

Bases d'algorithmique et de programmation

Sébastien Gagné, Université d'Orléans

L1 Algo - Prog - S1

Cours 04 - Structures de données complexes - Tuples et Listes

Plan du cours

- Modélisation des tableaux
- structure dynamique : les listes
- structure statique : les tuples

Les tableaux - algo

En algorithmique on évoque souvent la structure de tableau pour une collection de valeurs accessibles par un indice.

Dans tous les langages de programmation (java, c, ...) on retrouve cette représentation de tableau.

Le premier indice est 0, le dernier est la taille du tableau - 1.

Particularité python : on peut utiliser les indices négatifs (-1 pour le dernier élément, - taille pour le premier).

En python, on dispose des types suivants, qui peuvent représenter des tableaux :

- ***list***, qui est une structure muable (modifiable) ;
- ***tuple***, qui est une structure immuable (non modifiable).

Création d'une liste

Une liste peut être créée de différentes façons :

- création d'une liste vide

```
>>> l1 = []; print(l1)
[]
>>> l2 = list(); print(l2)
[]
```

la deuxième liste est créée par un appel explicite au *constructeur* **list()** (comme un appel de fonction).

- création d'une liste à partir d'éléments

```
>>> l3 = [4, 7, 3, 2]; print(l3)
[4, 7, 3, 2]
```

Création d'une liste

Une liste peut aussi être créée par le **constructeur list()**, en lui passant cette fois un itérable en paramètre :

```
>>> c = "Hello"
>>> t = (4, 3, 7)
>>> l = [5, 2, 6]
>>> d = {"x" : 4.2, "y" : 1.5}
>>> s = {4, 1, 3}
>>> r = range(2, 16, 3)
```

```
>>> list(c)
['H', 'e', 'l', 'l', 'o']
>>> list(t)
[4, 3, 7]
>>> list(l)
[5, 2, 6]
```

```
>>> list(d)
['x', 'y']
>>> list(s)
[1, 3, 4]
>>> list(r)
[2, 5, 8, 11, 14]
```

Les listes sont muables !

On peut changer la liste en agissant au même emplacement mémoire. C'est ce qui caractérise les objets **muables**.

```
>>> liste1 = [3, 1, 4, 1, 6]
>>> id(liste1)
127529362763328
>>> liste1[4] = 5
>>> liste1
[3, 1, 4, 1, 5]
>>> id(liste1)
127529362763328
>>> liste1.append(9)
>>> liste1
[3, 1, 4, 1, 5, 9]
>>> id(liste1)
127529362763328
```

Exemple d'application

On procède à l'expérience suivante.

On relève les prix du litre d'essence dans une série de stations.

On conservera les prix relevés, pour des statistiques ultérieures.

On arrête la saisie en entrant un prix égal à 0.

Le programme suivant permet cette saisie :

```
1 les_prix = []
2
3 fin_de_saisie = False
4
5 while not fin_de_saisie :
6     prix = float(input("nouveau prix : "))
7     fin_de_saisie = (prix == 0)
8     if prix == 0 :
9         print("saisie terminée")
10    else :
11        les_prix.append(prix)
12
```

```
nouveau prix : 2.18
nouveau prix : 2.16
nouveau prix : 2.13
nouveau prix : 2.17
nouveau prix : 2.12
nouveau prix : 0
saisie terminée
>>> les_prix
[2.18, 2.16, 2.13, 2.17, 2.12]
```

Exemple d'application

On peut maintenant opérer divers traitements sur la liste :

- trier la liste,
- calculer sa moyenne,
- calculer sa médiane, ...

```
>>> les_prix.sort()
>>> les_prix
[2.12, 2.13, 2.16, 2.17, 2.18]
>>> n = len(les_prix)
>>> mediane = les_prix[n // 2]
>>> mediane
2.16
>>> moyenne = sum(les_prix) / n
>>> moyenne
2.152
```

Rappel – méthodes sur les *list*

- `list1.append(n)` accole la valeur `n` en fin de *list1* ;
- `list1.remove(n)` supprime la première valeur `n` de *list1* ;
- `list1.index(n)` renvoie le premier indice où se trouve `n` (erreur si `n` absent) ;
- `list1.insert(i, n)` insère la valeur `n` à l'indice `i` ;
- `list1.pop(i)` supprime l'élément d'indice `i` et le retourne ;
- `list1.pop()` supprime le dernier élément et le retourne ;
- `list1.count(n)` retourne le nombre d'occurrences de `n` ;
- `list1.reverse()` renverse la liste ;
- `list1.sort()` trie la liste ;

