Chapitre 3 : Notion de listes

Axel Carpentier

Informatique

Python

Table des matières

Structure de données

Structure de données

Une structure de données est une façon de ranger et d'ordonner des objets. Il en existe de plusieurs types, qui se distinguent par la façon d'accéder aux éléments de la structure de données et par la façon de modifier cette dernière (en ajoutant ou en ôtant des éléments). Nous utiliserons principalement :

- les chaînes de caractères, qui sont des successions de caractères délimités par des guillemets (simples ou doubles);
- les listes, qui sont des successions d'objets séparés par des virgules et délimités par des crochets;
- les tuples, qui sont des successions d'objets séparés par des virgules et délimités par des parenthèses.

Structure de données

Exemple :

```
1 >>> type('1234')
2 <class 'str'>
3
4 >>> type([1,2,3,4])
5 <class 'list'>
6
7 >>> type((1,2,3,4))
8 <class 'tuple'>
```

Structure de données

Les fonctions `str`, `list`, `tuple` permettent de convertir un type de structure de donnée en une autre :

```
>>> list('1234')
['1','2','3','4']

>>> tuple([1,2,3,4])
(1,2,3,4)

>>> str((1,2,3,4))
'(1,2,3,4)'
```

Construction d'une liste

Définition et accès aux éléments

Une liste est une structure de données linéaire, les objets étant enclos par des crochets et séparés par des virgules.

```
1 l = [0 , 3.4 , 'M.Carpentier' , 7 , 'ri' ]
```

Comme on peut le constater, une liste peut contenir différents types d'élémentss (des entiers, des flottants, des chaînes de caractères...).
Notons déjà qu'une liste étant elle-même un objet python, il est tout à fait possible qu'une liste contienne des listes parmi ses éléments :

```
1 L = [[1] , [2 , 3] , [4 , 5, [6]]]
```

Définition et accès aux éléments

Les listes sont des objets fondamentaux, pas qu'en informatique. À ce titre, on peut leur trouver bien des utilisations :

- Coordonnées d'un point $A(4;5)$ dans le plan :

```
1 A = [4 , 5]
```

- Coordonnées d'un point $B(-2,1\ ;\ 7\ ;\ 8,2)$ dans l'espace

```
1 B = [-2.1 , 7 , 8.2]
```

- ...

On accède à chaque élément individuel de la liste par l'intermédiaire de son index.

Remarque

Attention, le premier élément de la liste possède l'index 0 (tout comme les chaînes de caractères).

Définition et accès aux éléments

Pour connaître l'index du dernier élément, on peut calculer la longueur d'une liste à l'aide de la fonction `len` ; l'index du dernier élément d'une liste nommée `l` est donc `len(l)-1`.

```
1  l = [0 , 3.4 , 'M.Carpentier' , 7 , 'ri' ]
2  L = [[1] , [2 , 3] , [4 , 5, [6]]]
3
4  >>> l[2]
5  'M.Carpentier'
6  >>> L[1]
7  [2 , 3]
8  >>> L[1][0]
9  2
10 >>> len(L)
11 3
```

Définition et accès aux éléments

Lorsque l'index est négatif, le décompte est pris en partant de la fin; ainsi le dernier élément porte l'index 1, l'avant-dernier l'index 2, etc...

```
1  l = [0 , 3.4 , 'M.Carpentier' , 7 , 'ri' ]
2  L = [[1] , [2 , 3] , [4 , 5, [6]]]
3
4  >>> l[-1]
5  'ri'
6
7  >>> L[-2]
8  [2 , 3]
9
10 >>> L[-2][0]
11 2
```

Définition et accès aux éléments

Python permet d'effectuer des coupes dans une liste : si `l` est une liste, alors `l[i:j]` crée une nouvelle liste constituée des éléments dont les index sont compris entre `i` et `j-1` :

```
1 l = [0 , 3.4 , 'M.Carpentier' , 7 , 'ri' ]
2
3 >>> l[1:3]
4 [3.4 , 'M.Carpentier' , 7]
```

Lorsque l'index `i` est absent, il est pris par défaut égal à 0; lorsque `j` est absent, il est pris par défaut égal à la longueur de la liste.

Définition et accès aux éléments

Si nécessaire, la syntaxe `l[debut:fin]` possède un troisième paramètre (égal par défaut à 1) indiquant le pas de la sélection. La commande `l[debut:fin:pas]` donne tous les éléments de la liste dont les index sont compris entre debut (au sens large) et fin (au sens strict) et espacés d'un pas égal à `pas`.

Exercice :

Qu'affiche le programme suivant ?

```
1 p = [0 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 , 512 , 1024 ]
2 print(p[2:9:3])
```

Construction d'une liste

Initialisation d'une liste par compréhension

Comme toutes les variables en python, on peut initialiser une liste directement en la définissant (comme dans tous les exemples précédents).
Si on y tient vraiment, par exemple pour éviter les interférences avec une variable du même qui pourrait avoir été utilisée auparavant, la commande suivante convient :

```
1 L=[]
```

Python est un langage qui permet de définir des listes en compréhension, c'est-à-dire des listes définie par filtrage, ou opération terme à terme, sur le contenu d'une autre liste.
Cette construction est assez similaire à celle utilisée en mathématiques, plus précisément en théorie des ensembles (ce qui n'est pas si étonnant, une liste étant un ensemble...).

