition d'une liste en compréhension

Pour définir une petite liste, on peut tout simplement la rentrer à la main, en écrivant par exemple
`L=[2,3,5,7]`
mais pour des listes plus compliquées, il nous faut d'autres stratégies.

Une première manière de définir une liste `L` à l'aide d'un code plus complexe est de :

- Initialiser `L` comme une liste vide, avec
  `L=[]`
  puis,
- Remplir `L` avec les éléments `a` voulus à l'aide de la commande `L.append(a)`.

Avec cette approche, on espère remplir `L` en ajoutant ses éléments petit à petit, et il est alors courant d'utiliser une boucle `for`.

Exemple 15. Anticiper les réponses aux questions, puis vérifier.

```
G=[]
for i in range(101):
        G.append(i**2)
#Que contient G à ce stade?
L=[]
for i in range(len(G)):
        if G[i]%2==0:
                L.append(G[i])
#Que contient L ?
```

Exercice 16. Écrire une fonction `retourne` prenant en entrée une liste `L` et renvoyant en sortie la liste formée des éléments de `L` mais dans l'ordre inverse à celui de `L`.

a) Parcours des éléments d'une liste

Listes et boucles for

Soit `L` une liste. Alors, la commande
`for i in L:`
initialise une boucle `for` dans laquelle la variable `i` prendra comme valeurs toutes les entrées de `L`, dans leur ordre d'apparition.

Exemple 17. L'exemple précédent peut se réécrire ainsi :

```
G=[]
for i in range(101):
        G.append(i**2)
#Jusqu'ici, rien ne change.
L=[]
for a in G:
```

```
if a%2==0:
        L.append(a)
```

b) Définition d'une liste en compréhension

Pour appliquer la stratégie de remplissage d'une liste avec une boucle for comme ci-dessus, on dispose d'une syntaxe plus rapide.

Définition d'une liste en compréhension

Soit `G` une liste. Alors, la commande
`L=[expression(x) for x in G if condition(x)]`
définit une liste `L` selon les modalités suivantes :

- `"expression(x)"` doit renvoyer une donnée pour chaque valeur possible de `x` dans `G`.
- `"condition(x)"` doit renvoyer un booléen pour chaque valeur possible de `x` dans `G`.
- Alors, `L` est la liste des éléments de la forme `expression(x)` pour chaque entrée `x` de `G` vérifiant la condition `condition(x)`.

Remarque. On peut utiliser cette syntaxe pour n'importe quelle type de boucle `for`.

Remarque. On peut utiliser cette syntaxe sans `"if condition(x)"`.

Exemple 18.

- Après la commande :
  `L = [ x for x in range(10,21)]`
  la liste `L` contient les entiers de 10 à 20, dans l'ordre croissant (`L` est la liste `[10,11,12,...,20]`).
- `L` étant la liste précédente, `[-x for x in L]` renvoie la liste des opposés des éléments de `L`, donc la liste `[-10,-11,-12,...,-20]`.
- Après la commande :
  `M=[2*x for x in range(100) if x%2==0]`
  la liste `M` contient les doubles des entiers pairs de 0 à 99 (donc `L` est la liste `[0,4,8,12...,196]`).

Exemple 19. Anticiper l'affichage des commandes `print`, puis vérifier.

```
G=[6,2,3,4,1,3]
L=[x**2 for x in G if x%2==1]
print(L)

M=[2*x+1 for x in range(10)]
print(M)

N=[i for i in range(100) if 3*i>34]
print(N)
P=[n/2 for n in N if n%2==0]
print(P)
```

Exercice 20. Reprendre l'exercice 14 à l'aide d'une définition de liste en compréhension.

II. Exercices

1. Pour commencer

Exercice 21. Écrire une fonction `parite` prenant en entrée une liste d'entiers `L` et renvoyant en sortie la liste obtenue à partir de `L` en mettant un `0` pour chaque entrée paire, et un `1` pour chaque entrée impaire.

Par exemple, `parite([2,3,1,3,4]` devra renvoyer la liste `[0,1,1,1,0]`.

Exercice 22. Écrire une fonction `testConstante` prenant en entrée une liste `L`, et renvoyant `True` si `L` est vide ou si toutes les entrées de `L` sont identiques, et `False` sinon.