Application

Complétez l'évolution de *liste*, *a* et *b* au cours des instructions

instruction	liste	a	b
<code>liste = [1, 5, 3, 4]</code>	<code>[1, 5, 3, 4]</code>		
<code>liste.append(6)</code>			
<code>a = liste.pop(2)</code>			
<code>b = liste.pop(0)</code>			
<code>liste.sort()</code>			
<code>liste.append(a)</code>			
<code>liste.insert(3, b)</code>			
<code>liste.reverse()</code>			
<code>liste.append(liste.pop(2))</code>			
<code>liste.insert(liste.index(4), 2)</code>			
<code>liste.insert(5, 2)</code>			
<code>liste.remove(2)</code>			
<code>liste.pop()</code>			

Application

Complétez l'évolution de *liste*, *a* et *b* au cours des instructions

instruction	liste	a	b
<code>liste = [1, 5, 3, 4]</code>	[1, 5, 3, 4]		
<code>liste.append(6)</code>	[1, 5, 3, 4, 6]		
<code>a = liste.pop(2)</code>	[1, 5, 4, 6]	3	
<code>b = liste.pop(0)</code>	[5, 4, 6]		1
<code>liste.sort()</code>	[4, 5, 6]		
<code>liste.append(a)</code>	[4, 5, 6, 3]		
<code>liste.insert(3, b)</code>	[4, 5, 6, 1, 3]		
<code>liste.reverse()</code>	[3, 1, 6, 5, 4]		
<code>liste.append(liste.pop(2))</code>	[3, 1, 5, 4, 6]		
<code>liste.insert(liste.index(4), 2)</code>	[3, 1, 5, 2, 4, 6]		
<code>liste.insert(5, 2)</code>	[3, 1, 5, 2, 4, 2, 6]		
<code>liste.remove(2)</code>	[3, 1, 5, 4, 2, 6]		
<code>liste.pop()</code>	[3, 1, 5, 4, 2]		

Autre méthode des *list* : *split*

- si *ch* est de type *str* et *s* un caractère (le "séparateur"), alors *ch.split(s)* retourne la liste des sous-chaînes de *ch*, après découpage selon chaque occurrence du séparateur ;

```
1 numTel = input('numéro de téléphone : ')
2 s = '.'
3 liste = numTel.split(s)
4 print(liste)
```

```
numéro de téléphone : 06.45.21.98.74
['06', '45', '21', '98', '74']
```

Cette méthode, qui s'applique à une chaîne de caractères, retourne une liste.

Méthode des *str* liée aux *list* : *join*

- si une liste n'est composée que de valeurs de type *str*, et si *s* est un caractère (le "séparateur"), alors *s.join(liste)* retourne la chaîne de caractères obtenue en concaténant les éléments de la liste, séparés par le séparateur *s*.

```
1 liste2 = ['c', 'a', 'l', 'c', 'u', 'l']  
2 separateur = '-'  
3 chaîne2 = separateur.join(liste2)  
4 print(chaîne2)
```

c-a-l-c-u-l

Cette méthode s'applique à une chaîne de caractères, prend en paramètre une liste et retourne une chaîne.

Quelques fonctions liées aux *list*

- comme pour tous les itérables, la fonction *len* (fonction native de python) retourne la taille de la liste.
- si les **éléments** d'une liste sont **comparables** entre eux, les fonctions *min* et *max* (déjà évoquées) retournent les valeurs minimale et maximale de la liste.

```
>>> liste1 = [3, 1, 4, 1, 5, 9]
>>> len(liste1)
6
>>> max(liste1)
9
>>> min(liste1)
1
```

Quelques fonctions liées aux *list*

ATTENTION ! Si les éléments ne sont pas comparables, les fonctions *min* et *max* déclenchent une erreur. Il est donc essentiel de vérifier la **cohérence** des types.

```
>>> liste2 = [1, 2, True, "Bye"]
>>> min(liste2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

```
>>> liste3 = [1, 2, False]
>>> min(liste3)
False
>>> max(liste3)
2
```

ici, *False* est assimilé à 0 (*True* serait assimilé à 1)

Quelques fonctions liées aux *list*

La fonction **sum** retourne la somme des éléments d'une liste (ou d'un itérable). Attention, les éléments doivent être numériques :

```
>>> liste1 = [4, 2, 3.1, 1.5]
>>> sum(liste1)
10.6
```

Dans l'exemple suivant, on définit une fonction **moyenne** à partir de **sum** et de **len** :

```
1 def moyenne(liste) :
2     return sum(liste) / len(liste)
```

Quelques fonctions liées aux *list*

La fonction *enumerate* permet de parcourir une liste (ou un itérable) en nommant l'indice et l'élément correspondant :

```
>>> liste1 = [3, True, "toto"]
>>> for element in liste1 :
    print(element)

3
True
toto

>>> for i in range(len(liste1)) :
    print("élément d'indice", i, ":", liste1[i])

élément d'indice 0 : 3
élément d'indice 1 : True
élément d'indice 2 : toto

>>> for i, element in enumerate(liste1) :
    print("élément d'indice", i, ":", element)

élément d'indice 0 : 3
élément d'indice 1 : True
élément d'indice 2 : toto
```

Compréhension de listes

Python propose une syntaxe efficace pour générer certaines listes : la **compréhension de listes**.

Exemple : créer la liste des entiers entre 0 et 20 dont le reste modulo 3 est 2 et dont le reste modulo 4 n'est pas 1.

