TD 05 Structure de données : les listes

Lycée Louis Thuillier - Informatique - PCSIB -

Nous avons étudié une première structure de données sous Python : les chaines de caractères. Ces structures sont utiles pour stocker du texte, leur propriété étant similaire à celle d'un fichier PDF : elles sont de longueurs fixes et non-modifiable.
On introduit dans ce TD un nouveau type de structures de données qui permet de stocker et manipuler "*n'importe quoi*".

Attention ! Les listes et les chaines de caractères sont deux types de structures de données en Python : il existe donc de nombreuses similitudes. Tout ce qu'on a appris sur les chaines est transposable "*quasi tel quel* aux lises mais on fera bien attention **aux différences entre les deux objets ! ! ! !**

Comme on le verra plus tard, on retrouvera les listes (ou des structures proches de ces dernières) **PARTOUT** ! Il est donc très important de bien maitriser ce TD.

1 Définition et structure

1.1 Définir une liste

Une liste est une séquence d'objets :
▷ séparés par des virgules
▷ placés entre crochets

$$[objet\_1\ ,\ objet\_2\ ,\ objet\_3]$$

Important : Les objets d'une liste ne sont pas nécessairement du même type.

On peut faire des listes de listes ou des listes vides.

```
liste_1=[2 , 2.3 , "abcde"]

listception = [ [0,2,4,6,8],[1,3,5,7,9] ]

liste_vide=[]
```

Application 1 :
▷ *Créer une liste miam_miam contenant vos deux plats préférés.*
▷ *Créer une liste nb_premier contenant les 3 premiers nombres premiers.*

1.2 Accéder à un élément d'une liste

Chaque élément d'une liste possède un indice, *i.e.* un nombre entier i reflétant sa position dans la liste.
Attention ! Le premier élément possède l'indice 0 ! !

On accède à un élément d'indice i d'une liste L de la même façon qu'on accède à un élément d'une chaîne.

♡ **Commande ♡. Accéder à un élément d'une liste**
On accède au $n^{\text{ième}}$ caractère d'une liste à l'aide de la commande $[n]$ apposé juste après le nome de la liste.

$$nom\_de\_la\_liste[n]$$

Attention ! On commence à compter à partir du 0 ! ! Du zéro ! ZERO ! ! 0000000000000

Tout comme pour les chaines de caractères, la commande *len*() permet d'accéder aux nombres d'élément d'une liste.

♡ **Commande** ♡. **Nombre d'éléments**
La commande *len*(), avec entre parenthèse le nom de la liste, permet de renvoyer la longueur de la liste.

Application 2 : La variable L_1 est une liste inconnue. Donner les commandes pour afficher :
- ▷ *le nombre d'éléments de L_1*
- ▷ *le premier élément de L_1*
- ▷ *le dernier élément de L_1*
- ▷ *l'avant dernier élément de L_1*

> ***Remarque :*** *La commande $L[-1]$ permet d'accéder au dernier élément de L, $L[-2]$ à l'avant dernier, etc ...*

1.3 Ajouter et modifier un élément d'une liste

Attention ! C'est la différence fondamentale avec une chaine de caractères : **une liste est modifiable**, c'est-à-dire que l'on peut
- ▷ modifier un élément au sein d'une liste
- ▷ ajouter un élément à une liste

▶ Modifier un élément d'une liste

♡ **Commande** ♡. **Modifier un élément d'une liste**

On peut réaffecter sa valeur à l'aide d'une affectation :

$$L[i] = y$$

remplace l'élément d'indice i de L par la nouvelle valeur y.

```
>>> L=[1,2,4,4]

>>> L[2]=3

>>> L
[1, 2, 3, 4]
```

Accéder à un bout d'une liste :

Tout comme une chaîne, on peut accéder à un morceau d'une liste L via la commande $L[i : j]$ qui représente une liste composée de tous les éléments de L d'indice compris entre i et $j - 1$.

On peut sélectionner un pas k grâce à la commande $L[i : j : k]$.

```
>>> L=[1,2,3,4,5,6,7,8,9,10]

>>> L[2:5]
[3, 4, 5]

>>> L[1:9:2]
[2, 4, 6, 8]
```

▶ Ajouter un élément à une liste

Contrairement à une chaîne, on peut ajouter des éléments à une liste. Cela se réalise grâce à la commande *append*().