Exercice 23. Écrire une fonction `maxList` prenant en entrée une liste non vide de nombres et renvoyant en sortie la valeur du plus grand élément de la liste.

Exercice 24. Écrire une fonction `maxListPos` prenant en entrée une liste non vide de nombres, et renvoyant en sortie un couple `a,i` où a est la valeur du maximum de la liste, et i la position de la première occurrence de a.

Exercice 25. Écrire une fonction `maxListOcc` prenant en entrée une liste non vide de nombres, et renvoyant en sortie le couple `a,k` où a est la valeur du maximum de la liste, et k le nombre d'occurrences de cette valeur maximale.

Exercice 26. Écrire une fonction `max2` prenant en entrée une liste de nombres de longueur au moins 2, et renvoyant en sortie le couple `a,b` des deux plus grands nombres de cette liste dans l'ordre décroissant.

2. Exercices classiques sur les suites

Exercice 27. Soit u la suite donnée par $u_0 = 1$ et $\forall n \in \mathbb{N}, u_{n+1} = u_n^2 + 2n + 1$. Écrire une fonction d'entête `def U(n):` prenant en entrée un entier naturel `n` et renvoyant en sortie la liste $[u_0, u_1, \ldots, u_{\texttt{n}}]$.

Exercice 28. Soit v la suite donnée par $v_0 = 1, v_1 = 2$ et $\forall n \in \mathbb{N}, v_{n+2} = 2v_n^2 + 3v_{n+1}$. Écrire une fonction d'entête `def V(n):` prenant en entrée un entier naturel `n` et renvoyant en sortie la liste $[v_0, v_1, \ldots, v_{\texttt{n}}]$.

3. Un peu de nombres premiers

Exercice 29. Soit n et d deux entiers naturels. On dit que d est un *diviseur* de n s'il existe un entier k tel que $n = kd$. On dit qu'un entier n est *premier* s'il possède exactement deux diviseurs distincts: 1 et n. En particulier, 1 n'est pas premier (il n'a qu'un diviseur : 1).

1. Écrire une fonction `est_premier` prenant en entrée un entier naturel non nul et renvoyant `True` si ce nombre est premier, et `False` sinon. Tester votre fonction pour une dizaine de valeurs (espacées) de l'argument.
2. Écrire une fonction `liste_premiers` prenant en entrée un paramètre $n > 0$ entier et renvoyant la liste des nombres premiers inférieurs ou égaux à n, sous la forme d'un objet de type `list`.
3. En déduire une fonction `compte_premiers` renvoyant le nombre de nombres premiers inférieurs à un entier donné en entrée.

4. Un exercice de tri

Exercice 30. Voici un premier exercice de tri, problématique classique sur les listes. Vérifiez bien vos réponses en les testant sur quelques listes entrées à la main.

Dans cet exercice, on ne modifiera pas la liste à trier.

1. Écrire une fonction `insert` prenant en entrée un triplet `M,i,e` et renvoyant la liste obtenue en insérant, à la liste `M`, l'élément `e` et i-ième position. Par exemple, `insert([2,3,4],1,9)` doit renvoyer `[2,9,3,4]`.
2. Écrire une fonction `rang` prenant en entrée `M,e` où `M` est une liste de nombres *supposée triée dans l'ordre croissant*, et renvoyant la position du premier élément de `M` supérieur à `e`. Vous pourrez utilisez la commande `break` qui, dans une boucle, interrompt la boucle la plus récente en cours.
3. En déduire une fonction `tri1` prenant en entrée une liste et renvoyant la liste obtenue en triant cette liste dans l'ordre croissant. On s'aidera de la remarque ci-dessous.

Il existe de nombreuses méthodes de tri. Cet exercice implémente le tri appelé : tri par insertion. L'idée est la suivante.

- *On initialise une liste* `M` *vide, qui contiendra la version triée de* `L`.
- *On parcourt la liste* `L` *pour ajouter un à un les éléments de* `L` *à a la liste* `M`.
- *On utilise* `rang` *pour savoir où mettre, dans* `M`*, les éléments de* `L` *et* `insert` *pour les mettre au bon endroit.*

On verra au prochain TP une autre méthode de tri : le tri à bulles, que l'on retiendra (il est bien plus facile à coder).