```
>>> l1 = [n for n in range(21) if n % 3 == 2 and n % 4 != 1]
>>> l1
[2, 8, 11, 14, 20]
```

C'est l'équivalent de :

```
>>> l1 = []
>>> for n in range(21) :
        if n % 3 == 2 and n % 4 != 1 :
            l1.append(n)
```

Le mot-clé *del*

Le mot-clé *del* détruit le lien qui existe entre le nom d'une variable et l'emplacement mémoire correspondant.

Ce n'est **pas une fonction**.

```
après liste1 = ["a", 2.5, "bye"] :  
- liste1           : ['a', 2.5, 'bye']  
- id(liste1)       : 132346231478464  
- id(liste1[0])    : 132346231858736  
- id(liste1[1])    : 132346228898704  
- id(liste1[2])    : 132346226873200  
  
après del liste1[1] :  
- liste1           : ['a', 'bye']  
- id(liste1)       : 132346231478464  
- id(liste1[0])    : 132346231858736  
- id(liste1[1])    : 132346226873200
```

Les éléments "a" et "bye" n'ont pas bougé en mémoire. Le lien vers 2.5 est cassé.

Bilan sur les listes

- Une liste est une collection **ordonnée** et **muable** (ou encore **modifiable**) de données.
- Chaque élément de la liste est accessible via son **indice** par l'**opérateur** `[ ]`.
- Cet indice d'accès peut être **positif** (de 0 à taille - 1) ou **négatif** (de -1 à -taille).
- Les éléments d'une liste sont **modifiables**.
- Les méthodes qui modifient une liste sont efficaces (voir comparaison entre *append* et `+`) car elles ne nécessitent pas de temps pour la gestion de la mémoire.
- Les listes sont adaptées aux problématiques où les données évoluent.

Création d'un tuple

Un tuple peut être créé de différentes façons :

- création d'un tuple vide

```
>>> t1 = (); print(t1)
()
>>> t2 = tuple(); print(t2)
()
```

le deuxième tuple est créé par un appel explicite au constructeur `tuple()` (comme un appel de fonction).

- création d'un tuple à partir d'éléments

```
>>> t3 = (4, 5, 3, 6); print(t3)
(4, 5, 3, 6)
```

Création d'un tuple

Comme une liste, un tuple peut être créé par un appel au **constructeur**, appelé comme une fonction "**tuple**" en lui passant un itérable en paramètre :

```
>>> c = "Hello"
>>> t = (4, 3, 7)
>>> l = [5, 2, 6]
>>> d = {"x" : 4.2, "y" : 1.5}
>>> s = {4, 1, 3}
>>> r = range(2, 16, 3)
```

```
>>> tuple(c)
('H', 'e', 'l', 'l', 'o')
>>> tuple(t)
(4, 3, 7)
>>> tuple(l)
(5, 2, 6)
```

```
>>> tuple(d)
('x', 'y')
>>> tuple(s)
(1, 3, 4)
>>> tuple(r)
(2, 5, 8, 11, 14)
```

Les tuples sont immuables !

- Tout changement de valeur d'un élément de tuple est impossible et déclenche une erreur. De même, l'instruction `del` est inopérante sur l'un des éléments d'un tuple ;
- Toute nouvelle affectation du tuple détruit et recrée le tuple (l'identifiant change !).

```
>>> t1 = (5, 4, 3)
>>> id(t1)
129796492754816
>>> t1
(5, 4, 3)
>>> t1 = t1 + (2, 1)
>>> id(t1)
129796490239392
>>> t1
(5, 4, 3, 2, 1)
```

Opérateur $+$ sur les tuples

- On peut (déjà vu au CM 02) concaténer des tuples (ou des listes, ou des chaînes de caractères). Cette opération s'effectue par l'opérateur $+$;
- Pour les tuples, il y a quelques précautions à prendre, liées aux parenthèses.

```
>>> t1 = (5, 4, 3)
>>> t2 = t1 + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple

>>> t2 = t1 + (2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "int") to tuple

>>> t2 = t1 + (2,)
>>> t2
(5, 4, 3, 2)
```

Bilan sur les tuples

- Un tuple est une collection **ordonnée** et **immuable** (donc **non modifiable**) de données.
- Chaque élément du tuple est accessible via son **indice** par l'opérateur `[ ]`.
- Cet indice d'accès peut être **positif** (de 0 à taille - 1) ou **négatif** (de -1 à -taille).
- Les éléments d'un tuple ne sont pas **modifiables**.
- Les tuples sont adaptés aux problématiques où les données sont fixes.