

Bases d'algorithmique et de programmation

Sébastien Gagné, Université d'Orléans

L1 Algo - Prog - S1

Cours 05 - Structures de données complexes -
Dictionnaires, Ensembles

Plan du cours

- Les tableaux associatifs (dictionnaires)
- méthodes et fonctions sur les tableaux associatifs
- Les ensembles
- méthodes et fonctions sur les ensembles

Les tableaux associatifs

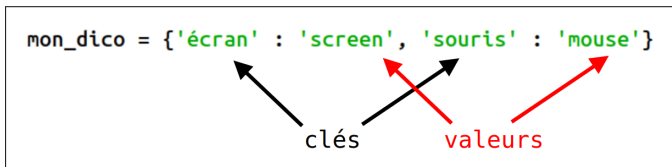
Certains langages proposent une structure particulière de données : les **tableaux associatifs** (ou **dictionnaires**).

Le type python correspondant est le type ***dict***.

- Un **tableau associatif** fonctionne avec des **clés** ;
- Ces **clés** sont des entiers ou des chaînes de caractères ;
- A chaque **clé** correspond une **valeur** ;
- Le tableau associatif se présente comme une collection **non ordonnée** de **paires clé/valeur** entre accolades.
- les paires clé/valeur sont séparées par une virgule, et la clé est séparée de la valeur par un " : " ;

Exemple de tableau associatif

Voici le tableau associatif (ou dictionnaire) `mon_dico`.



Son type python est *dict*.

C'est une variable python comportant des paires clé / valeur.

- Les variables de type *dict* ne fonctionnent pas avec des indices, mais avec des *clés*.
- Les variables de type *dict* sont **muables** ;
- Les variables de type *dict* sont **itérables** : elles peuvent être parcourues par une boucle *for*.

Construction d'un *dict*

- Un tableau associatif peut être créé vide, soit par une syntaxe `{ }`, soit par un appel au constructeur `dict()` :

```
d1 = {}  
d2 = dict()
```

- Un tableau associatif peut être créé en donnant des paires clé / valeur entre `{ }`, ou par un appel au constructeur :

```
d3 = {'x' : 3.2, 'y' : 2.5, 'z' : -0.2}  
d4 = dict(nom = 'toto', age = 25)
```

```
>>> d3  
{'x': 3.2, 'y': 2.5, 'z': -0.2}  
>>> d4  
{'nom': 'toto', 'age': 25}
```

Construction d'un *dict*

- Un tableau associatif peut être créé à partir d'une liste de tuples (un tuple représente une paire clé / valeur) passée en paramètre au constructeur :

```
>>> liste = [("nom", "Dupont"), ("prenom", "Antoine"), ("age", 28)]
>>>
>>> d = dict(liste)
>>> d
{'nom': 'Dupont', 'prenom': 'Antoine', 'age': 28}
```

- Un tableau associatif peut être créé *en compréhension* :

```
>>> d = {x : x*x for x in range(1, 6)}
>>> d
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> d[3]
9
```

Le type dict

Dans un tableau associatif, on doit toujours utiliser une clé valide, sinon une erreur est déclenchée ;

```
>>> mon_dico = {'écran' : 'screen', 'souris' : 'mouse'}
>>> mon_dico['écran']
'screen'
>>> mon_dico['ecran']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'ecran'
>>> mon_dico[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Le type dict

- On peut ajouter une paire clé / valeur à un dictionnaire, ou encore remplacer la valeur associée à une clé, par :

`<dictionnaire>[<cle>] = <valeur>`

```
>>> mon_dico
{'écran': 'screen', 'souris': 'mouse'}
>>> mon_dico['clavier'] = 'keyboard'
>>> mon_dico
{'écran': 'screen', 'souris': 'mouse', 'clavier': 'keyboard'}
```

- On peut aussi supprimer une paire clé / valeur :

`del <dictionnaire>[<cle>]`

```
>>> del mon_dico['écran']
>>> mon_dico
{'souris': 'mouse', 'clavier': 'keyboard'}
```

Le type dict

- La fonction `len` retourne le nombre de paires clé/valeur ;

