CPES Scientifique

Informatique

TP 7

Listes

Résumé

Ce TP introduit les notions de base sur les listes.

1 Premiers pas avec les listes

1.1 Définition et exemples

Les listes Python permettent de manipuler des listes (finies) ordonnées (c'est-à-dire des séquences finies) de données. Pour créer une liste, on met les données dans l'ordre que l'on souhaite, séparées par virgules, et entre des crochets « `[` » et « `]` ».

Exemple 1.1. *Par exemple, pour représenter en Python la liste d'entiers*

10 3 4 5 8

on écrit simplement `[10,3,4,5,8]`*. Dans la console Python :*

```
>>> [10,3,4,5,8]
[10, 3, 4, 5, 8]
```

Attention les listes sont ordonnées.

Question 1.1. *Comparer en Python les listes* `[10,3,4,5,8]` *et* `[8,5,4,3,10]`*.*

Il existe bien entendu des liste à un élément, par exemple

```
>>> [1]
[1]
```

ainsi que la **liste vide**, notée `[]` :

```
>>> []
[]
```

Question 1.2. *Comparer en Python la liste* `[1]` *et l'entier* `1`*.*

Enfin, les variables Python peuvent désigner des listes, en d'autres termes, on peut affecter des listes aux variables Python.

Exemple 1.2. *Par exemple, dans la console Python,*

```
>>> x = [10,3,4,5,8]
>>> x
[10, 3, 4, 5, 8]
```

1.2 Accès

Pour accéder à un élément d'une liste, on met son **indice** entre crochets :

Exemple 1.3.

```
>>> y = [10,3,4]
>>> y[0]
10
>>> y[1]
3
>>> y[2]
4
```

Remarque 1.4. *On remarque que les éléments de la liste* `[10,3,4]` *sont indicés de* 0 *à* 2*.*

Si on essaye d'accéder à un indice supérieur ou égal à la longueur d'une liste, cela produit une erreur, appellée « **erreur d'indice** ». Par exemple :

```
>>> y = [10,3,4]
>>> y[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Question 1.3. *Affecter la liste* `[3,4,5,8]` *à la variable «* `z` *» et accéder au premier et au dernier élément de cette liste.*

Remarque 1.5. *En réalité, il n'est pas nécessaire qu'une liste soit désignée par une variable pour accéder à ses éléments. Par exemple, dans la console Python, on a*

```
>>> [10,3,4][0]
10
```

1.3 Modification de listes

Les listes sont des objets Python dits **modifiables**, ce qui veut dire que l'on peut modifier les éléments d'une liste. Pour cela, il suffit de réaffecter l'élément d'indice correspondant. Par exemple, avec

```
>>> y = [10,3,4]
```

on a

```
>>> y[1]
3
```

et on peut modifier l'élément d'indice 1 de « `y` » :

```
>>> y[1] = 0
>>> y
[10, 0, 4]
>>> y[1]
0
```

Remarque 1.6. *La possibilité de modifier les éléments d'une liste est un aspect important des listes sur lequel on reviendra souvent.*

1.4 Fonctions de base sur les listes

Concaténation. Les listes peuvent être concaténées (c'est-à-dire mises bout-à-bout) avec l'opérateur _ + _.

Exemple 1.7. *Dans la console :*

```
>>> [10,3,4] + [5,8]
[10, 3, 4, 5, 8]
```

Question 1.4. *Comparer en Python les listes* `([10,3,4] + [5,8])` *et* `[10,3,4,5,8]`.

La liste vide est l'élément neutre de la concaténation.

Exemple 1.8. *Dans la console :*