Initialisation d'une liste par compréhension

Exemple :

- Initialiser une liste de la taille souhaitée, avec des zéros par exemple.

```
1 U=[0 for i in range(10)]
2 print(U)
3 #Affiche [0,0,0,0,0,0,0,0,0,0]
```

- Ajouter terme à terme les éléments de deux listes (de même taille bien sûr):

```
1 L=[0,1,2]
2 M=[3,4,5]
3 N=[L[i]+M[i] for i in range(len(L))]
4 print(N)
5 #affiche [3,5,7]
```

Initialisation d'une liste par compréhension

- Faire des calculs terme à terme sur les éléments d'un liste:

```
1 L=[0,1,2]
2 N=[3*x+2 for x in L]
3 print(N)
4 #affiche [2,5,8]
```

- Représenter des ensembles mathématiques $\{x \in [|0, 11|],\ x^2 \leq 50\}$:

```
1 >>> [x for x in range(11) if x * x <= 50]
2 [0, 1, 2, 3, 4, 5, 6, 7]
```

Initialisation d'une liste par compréhension

- Calculer le produit cartésien de deux listes :

```
1 a, b = [1, 3, 5], [2, 4, 6]
2
3 >>>  [(x, y) for x in a for y in b]
4 [(1, 2), (1, 4), (1, 6), (3, 2), (3, 4), (3, 6),
5 (5, 2), (5, 4), (5, 6)]
```

Construction d'une liste

Mutation d'une liste

En python, une liste est un objet mutable, dans le sens où il est possible de modifier le contenu d'une liste. Il est possible de modifier un ou plusieurs éléments de la liste, d'en supprimer ou d'en ajouter.

- Si `l` est une liste, l'instruction `l[i] = x` remplace l'élément d'indice `i` par la valeur `x`.

```
1  l = list(range(6))
2  l[2] = 'M.Carpentier'
3
4  >>>  l
5  [0,1,'M.Carpentier',3,4,5]
```

Mutation d'une liste

- L'instruction `del` permet de supprimer un ou plusieurs éléments :

```
1  L=[1,2,5,78]
2  del(L[0])
3
4  l = list(range(11))
5  del(l[3:6])
6
7  >>> L
8  [2,5,78]
9
10 >>> l
11 [0, 1, 2, 6, 7, 8, 9, 10]
```

Mutation d'une liste

- L'instruction `del` permet de supprimer un ou plusieurs éléments :

```
1  L=[1,2,5,78]
2  del(L[0])
3
4  l = list(range(11))
5  del(l[3:6])
6
7  >>> L
8  [2,5,78]
9
10 >>> l
11 [0, 1, 2, 6, 7, 8, 9, 10]
```

Mutation d'une liste

- Il existe principalement deux méthodes associées aux listes qui permettent l'insertion de nouveaux éléments : l'instruction `append(x)` ajoute l'élément `x` en fin de liste, et l'instruction `insert(i, x)` insère l'élément `x` à l'index `i`.

```
1 l = ['a', 'b', 'c', 'd']
2 l.append('e')
3 print(l)
4 #affiche ['a', 'b', 'c', 'd', 'e']
5
6 l.insert(2, 'x')
7 print(l)
8 #affiche ['a', 'b', 'x', 'c', 'd', 'e']
```

Mutation d'une liste

Il est également possible d'effectuer d'autres types d'actions sur les listes (moins couramment utilisées) via les instructions suivantes.

- L'instruction `remove(x)` supprime la première occurence de `x` dans la liste.

```
1 l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
2 l.remove(4)
3 print(l)
4 #affiche [1, 2, 3, 5, 1, 2, 3, 4, 5]
```

- L'instruction `pop(i)` supprime l'élément d'indice `i` et retourne cet élément.

```
1 l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
2
3 >>>l.pop(4)
4 5
```

Mutation d'une liste

- L'instruction `reverse()` inverse l'ordre des éléments d'une liste.

```
1 l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
2 l.reverse()
3
4 >>> l
5 [4, 3, 2, 1, 4, 3, 2, 1]
```

- L'instruction `sort()` trie la liste (par ordre croissant).

```
1 l = [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
2 l.sort()
3
4 >>> l
5 [1, 1, 2, 2, 3, 3, 4, 4]
```

Mutation d'une liste

- L'instruction `extend(L)` ajoute les éléments de la liste `L` dans la liste.

```
1  A=[1,3,5,7]
2  B=[2,4,6,8]
3  A.extend(B)
4
5  >>> A
6  [1, 3, 5, 7, 2, 4, 6, 8]
```

Exercice :

Définir une fonction Python qui retourne le même résultat que l'instruction `reverse()`.

Parcours de listes

Parcours de listes

Il a été mention à plusieurs reprises dans les chapitres précédents des boucles enumérées de la forme :

```
1 for i in range(...)
```

Ce ne sont en réalité qu'un cas particulier d'une syntaxe plus générale :

```
1 for x in L
```

Une telle boucle définit une variable (ici nommée `x`) qui prend successivement toute les valeurs de la liste (ici nommée `L`) par ordre d'indice croissant.

Il est possible de trouver diverses applications à cette méthode qui simplifie bien des fonctions.

Parcours de listes

- Calcul de la somme des éléments d'une liste.

```
1  def somme(L):
2      s=0
3      for x in L:
4          s+=x
5      return s
```

- Calcul de la moyenne d'une liste.

```
1  def moyenne(L):
2      n=len(L)
3      s=0
4      for x in L:
5          s+= x
6      return s/n
```

Parcours de listes

- Calcul de la variance d'une liste.

```
1  def variance(L):
2      n=len(L)
3      s,c=0,0
4          for x in l:
5      s+= x
6      c+= x * x
7      return c/n - (s/n) * (s/n)
```

Exercice :

1. Ecrire une fonction `moyenne_bis(L)` qui fait appel à la fonction `somme(L)`.
2. Ecrire une fonction `variance_bis(L)` qui fait appel à la fonction `moyenne(L)`.