```
>>> d = {"x" : 3.1, "y" : 4.2, "z" : -2.1, "t" : 7}
>>> len(d)
4
>>> del d["t"]
>>> len(d)
3
```

Le type dict

Exemple d'utilisation d'un dictionnaire ou tableau associatif

```
>>> stock = {'bananes' : 14, 'pommes' : 27, 'poires' : 12}
>>> stock['poires']
12
>>> stock['poires'] = 15
>>> stock
{'bananes': 14, 'pommes': 27, 'poires': 15}
>>> stock['prunes'] = 22
>>> stock
{'bananes': 14, 'pommes': 27, 'poires': 15, 'prunes': 22}
>>> del stock['bananes']
>>> stock
{'pommes': 27, 'poires': 15, 'prunes': 22}
>>> len(stock)
3
```

Quelques méthodes et fonctions

La méthode `items()` retourne une séquence formée des paires clé/valeur du dictionnaire. Ces paires sont données sous forme de tuples (`<cle>`, `<valeur>`);

```
>>> stock = {'prunes' : 11, 'pommes' : 24, 'poires' : 17}
>>> stock.items()
dict_items([('prunes', 11), ('pommes', 24), ('poires', 17)])
>>> for t in stock.items() :
    print(t)

('prunes', 11)
('pommes', 24)
('poires', 17)
>>> for cle, valeur in stock.items() :
    print("pour la clé", cle, "la valeur est", valeur)

pour la clé prunes la valeur est 11
pour la clé pommes la valeur est 24
pour la clé poires la valeur est 17
>>> list(stock.items())
[('prunes', 11), ('pommes', 24), ('poires', 17)]
>>>
```

Quelques méthodes et fonctions

La méthode `keys()` retourne une séquence formée des clés du dictionnaire ;

```
>>> stock = {'prunes' : 11, 'pommes' : 24, 'poires' : 17}
>>> stock.keys()
dict_keys(['prunes', 'pommes', 'poires'])
>>> for cle in stock.keys() :
    print(cle)

prunes
pommes
poires
>>> stock['pêches'] = 7
>>> len(stock.keys())
4
>>> list(stock.keys())
['prunes', 'pommes', 'poires', 'pêches']
```

Quelques méthodes et fonctions

La méthode *values()* retourne une séquence formée des valeurs du dictionnaire ;

```
>>> stock
{'prunes': 11, 'pommes': 24, 'poires': 17, 'pêches': 7}
>>> stock.values()
dict_values([11, 24, 17, 7])
>>> for valeur in stock.values() :
    print("valeur =", valeur)

valeur = 11
valeur = 24
valeur = 17
valeur = 7
>>> list(stock.values())
[11, 24, 17, 7]
```

Quelques méthodes et fonctions

Si la clé demandée n'est pas valide (ou encore n'est pas dans `keys()`), alors `<dictionnaire>[<cle>]` retourne une erreur.

Pour échapper à cette erreur, on peut utiliser la méthode `get` : Elle prévoit une valeur par défaut si la clé demandée n'est pas valide. Dans l'exemple, elle améliore donc `stock[article]`.

```
>>> stock
{'prunes': 11, 'pommes': 24, 'poires': 17, 'pêches': 7}
>>> commande = {'poires' : 3, 'pommes' : 2, 'pêches' : 3, 'bananes' : 2}
>>> for article in commande.keys() :
    print("pour l'article", article, ", qté en stock =", stock[article])

pour l'article poires , qté en stock = 17
pour l'article pommes , qté en stock = 24
pour l'article pêches , qté en stock = 7
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'bananes'
>>> for article in commande.keys() :
    print("pour l'article", article, ", qté en stock =", stock.get(article, 0))

pour l'article poires , qté en stock = 17
pour l'article pommes , qté en stock = 24
pour l'article pêches , qté en stock = 7
pour l'article bananes , qté en stock = 0
```

Quelques méthodes et fonctions

La méthode `clear()` est à employer avec prudence : elle vide le tableau associatif

```
>>> stock
{'bananes': 7, 'pommes': 27, 'poires': 12}
>>> stock.clear()
>>> stock
{}
```

Attention aux références !

Recopier un dictionnaire n'est pas sans risque. Comme tout objet complexe, il est à manipuler en connaissance de cause !

