Informatique - Notes de cours 6

Listes 2 : parcours par indices, modifications et suppressions d'éléments

M. Marmorat

11 décembre 2024

1 Indices des éléments d'une liste

1.1 Numérotation des éléments

Rappel : on peut accéder au nombre d'éléments d'une liste `L` via l'instruction
Ainsi si `L=[2,4,5]` alors

Soit `L` une liste quelconque ; notons n = `len(L)`. Alors les éléments de la liste `L` sont numérotés de 0 à $n-1$. De plus on accède au `k`-ème élément de `L` via la syntaxe `L[k]`. Autrement dit on a :

```
L = [L[0], L[1], L[2], ..., L[n-1]]
```

Exercice 1. Que contient `x` après exécution des scripts suivants ?

```
L = [6,-5,4,2,0]
x = L[3]
```

```
L = [[3,5,2],[2,6],[1,3,0]]
y = L[1]
x = y[1]
```

Attention, si on dépasse la longueur de la liste, Python renvoie l'erreur `list index out of range`. C'est par exemple le cas avec `L=[1,0,6]` si on demande `L[ ]`.

On peut en renvanche utiliser un indice négatif. Python compte alors les éléments de la liste depuis la fin. Ainsi `L[-1]` renvoie le dernier élément de `L`, `L[-2]` l'avant dernier, etc.

1.2 Parcours des éléments par indices

Pour parcourir les éléments d'une liste par leurs indices, on utilise la syntaxe :

```
for k in range(len(L)) :
    # instructions exploitant (ou non) l'indice k et l'element L[k] de
       la liste
```

C'est naturel, puisque l'entier `k` prend alors les valeurs :

Exercice 2. Écrire une fonction `places1` prenant en argument une liste de nombres `L` et renvoyant une liste `Lbis` indiquant à quelles positions se trouvent tous les 1 que contient `L`. Autrement dit, `Lbis` contiendra l'entier k si et seulement si `L[k]` vaut 1. Par exemple, `places1([1,3,1,1,5,1])` doit renvoyer

Exercice 3. Écrire une fonction prenant en argument une liste `L` dont les éléments sont des nombres $x_0, x_1, \ldots, x_{n-1}$, et renvoyant la quantité : $\sum_{k=0}^{n-1} x_k\, 2^k$.

Attention à ne pas confondre les syntaxes pour les deux parcours :

parcours par éléments : parcours par indices :

Remarque 1.
- En particulier, les syntaxes : `for k in len(L)` ou `for x in range(L)` n'ont pas de sens.
- Utilisez des noms de variables indiquant si vous faites un parcours par éléments ou par indices : il est naturel d'appeler `k` *l'indice* permettant de parcourir la liste (c'est un entier) et `x` *l'élément* de la liste (qui peut être de n'importe quel type).

Comment choisir entre un parcours par indices ou par éléments ?
Tout ce qui est fait avec un parcours par éléments peut être fait avec un parcours par indices. Mais il est maladroit d'utiliser un parcours par indices lorsque ce n'est pas nécessaire. Pour choisir entre les 2 parcours, il faut se demander : ai-je besoin de savoir à quel indice de la liste se situe l'élément en question ?

Exercice 4. Dans chacun des cas suivants, décider s'il faudra utiliser un parcours par indices ou par éléments :

1. Écrire une fonction prenant en argument une liste `L` contenant les nombres $x_1, x_2, \ldots, x_n$ et renvoyant :

   (a) $\displaystyle\sum_{k=1}^{n} \frac{x_k}{k^2+1}$ (b) $\displaystyle\prod_{k=1}^{n}(1+3x_k)$ (c) $\displaystyle\sum_{k=1}^{n} x_k x_{n+1-k}$

2. Écrire une fonction prenant en argument une liste `L` de nombres et renvoyant la liste :
   (a) contenant tous les éléments de `L` qui sont pairs
   (b) contenant tous les éléments de `L` qui sont à une position paire

