Structure de données élémentaires en Python : listes, tuples, chaînes de caractères, dictionnaires

DIU Enseigner l'Informatique au Lycée

E. Jahier, L. Rieg, C. Parent-Vigouroux, B. Wack

UFR IM2AG, Université Grenoble Alpes

juillet 2020

Remerciements : Stéphane Gonnord, Matthieu Moy, Marc de Falco

Plan

Plan

Les tableaux Python

- Hétérogènes

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

- Accès aléatoire, **en arrière**

```
>>> L[0], L[-1]
1, 3
```

- **Modifiables** (on dit aussi *mutable*)

```
>>> L[0] = 42
```

- Ajout en queue (modifie le tableau, ne renvoie rien)

```
>>> L.append(4)
```

- Retrait en queue (modifie le tableau, renvoie l'élément retiré)

```
>>> L.pop()
4
```

Les tableaux Python (on parle plutôt de listes)

- Hétérogènes

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

- Accès aléatoire, **en arrière**

```
>>> L[0], L[-1]
1, 3
```

- **Modifiables** (on dit aussi *mutable*)

```
>>> L[0] = 42
```

- Ajout en queue (modifie le tableau, ne renvoie rien)

```
>>> L.append(4)
```

- Retrait en queue (modifie le tableau, renvoie l'élément retiré)

```
>>> L.pop()
4
```

Quelques explications

Une liste est un objet

- La plupart des opérations de manipulation sont donc des méthodes.

```
>>> nomdelaliste.nomdelamethode(...)
```

- Comme pour les modules :

```
>>> dir(maliste)
# liste des methodes
>>> help(maliste.methode) # doc d'une methode
```

Attention : ce ne sont pas des listes chaînées

- accès et modification aléatoire en temps constant
- ajout et suppression en temps constant amorti

Concaténation

La **concaténation** + crée une nouvelle liste :

```
>>> a = [2, 4, 6]
>>> b = [8, 10, 12]
>>> c = a + b
>>> c
[2, 4, 6, 8, 10, 12]
```

Concaténation

La **concaténation** + crée une nouvelle liste :

```
>>> a = [2, 4, 6]
>>> b = [8, 10, 12]
>>> c = a + b
>>> c
[2, 4, 6, 8, 10, 12]
```

Concaténations multiples : `L * n` concatène n instances de `L`.

```
>>> [4, 2, 1] * 4
[4, 2, 1, 4, 2, 1, 4, 2, 1, 4, 2, 1]
```

Autres opérateurs

- L'opérateur **in** permet de tester l'appartenance à une liste :

```
>>> 7 in [3, 5, 7, 11, 13]
True
>>> if v in premiers:
...     print(v, 'est un nombre premier.')
```

On dispose aussi de `v not in L`.

Autres opérateurs

- L'opérateur `in` permet de tester l'appartenance à une liste :

```
>>> 7 in [3, 5, 7, 11, 13]
True
>>> if v in premiers:
...     print(v, 'est un nombre premier.')
```

  On dispose aussi de `v not in L`.

- D'autres méthodes utiles : `count`, `index`, `sort`, `sum`...
- On peut aussi donner à `pop` l'indice de l'élément à supprimer :

```
>>> c.pop(2)
6
>>> c
[2, 4, 8, 10, 12]
```

Plan

Listes définies «par compréhension»

- De manière analogique à la notation ensembliste $\{3i^2 + 1 \mid 1 \leq i \leq 9\}$ en mathématiques :

```
>>> [ 3*i**2 + 1 for i in range(1,10) ]
[4, 13, 28, 49, 76, 109, 148, 193, 244]
```

Listes définies «par compréhension»

- De manière analogique à la notation ensembliste $\{3i^2 + 1 \mid 1 \leq i \leq 9\}$ en mathématiques :

```
>>> [ 3*i**2 + 1 for i in range(1,10) ]
[4, 13, 28, 49, 76, 109, 148, 193, 244]
```

- Il est possible de combiner les compréhensions :

```
>>> [(i,j) for i in range(4) for j in range(i)]
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2)]
```

Pour des listes de listes

Attention, c'est différent si les compréhensions sont vraiment emboîtées :

```
>>> [[(i,j) for i in range(4)] for j in range(i)]
NameError: name 'i' is not defined
```

```
>>> [[(i,j) for j in range(i)] for i in range(4)]
[[], [(1, 0)], [(2, 0), (2, 1)], [(3, 0), (3, 1),
```

Pour des listes de listes

Attention, c'est différent si les compréhensions sont vraiment emboîtées :

```
>>> [[(i,j) for i in range(4)] for j in range(i)]
NameError: name 'i' is not defined

>>> [[(i,j) for j in range(i)] for i in range(4)]
[[], [(1, 0)], [(2, 0), (2, 1)], [(3, 0), (3, 1),
```

On peut également également rajouter un **if** :

- soit pour sélectionner :

```
>>> [i**2 for i in range(10) if i%3 != 0]
[1, 4, 16, 25, 49, 64]
```

- soit pour obtenir des valeurs conditionnelles :

```
>>> [i if i%2==0 else -i for i in range(10)]
[0, -1, 2, -3, 4, -5, 6, -7, 8, -9]
```

Plan

Principes de base

On peut découper une liste en « tranches » (*slicing* en anglais).
Par exemple, pour obtenir les éléments d'indices 2 à 4 d'une liste :

```
>>> L
['do', 're', 'mi', 'fa', 'sol', 'la', 'si', 'do']
>>> L[2:4]
['mi', 'fa']
```

Attention : comme d'habitude en Python, intervalle fermé à gauche et ouvert à droite.
`L[i:j]` désigne les éléments d'indice compris dans $[\![i; j[\![$.