```
>>> [1,2,3] + []
[1, 2, 3]
>>> [] + [1,2,3]
[1, 2, 3]
```

Question 1.5. *Comparer en Python les listes* `[10,3,4]`, `([10,3,4] + [])`, *et* `([] + [10,3,4])`.

La fonction « len ». La fonction Python « `len` » renvoie la longueur d'une liste.

Question 1.6. *Utiliser la fonction « `len` » pour calculer la longueur des listes* `[]`, `[10,3,4]` *et* `[10,3,4,5,8]`.

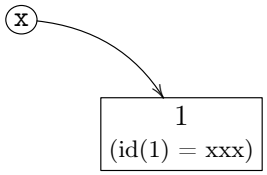
La fonction « range ». La fonction Python « `range` » permet de créer des listes de longueur arbitraire : « `list(range(n))` », où `n` est un entier, renvoie la liste des entiers de `0` à `n-1`.

Question 1.7. *Utiliser la fonction « `range` » pour créer les listes* `[]`, `[0]` *et* `[0,1,2,3,4,5,6,7,8,9]`.

2 Copie et modification de listes

Les copies et modifications de listes ont quelques subtilités qu'il faut avoir en tête. Ces subtilités sont liées à la notion de variable, telle qu'on l'a vue précédemment. [1]

Rappellons tout d'abord qu'une variable contenant un entier peut être vue comme une étiquette pointant vers la représentation mémoire de cet entier :



Le cas des listes est similaire, à la différence près qu'une variable « contenant une liste » pointe vers **une copie particulière** de cette liste.

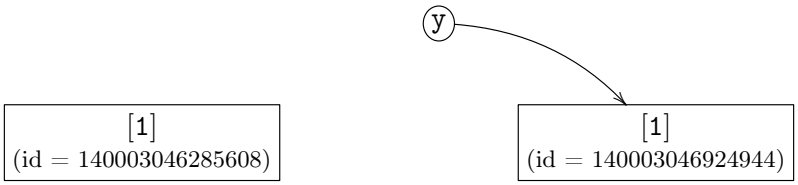
Question 2.1.

(a) Affecter l'entier 1 *à la variable « `x` », et comparer leurs identifiants* [2].

1. Voir le TP 2.
2. On rappelle que les identifiants Python sont obtenus grâce à la fonction `id`.

(b) Affecter la liste `[1]` *à la variable « y », et comparer leurs identifiants.*

La situation de la Question 2.1.(b) peut être représentée comme suit :

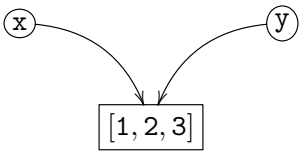


On voit donc qu'en mémoire cohabitent **deux copies distinces** de la liste `[1]`.

Dans ce contexte, il est important de connaître l'effet de l'affectation pour les listes :

Question 2.2. *Affecter la liste* `[1,2,3]` *à la variable « x », et comparer leurs identifiants. Affecter maintenant « x » à la variable « y » et comparer leurs identifiants.*

Ainsi si une variable « `x` » contient une (copie d'une) liste `ls` et que l'on affecte le contenu de « `x` » à une autre variable « `y` », alors « `y` » pointera vers **la même copie** de `ls` que « `x` ». La situation de la Question 2.2 peut être représentée comme suit :



Ceci a pour conséquence que si on modifie la liste contenue dans « `x` », alors la liste contenue dans « `y` » est aussi modifiée.

Question 2.3.

(a) Affecter la liste `[1,2,3]` *à la variable « x », et ensuite affecter « x » à la variable « y ». Modifier maintenant un des éléments de « x » et ensuite comparer « x» et « y ».*

(b) Comparer avec ce qu'il se passe pour les entiers : Affecter un entier à la variable « a », ensuite affecter « a » à la variable « b », modifier la variable « a » et comparer « a » et « b ».

Bilan et conséquences. Les comportements que l'on vient d'observer ont un nom : on dit que les listes sont manipulées **par références**, c'est-à-dire qu'une liste est uniquement une adresse mémoire (un identifiant Python) désignant une donnée, mais pas directement cette donnée. En revanche, les entiers ou les Booléens par exemple, sont eux manipulés **par valeurs**.

Cela a pour première conséquence que si on veut réélement **copier** des listes, c'est-à-dire obtenir deux copies indépendantes d'une même liste (contrairement à ce qu'il se passe à la Question 2.3), **alors il ne faut pas utiliser l'affectation**.

De manière généralle, les données Python peuvent être classifiées en deux catégories :

— les données **non-modifiables** telles que les entiers, les Booléens, les flottants, qui sont manipulées par **valeurs**,

— les données **modifiables** telles que les listes, qui sont manipulées par **références**.

Les paramètres de fonctions, lorsqu'ils sont des objets non modifiables, sont passés par **valeurs** : lorsqu'une variable est donnée en argument à une fonction, c'est en fait l'objet mémoire désigné par la variable qui est passé en argument à la fonction. Si la fonction modifie la variable correspondant à cet argument, le paramètre effectif n'est pas modifié :