1.3 Sous-listes (slicing)

Il est possible d'accéder aux *sous-listes* d'une liste `L`, c'est-à-dire aux listes composées d'éléments consécutifs de `L`. Pour `p` et `q` deux entiers compris entre `0` et `len(L)` inclus, la syntaxe `L[p:q]` permet d'obtenir la liste des éléments de `L` compris entre les indices `p` (inclus) et `q` (exclu). En d'autres termes :

```
L[p:q] = [ L[p] , L[p+1] , L[p+2] , ... , L[q-1] ]
```

Si l'indice de départ (`p`) vaut 0, autrement dit si on considère une sous-liste qui est un *préfixe* de `L`, alors cet indice peut être omis.
De même si l'indice d'arrivée (`q`) vaut `len(L)`, autrement dit si on considère une sous-liste qui est un *suffixe* de `L`, alors cet indice peut être omis.
Lorsque les deux indices sont omis, c'est qu'on considère la sous-liste de tous les éléments de `L`.
En un mot :

```
L[:q] = L[0:q],  L[p:] = L[p:len(L)],  L[:] = L[0:len(L)] = L
```

Exercice 5. Que vaut `M` dans chacun des cas suivants ?

```
L = [4,7,-1,0,3,2]
M = L[2:5]
```

```
L = [4,7,-1,0,3,2]
M = L[:3]
```

```
L = [4,7,-1,0,3,2]
M = L[4:]
```

```
L = [4,7,-1,0,3,2]
L_bis = L[2:]
M = L_bis[:2]
```

Plus rarement, on peut avoir besoin de faire des sous-listes contenant des éléments non consécutifs dans la liste de départ. Par exemple, on peut vouloir extraire uniquement les éléments d'incides pairs. Pour cela, la syntaxe `M = L[p:q:r]` permet d'obtenir la sous-liste qu'on pourrait aussi écrire de la manière suivante :

```
M = [L[k] for k in range(p,q,r)]
```

En d'autres termes `L[p:q:r]` contient

Exercice 6. Comment accéder à la liste des éléments d'indices pairs de `L` ?

Exercice 7. On décrit un brin d'ADN par une liste dont les éléments sont toujours un des quatre caractères `"A"`, `"T"`, `"C"` et `"G"`. Le début d'un message génétique est généralement signalé par le codon `"A","T","G"`. Écrire une fonction `depart` qui prend en argument un brin d'ADN et renvoie l'indice du début du message génétique.
Par exemple, `depart(["C","G","A","A","T","C","A","T","G","G","T","A"])` doit renvoyer `6`.
La fonction renverra un message d'erreur si le codon de départ n'est pas présent dans le message génétique considéré.

2 Modifications, suppressions

Les listes ne sont pas les seules structures en Python permettant de stocker plusieurs variables. On peut par exemple stocker n variables $x_1, \ldots, x_n$ dans un n-uplet (ou *tuple* en Python) :

$$\texttt{x} = (x_1, x_2, \ldots, x_n)$$

Notez la différence avec les listes :

Comme pour les listes, on accède aux éléments d'un tuple grâce à des crochets, avec une numérotation commençant à 0. Par exemple, si `x = (3,5,1,2)` alors `x[1]` vaut , `x[3]` vaut , `x[0]` vaut et `x[4]`

L'avantage des listes réside dans le fait qu'il s'agit de variables *mutables* : on peut modifier leur contenu, supprimer certains de leurs éléments et en ajouter de nouveaux (comme nous l'avons déjà fait avec la commande `append`).