En particulier `L[i:i]` est une liste vide.

À noter : une tranche est une copie d'une sous-liste (voir plus loin la section sur les références)

Plan

Rappel : itération simple

En Python, un objet est dit *itérable* s'il est capable de renvoyer un par un tous ses éléments. Quand on écrit `for x in a`, la variable x prend successivement toutes les valeurs des éléments de a.

```
>>> for couleur in L :
        print(couleur)
bleu
vert
rouge
```

On peut itérer sur les `range` mais aussi sur les listes, les chaînes...

Exemple : extraction d'une sous-liste

Seuil

Cette fonction prend une liste `L` d'entiers et renvoie la sous-liste des entiers dépassant un seuil `s`.

```
def seuil(L, s):
    M = []
    for x in L:
        if x > s:
            M.append(x)
    return M
```

Le Pythonneux écrira plutôt `[ x for x in L if x>s ]` !

Double parcours

Bégaiement

Cette fonction prend une liste de chaînes et affiche chaque caractère de chaque chaîne deux fois de suite.

```
def begaie(L):
    for chaine in L:
        for carac in chaine:
            print (carac, carac)
```

Plan

Tuples (p-uplets) et chaînes de caractères

p-uplets :

```
>>> t = (True, 42)
>>> (x, y) = (2, 3)
>>> x, y = y, x
```

Chaînes :

```
>>> a = 'Lycée'
>>> a.upper()  # nouvelle chaine
'LYCÉE'
>>> a[0]
'L'
>>> '-'.join(['Un', 'mot', 'et', 'un', 'autre'])
'Un-mot-et-un-autre'
```

Ces types sont non modifiables.

Les dictionnaires

- Associent des valeurs à des clés.

```
>>> d = { 'cle1':17, 42:'val2' }
```

- Les clés doivent être de type non modifiable
(souvent entiers et/ou chaînes de caractères)

```
>>> d['abc'] = 'def'
```

- Une même clé ne peut apparaître qu'en un exemplaire

```
>>> d[42] = 'toto'
>>> d
{'cle1':17, 42:'toto', 'abc':'def'}
```

Exemples d'utilisations :

- Tableaux non contigüs, ou remplis dans le désordre
- Enregistrements nommés

Opérateurs sur les dictionnaires

- L'accès est rapide (temps constant en moyenne)

```
>>> d['cle1']
17
>>> 'abc' in d
True
```

- Parcours :

```
>>> for k in d:   # dans l'ordre d'insertion
        print(k, '--->', d[k])
cle1 ---> 17
42 ---> toto
abc ---> def
```

Plan

Les cauchemars commencent : Références implicites

Les listes, comme tous les autres types modifiables, sont manipulées par référence.

Version int

```
>>> x = 3
>>> y = x
>>> x = x + 1
>>> x
4
>>> y
3
```

Version liste

```
>>> a = [3]
>>> b = a
>>> a.append(1)
>>> a
[3, 1]
>>> b
[3, 1]
```

Les deux variables `a` et `b` pointent vers la *même* liste (aliasing).

Les cauchemars commencent : Références implicites

Les listes, comme tous les autres types modifiables, sont manipulées par référence.

Version int

```
>>> x = 3
>>> y = x
>>> x = x + 1
>>> x
4
>>> y
3
```

Version liste

```
>>> a = [3]
>>> b = a
>>> a.append(1)
>>> a
[3, 1]
>>> b
[3, 1]
```

Les deux variables `a` et `b` pointent vers la *même* liste (aliasing).

Pour obtenir une *vraie* copie :
`y = copy.copy(x)` après un `import copy`

Attention au passage de paramètres

Puisque *tout* est référence :

```
def f(t):
    t[2] = 0

a = [1, 2, 3, 4, 5]
f(a)
print(a)      # [1, 2, 0, 4, 5]
```

Une fonction est susceptible de modifier ses arguments pour tous les types de données mutables.

Cas concrets qu'on rencontrera tôt ou tard

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

Cas concrets qu'on rencontrera tôt ou tard

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

- Si, maintenant, on affectait la case (0,0) :

```
>>> m[0][0] = 1
>>> m
[[1, 0], [1, 0]]
```

  - Argh ! Comment expliquer ça ?

Cas concrets qu'on rencontrera tôt ou tard

- Considérons cet exemple :

```
>>> m = [[0]*2]*2
>>> m
[[0, 0], [0, 0]]
```

- Si, maintenant, on affectait la case (0,0) :

```
>>> m[0][0] = 1
>>> m
[[1, 0], [1, 0]]
```

  - Argh ! Comment expliquer ça ?

- Python va d'abord évaluer l'expression `[0]*2`, et comme c'est une liste va utiliser la référence du résultat pour l'opération suivante