```
>>> def f(x):
...   x = 1
...   return x
...
>>> a = 0
>>> f(a)
1
>>> a
0
```

A contrario, les arguments désignant des objets modifiables (comme les listes) sont passés par **références** : lorsqu'une variable est donnée en argument à une fonction, c'est en fait **l'adresse** de l'objet mémoire qui est passé en argument. Si la fonction modifie un élément de la liste passée en argument, alors le paramètre effectif est modifié :

```
>>> def f(x) :
...   x[0] = 1
...
>>> a = [2]
>>> f(a)
>>> a
[1]
```

2.1 Exercices

Question 2.4 (Copie de listes d'éléments non-modifiables)**.** *Écrire une fonction «* `copie_liste` *» qui prend en argument une liste et qui renvoie une «* **copie profonde** *» de cette liste, c'est-à-dire telle qu'on ait, dans la console*

```
>>> x = [1,2,3]
>>> y = copie_liste(x)
>>> x == y
True
>>> x[0] = 10
>>> x == y
False
```

Question 2.5 (Renversement d'une liste)**.** *Écrire une fonction «* `rev` *» qui prend en argument une liste et qui renvoie une liste ayant les mêmes éléments, mais dans l'ordre inverse.*

3 Exercices d'entraînement

Ces premiers exercices ont pour but de se familiariser avec la manipulation de listes. Ils consistent essentiellement à reprendre, dans le contexte des listes, des choses vues auparavant.

Question 3.1 (Somme et produit des éléments d'une liste)**.**

(a) Écrire une fonction Python qui prend argument une liste d'entiers et qui renvoie la somme de ses éléments.

(b) Même chose, mais avec le produit des éléments.

Question 3.2 (Suites arithmético-géométriques)**.** *Écrire une fonction Python qui prend en argument les entiers u, a, b et n et qui renvoie la liste des n-premiers termes de la suite $(u_k)_{k\in\mathbb{N}}$ définie par*

$$u_0 = u \qquad et \qquad u_{n+1} = a \cdot u_n + b$$

Si $n = 0$ alors la fonction doit renvoyer la liste vide, et si $n > 0$ alors elle doit renvoyer la liste $[u_0, \ldots, u_{n-1}]$.

Question* 3.3 (Suite de Fibonacci)**.** *Écrire une fonction Python qui prend en argument un entier n et qui renvoie la liste des n-premiers termes de la suite de Fibonnaci.*

Si $n = 0$ alors la fonction doit renvoyer la liste vide, et si $n > 0$ alors elle doit renvoyer la liste $[u_0, \ldots, u_{n-1}]$ où $u_0 = u_1 = 1$ et $u_{k+2} = u_{k+1} + u_k$.

Question 3.4 (Décomposition base k)**.** *Écrire une fonction Python qui prend en argument deux entiers n et k et qui renvoie la liste des coefficients de la décomposition base k de n.*

4 Notions d'algorithmique sur les tableaux

4.1 Algorithmes de recherche

4.1.1 Recherche simple

Question 4.1 (Recherche simple)**.** *Écrire une fonction qui prend en argument une liste et une valeur, et qui renvoie « `True` » si cette valeur apparaît dans la liste et « `False` » sinon.*

Question 4.2 (Recherche du minimum d'une liste d'entiers)**.** *Écrire une fonction qui prend en argument une liste d'entiers et qui renvoie son minimum.*

4.2 Tris

Question* 4.3 (Liste triée ?)**.** *Écrire une fonction Python qui prend en argument une liste d'entiers ; et qui renvoie « `True` » si la liste est triée et « `False` » sinon.*