```
>>> stock = {'bananes' : 14, 'pommes' : 27, 'poires' : 12}
>>> stock2 = stock
>>> stock2
{'bananes': 14, 'pommes': 27, 'poires': 12}
>>> stock2['melons'] = 10
>>> stock2
{'bananes': 14, 'pommes': 27, 'poires': 12, 'melons': 10}
>>> stock
{'bananes': 14, 'pommes': 27, 'poires': 12, 'melons': 10}
>>> stock2['bananes'] = 19
>>> stock
{'bananes': 19, 'pommes': 27, 'poires': 12, 'melons': 10}
>>>
```

Attention aux références !

Pour ne recopier que les paires clé/valeur sans recopier la référence, on utilise la méthode adaptée *copy()* :

```
>>> stock = {'bananes' : 14, 'pommes' : 27, 'poires' : 12}
>>> stock2 = stock.copy()
>>> stock2
{'bananes': 14, 'pommes': 27, 'poires': 12}
>>> print("stock :", id(stock), ", stock2 :", id(stock2))
stock : 131285656453696 , stock2 : 131285656411904
>>> stock2['melons'] = 10
>>> stock2
{'bananes': 14, 'pommes': 27, 'poires': 12, 'melons': 10}
>>> stock
{'bananes': 14, 'pommes': 27, 'poires': 12}
>>> stock2.clear()
>>> stock2
{}
>>> stock
{'bananes': 14, 'pommes': 27, 'poires': 12}
```

Particularités

- Le parcours "naïf" d'un tableau associatif ne se fait pas sur les items mais sur les clés :

```
>>> stock
{'bananes': 14, 'pommes': 27, 'poires': 12}
>>> for element in stock :
    print(element)

bananes
pommes
poires
```

- Le parcours complet se fait plutôt par les items

```
>>> for cle, valeur in stock.items() :
    print("clé", cle, "... valeur", valeur)

clé bananes ... valeur 14
clé pommes ... valeur 27
clé poires ... valeur 12
```

Particularités

- L'opérateur d'appartenance *in* permet donc de savoir si un élément fait partie des clés d'un tableau associatif

```
>>> stock
{'pommes': 27, 'poires': 15, 'prunes': 22, 'bananes': 7}
>>> 'melons' in stock
False
>>> 'pommes' in stock
True
```

Bilan sur les tableaux associatifs

- Comme une liste, un dictionnaire (tableau associatif) est une structure muable, itérable ;
- Ce n'est pas "vraiment" une structure ordonnée, comme le sont les listes, tuples et chaînes de caractères : on n'a pas recours aux indices.
- Comme une liste, un tableau associatif est une structure adaptée aux situations où les données ne sont pas figées (elles peuvent être supprimées, ajoutées) ;
- Alors que les listes sont astreintes à un système d'indices entiers (0, 1, 2, ...), les tableaux associatifs ont des clés plus adaptables (les clés n'ont aucun lien entre elles, peuvent être des entiers ou des chaînes de caractères) ;

Les ensembles

Python propose une autre structure de données complexe : les ensembles (type *set*).

- Un **ensemble** est un groupement non ordonné d'éléments uniques : ni répétition, ni ordre.

C'est l'équivalent de l'ensemble au sens mathématique ;

- Il se représente (comme les tableaux associatifs) entre **accolades**, les éléments sont séparés par une virgule ;

```
>>> ens_1 = {5, 7, 14, 12, 9}
```

- Le nombre d'éléments de l'ensemble est donné par la fonction *len* ;

Les ensembles

- Un ensemble n'a ni indice, ni clé. Juste des valeurs ;
- Les accolades servent déjà aux tableaux associatifs. Il y a donc ambiguïté pour définir un ensemble vide.
 - ▶ { } construit un dictionnaire vide ;
 - ▶ on construit un ensemble vide par le constructeur set().