♡ **Commande** ♡. **Ajouter un élément :** *append*

Pour ajouter un élément x à une liste L, on utilise la commande : $L.append(x)$

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

>>> L.append(4)

>>> L
[1, 2, 3, 4]
```

Application 3 :
▷ *Ajouter à miam_miam votre troisième plat préféré.*
▷ *Ajouter à nb_premier le quatrième nombre premier.*

1.4 Construire une liste

Pour construire une liste, on va généralement partir d'une liste vide qu'on va remplir pas-à-pas.

Structure : Construire une liste

```
L=[]
for k in range(....):
    L.append(....)
```

Application 4 :
1. *Construire une liste contenant tous les entiers multiples de 7 compris entre 0 et 1000.*
2. *Remplacer par 0 tous les nombres de la liste précédente dont la position dans la liste est un multiple de 9 .*
3. *Remplacer chaque élément de la liste par la valeur de l'élément suivant. On considèrera que le premier élément est l'élément suivant le dernier.*
   Attention ! *Il y a un piège à la fin !*

1.5 (Pour aller plus loin) Notion de méthode :

La commande *append* est une **méthode** associée à l'objet "liste".

Les méthodes sont des fonctions (sous Python) associées à des objets particuliers (listes, chaînes, Booléens, nombre entier, ...) qui permettent de les manipuler de façon simple et pratique.
La syntaxe d'utilisation d'une méthode est toujours la suivante : le nom de la donnée, suivi d'un point, suivi du nom de la méthode avec entre parenthèses ses paramètres.
Exemple : $L.append(x)$ ajoute à la liste L l'élément x.

Attention ! Une méthode **agit** directement sur l'objet : en l'utilisant, on modifie l'objet.

Exemple 1 : On veut créer la liste des sept nains mais on a oublié Grincheux :

```
>>> sept_nains=["Prof" , "Joyeux" ,"Dormeur" , " Simplet" , "Atchoum" , "Timide"]

>>> sept_nains.append("Grincheux")

>>> sept_nains
['Prof', 'Joyeux', 'Dormeur', ' Simplet', 'Atchoum', 'Timide', 'Grincheux']
```

En ajoutant Grincheux on a modifier la liste sept_nains.
Si on n'aime pas Prof, on peut l'enlever grâce à la méthode remove().

```
>>> sept_nains.remove("Prof")

>>> sept_nains
['Joyeux', 'Dormeur', ' Simplet', 'Atchoum', 'Timide', 'Grincheux']
```

Il existe de nombreuses méthodes associées aux objets listes :

Méthode	Résultat
liste.append(x)	Ajoute l'élément x à la fin de la liste
liste.index(x)	Renvoie l'indice de la première occurrence de l'élément x la liste. Message d'erreur si aucune
liste.count(x)	Renvoie le nombre d'occurrence de x dans *liste* .
liste.remove(x)	Retire la première occurrence de x dans la *liste*. Si x n'appartient pas à la liste, renvoie un message d'erreur .
liste.insert(i,x)	Insère l'élément x à l'indice i. Si cet indice dépasse la taille de la liste, l'élément est ajouté à la fin de la liste.
liste.sort()	Ordonne les éléments de la liste par ordre croissant
liste.pop(i)	Retire et renvoie l'élément en position i dans la liste. Si aucun indice est indiqué, la méthode retire et renvoie l'élément en fin de liste.
liste.extend(liste2)	Ajoute les éléments de la liste 2 à la suite des éléments de la liste
liste.reverse()	Renverse l'ordre des éléments de la liste
liste.clear()	Supprime tous les éléments de la liste

Il n'est évidemment pas nécessaire de connaître toutes ces méthodes. Dans un sujet, elles seront rappelées en annexe sous la forme d'un tableau comme celui ci-dessus.

2 Manipulation de liste

2.1 Fusion et concaténation

Concaténer deux listes L_1 et L_2 revient à créer une nouvelle liste en plaçant la deuxième à la suite de la première.
Cela se réalise à l'aide de l'opérateur $+$:

$$L\_1 + L\_2$$

La commande $*$ permet de concaténer plusieurs fois une liste avec elle même.

```
>>> L_1=[1,2,3]

>>> L_2=[4,5,6]

>>> L_1+L_2
[1, 2, 3, 4, 5, 6]
```

2.2 Construction par compréhension (pour aller plus loin)

Un des défaut du langage Python est qu'il prend beaucoup de place à écrire (beaucoup de saut de ligne, d'indentation, ...). Une façon compacte de définir une liste est :

```
>>> L=[2*i for i in range(0,11)]

>>> L
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

> *Exemple 2 : On peut ajouter une condition à la construction :*
>
> ```
> >>> L=[i**2 for i in range(0,100) if i%2==0]
> ```

2.3 Opération sur les listes

▶ Test d'appartenance

♡ Commande ♡. Test d'appartenance

Pour tester si un élément x est dans une liste L, on utilise l'opérateur *in*

$$x \text{ in } L$$

```
>>> L=[1,2,3,4,5,6,7,8,9,10]

>>> 2 in L
True
```

Ce test d'appartenance est un Booléen. Il vaut :
▷ *True* si l'élément x est dans la chaîne
▷ *False* sinon

▶ Se promener dans une liste

Il existe deux façons d'accéder successivement aux différents éléments d'une liste :

Structure :

▷ *via les indices*

```
1 for k in range(len(L)) :
2     ...L[k]...
```

▷ *via les éléments directement*

```
1 for elt in L :
2     ...elt...
```

Application 5 :
▷ *Ecrire une fonction moyenne qui, recevant en entrée une liste de nombre, renvoie la somme de ses éléments*
▷ *Ecrire une fonction moyenne_locale qui, recevant en entrée une liste de nombre L et un entier j, renvoie la moyenne des éléments de Lde position $j-1$, j et $j+1$.*
▷ *Ecrire une fonction somme_miroir qui, recevant en entrée une liste L, renvoie une liste où :*
  ▷ *le premier élément est la somme du premier et du dernier élément de L*
  ▷ *le second est la somme du deuxième et de l'avant dernier élément de L*
  ▷ *...*

2.4 Recherche d'un maximum

Un problème récurent en informatique est la recherche d'un maximum (ou d'un minimum) dans une liste donnée.