2.1 Modification d'un élément d'une liste

On peut modifier le `k`-ème élément d'une liste `L`, pour lui donner la valeur `val` grâce à la commande suivante :

```
L[k] = val
```

Exercice 8. Que vaut `L` après exécution des scripts suivants ?

```
L = [4,5,2,7]
L[1] = 0
L[3] = 1
```

```
L = [4,5,2,7]
L[-2] = 0
L.append(6)
L[-2] = 1
```

Remarque 2. Ces opérations ne sont pas possibles pour les tuples : si `x=(4,5,2,7)` alors `x[1] = 0` renvoie un message d'erreur ! Les tuples, (tout comme les constantes) ne sont pas *mutables*.

Exercice 9. Écrire une fonction `transforme` qui prend en argument une liste d'entiers et la renvoie après avoir remplacé tous les `4` qu'elle contient par des `5`.

Remarque : Tout parcours d'une liste qui vise à en modifier les éléments se fait par
En effet, pour modifier un élément d'une liste, on utilise la syntaxe

2.2 Suppression d'un élément

On peut aussi supprimer un élément d'une liste. Pour cela, la commande `L.pop(k)` permet de supprimer et renvoyer le `k`-ème élément de `L`. Ainsi en écrivant :

- `L.pop(k)` on supprime l'élément `L[k]` de la liste
- `x = L.pop(k)`, on supprime l'élément `L[k]` de la liste et on le stocke dans la variable `x`

Exercice 10. Que valent `L` et `x` après exécution des scripts suivants ?

```
L = [4,5,6,3]
x = L.pop(2)
```

```
L = [4,5,6,3]
x = L.pop(0)
L.append(x)
```

Remarque 3. Lorsqu'on l'utilise sans argument, la commande `pop` agit sur le dernier élément de la liste. Ainsi `x = L.pop()` supprime le dernier élément de `L` et le place dans la variable `x`.

Remarque 4. Attention à ne pas confondre les syntaxes `L.append(x)` et `x = L.pop()`. Lorsqu'on veut rajouter un élément `x` à une liste, il est naturel de passer `x` en *argument* de la commande `append`. En revanche, lorsqu'on veut *récupérer* le dernier élément d'une liste, qui nous est inconnu, il est naturel de *définir* `x = L.pop()`.

Exercice 11. Écrire une fonction `miroir` prenant en argument une liste `L` et renvoyant la liste `L` inversée. Par exemple `miroir([1,2,3])` renverra `[3,2,1]`. On écrira deux versions de la fonction :

- `miroir1` qui videra la liste `L` pour créer sa liste inversée, et
- `miroir2` qui n'aura pas d'effet sur `L` et renverra une nouvelle liste.

3 Remarques spéciales pour les listes

3.1 Effets de bords

Dans l'exercice précédent, on a distingué deux façons de renvoyer le mirroir d'une liste : une première fonction *modifiant* la liste, et une autre renvoyant *une autre* liste.

Lorsqu'on manipule des listes, il est en fait courant de modifier une liste prise en argument par une fonction (contrairement à ce à quoi on est habitué !). C'est le cas dès qu'on utilise sur cette liste les instructions :

Dans ce cas, on parle d'*effet de bord*, à comprendre au sens de *side effect*, c'est-à-dire effet secondaire. L'exécution de la fonction impacte la variable prise en argument de la fonction. Sauf indication contraire, on privilégiera les fonctions n'ayant pas d'effet de bords.

Exercice 12. 1. Que fait la fonction suivante ?

```
def double(L):
    for x in L:
        L.append(x)
    return L
```

2. Dans la console, un utilisateur entre les commandes suivantes :

```
L = [7,2]
double(L)
```

Que renvoie Python ?

3. Ayant été distrait par un camarade, l'utilisateur tape à nouveau dans la console `double(L)`. Que renvoie Python ?

4. Proposer une autre façon de définir la fonction `double` n'ayant pas ce défaut.

3.2 Copie d'une liste

Pour résoudre le problème des effets de bords, on peut être tenté de copier la liste `L` prise en argument dans une liste `M` puis de travailler sur `M`. Cette solution est possible à condition de savoir comment copier une liste en Python, ce qui peut être un exercice risqué.