▶ Méthode de recherche :

Une façon "*intuitive*" de rechercher un maximum est de parcourir une liste en sauvegardant à chaque étape l'élément le plus grand déjà rencontré.

Application 6 :
1. *Compléter l'algorithme (i.e ; les étapes par lesquelles passer) pour rechercher le maximum dans une liste L. (Il est évidemment interdit d'utiliser la fonction max).*

   max = premier élément de L
   Pour k allant de ..... à ..... :
   Si max ..... :
   max = ...

2. *Écrire un fonction maximum qui, prenant en entrée un liste L, renvoie l'élément le plus grand de L.*
3. *Écrire un fonction second_maximum qui, prenant en entrée un liste L, renvoie le deuxième élément le plus grand de L.*

Notion de complexité et problèmes engendrés

> *Exemple 3 : En utilisant le programme de l'exemple 1, appliquer la fonction maximum à une liste comprenant tous les entiers entre :*
> ▷ 0 *et* 1000
> ▷ 0 *et* 10^6
> ▷ 0 *et* 10^7
> *Quels problèmes rencontre-t-on alors ?*

Définition. On appelle **temps de calcul** la durée nécessaire à l'exécution d'un programme.

Le temps de calcul est un paramètre important d'un programme qu'on cherchera toujours à minimiser pour des raisons pratiques évidentes.

Pour estimer le temps de de calcul que va nécessiter la recherche du maximum dans l'exemple précédent, on peut chercher à répondre à cette question :

En fonction de la taille n de la liste L en entrée, combien d'opérations élémentaires mon programme va-t-il effectuer dans le pire des cas ?

Nous verrons plus tard dans l'année comment calculer et analyser la complexité, et donc le temps de calcul, d'un programme.

▶ Application

> *Application 7 :*
>
> 1. *Décrire ce que fait la fonction ci-contre*
> 2. *Pourquoi a piorir cette fonction risque de prendre beaucoup plus de temps à fonctionner que la fonction maximum précédente ?*

```
def mystere(L):
    result=0
    for k in range(len(L)):
        for i in range(len(L)):
            if abs(L[k]-L[i])>result:
                result = abs(L[k]-L[i])
    return result
```

3 Exercices d'application

3.1 Petites fonctions pratiques

1. Écrire une fonction *trouver_premier* qui, recevant un nombre 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 $False$.
2. Modifier la fonction précédente pour afficher la position de la deuxième occurrence de x.
3. Écrire une fonction *nombre_occurence* qui, recevant un nombre x et une liste L, renvoie le nombre de fois que x apparaît dans L.
4. Écrire une fonction *echanger* qui, recevant une liste L et deux entiers i et j, renvoie une liste similaire à L mais où les éléments d'indices i et j de L ont été intervertis.

3.2 Arithmétique

1. Ecrire une fonction $chiffres$ qui prend en entrée un nombre x compris entre 1 et 10^9 et qui renvoie une liste constitué de ses chiffres.
   Exemple : $chiffres(153)$ doit renvoyer $[1, 5, 3]$.
   Astuce : la division euclidienne par 10 est votre meilleur ami.
2. Afficher le dernier chiffre des multiples de 5 compris entre 5 et 100.

3. Afficher la somme des chiffres des multiples de 3 compris entre 3 et 100.
4. Trouver tous les entiers naturels égaux au cube de la somme de leurs chiffres.
   Exemple : 153 ne marche pas car $153 \neq (1+5+3)^3$
5. Trouver tous les entiers naturels égaux à la somme des cubes de leurs chiffres.
   Exemple : 153 marche car $153 = 1^3 + 5^3 + 3^3$

3.3 Un peu de ménage dans les listes

Consigne : *pour ces questions, on n'utilisera aucune méthode Python autre que append.*

1. Écrire une fonction *supprimer_doublons* qui, recevant une liste L, renvoie une liste des éléments de L mais où chaque élément ne se trouve qu'en un seul exemplaire.
2. Écrire une fonction *minimum* qui, recevant une liste de nombres en entrée, renvoie la valeur du minimum de la liste.
3. Écrire une fonction *supprimer_element* qui, recevant en entrée une liste L de nombres et un nombre x, renvoie la liste des éléments que L mais sans ceux égaux à x.
4. A partir de deux fonctions précédentes, écrire une fonction *enlever_minimum* qui, recevant en entrée une liste L, renvoie la même liste L mais sans son minimum.