En effet, si `L` est une liste, alors l'instruction `M = L` ne crée pas une variable indépendante de `L`, mais donne simplement un deuxième nom à la liste `L` (contrairement à ce à quoi on est habitué !). Ainsi si on exécute par exemple les scripts suivants :

```
L1 = [4,7,1]
M1 = L1
L1[0] = 2
```

```
L2 = [4,7,1]
M2 = L2
M2.pop(1)
```

alors `L1` et `M1` valent
et `L2` et `M2` valent

Pour résoudre ce problème, il faut forcer Python à copier les éléments de `L` dans une nouvelle liste `M` en utilisant une des syntaxes suivantes :

```
M = L[:]
```

ou

```
M = [x for x in L]
```

4 Entraînement

Exercice 13. 1. Écrire une fonction prenant en argument une liste de nombres réels `L` = $[x_0, x_1, \ldots, x_{n-1}]$ et renvoyant la quantité $\prod_{k=0}^{n-1}(k + x_k)$.

2. Écrire une fonction prenant en argument une liste de nombres réels `L` = $[x_1, x_2, \ldots, x_n]$ et renvoyant la quantité $\sum_{k=1}^{n} \frac{x_k}{k}$.
3. Écrire une fonction prenant en argument une liste de nombres réels `L` = $[x_1, x_2, \ldots, x_n]$ et renvoyant la quantité $\prod_{k=1}^{n} \frac{x_k}{x_k^2+1}$.
4. Écrire une fonction prenant en argument une liste de nombres réels `L` = $[x_0, x_1, \ldots, x_{n-1}]$ et renvoyant la quantité $\sum_{k=0}^{n-1} x_k x_{n-1-k}$.

Exercice 14. Écrire des fonctions Python prenant en argument une liste de nombres réels `L` et renvoyant :

1. le nombre d'éléments pairs de `L`. Par exemple pour `L = [8,1,3,2,2,1,-4]` votre fonction doit renvoyer 4 (puisque les éléments pairs de `L` sont $8, 2, 2$ et -4).
2. le nombre d'éléments pairs de `L` placés à une position paire.
   Par exemple pour `L = [8,1,3,2,2,1,-4]` votre fonction doit renvoyer 3 (puisque $8, 2, 2$ et -4 sont placés aux positions $0, 3, 4$ et 6 ; ainsi seuls le 8, le deuxième 2 et -4 sont comptés).

Exercice 15. Écrire une fonction `recherche` prenant en argument une liste `L` et un élément de type quelconque `a` et renvoyant l'indice de la première occurence de `a` dans la liste `L`. La fonction renverra `'introuvable'` si `a` n'est pas un élément de `L`.

Exercice 16. Écrire une fonction `indmin` prenant en argument une liste de nombres réels et renvoyant *l'indice* du minimum de la liste (on renverra le plus petit indice si le minimum n'est pas unique).

Exercice 17. Écrire une fonction `suffixes` qui prend en argument une liste et renvoie la liste de tous ses suffixes. Par exemple, `suffixes([1,2,3])` doit renvoyer `[[1,2,3],[2,3],[3],[]]`.

Exercice 18. 1. Écrire une fonction `sous_mot` prenant en arguments une liste `texte` et une liste `mot` et renvoyant `True` si `mot` est une sous-liste[1] de `texte`, et `False` sinon.

2. Si la liste `texte` est de longueur n et la liste `mot` de longueur m, combien d'opérations "élémentaires" (i.e. ici de tests d'égalités entre deux éléments des listes) réalise la fonction `sous_mot` dans le pire des cas ?

1. On entend par "sous-liste" que `L2` est toujours une sous-liste de `L1+L2+L3`, par exemple `[3,6,2]` est une sous-liste de `[5,9,3,6,2,0]`. On trouve souvent à la place de "sous-liste" le terme "facteur".