```
>>> ens_vide = set()
```

- Comme un ensemble n'a ni indice ni clé, il est impossible d'accéder à ses éléments par l'opérateur [].

Les ensembles

- L'opérateur d'appartenance *in*, et son contraire *not in*, disent si un élément est (ou non) dans un ensemble ;

```
>>> ens_1 = {5, 7, 14, 12, 9}
>>> 5 in ens_1
True
>>> 14 not in ens_1
False
```

- Même si on ne peut pas accéder aux éléments par [], on peut, comme pour tout itérable, parcourir un ensemble avec une boucle *for* et l'opérateur *in* ;

```
>>> for elt in ens_1 :
        print(elt, end = ', ')

5, 7, 9, 12, 14,
```

Le constructeur set()

Le constructeur `set()` permet, selon le paramètre qui lui est passé, de créer un ensemble à partir d'un tuple, d'une liste ou d'une chaîne de caractères ;

```
>>> chaîne = "ressasser"  
>>> ens_chaîne = set(chaîne)  
>>> ens_chaîne  
{ 's', 'r', 'a', 'e' }
```

```
>>> t = (4, 1, 2, 1, 3, 4)  
>>> ens_t = set(t)  
>>> ens_t  
{ 1, 2, 3, 4 }
```

```
>>> l = ['a', 'b', 'a', 'd', 'c']  
>>> ens_l = set(l)  
>>> ens_l  
{ 'c', 'd', 'b', 'a' }
```

Comparer des ensembles

On peut utiliser les comparateurs classiques entre *set* :

- L'opérateur `<` pour l'inclusion stricte ;
- Les opérateurs `<=`, `>`, `>=` (inclusion large, inclusions stricte et large inversées) ;

```
>>> set_1 = {'a', 'r', 'e', 's'}
>>> set_2 = {'s', 'e', 't'}
>>> set_3 = {'s', 'e'}
>>> set_4 = {'e', 's'}
>>> set_3 < set_1
True
>>> set_2 < set_1
False
```

```
>>> set_3 < set_4
False
>>> set_4 >= set_3
True
>>> set_2 > set_3
True
```

Comparateurs == et !=

- Deux ensembles sont égaux (au sens de "==") s'ils ont exactement les mêmes éléments.
- Par contre, ils peuvent être égaux sans avoir le même identifiant (== n'est pas *is*).

```
>>> set_1 = {'a', 'r', 'e', 's'}
>>> set_2 = {'s', 'e', 't'}
>>> set_3 = {'s', 'e'}
>>> set_4 = {'e', 's'}
>>> set_3 == set_4
True
>>> set_1 != set_2
True
>>> set_2 == set_3
False
>>> id(set_3)
131285656232320
>>> id(set_4)
131285656231872
```

Quelques méthodes et fonctions

Quand un ensemble ne contient que des éléments comparables, les fonctions *min* et *max* s'appliquent à l'ensemble.

```
>>> ens = {4, 5, 3, 7, 2}
>>> min(ens)
2
>>> max(ens)
7
>>> ens2 = {"algorithme", "programmation", True, 5}
>>> min(ens2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Quelques méthodes et fonctions

- La méthode *add* permet d'ajouter un élément à un ensemble (pas d'ajout si l'élément est déjà présent) ;
- La méthode *remove* permet d'ajouter un élément à un ensemble (erreur si l'élément à retirer n'est pas présent) ;

```
>>> ens = {'a', 'e', 'i', 'u', 'y', 'w'}
>>> ens.add('o')
>>> ens
{'u', 'a', 'w', 'o', 'e', 'i', 'y'}
>>> ens.remove('w')
>>> ens
{'u', 'a', 'o', 'e', 'i', 'y'}
>>> ens.add('a')
>>> ens
{'u', 'a', 'o', 'e', 'i', 'y'}
>>> ens.remove('b')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
```

Quelques méthodes et fonctions

- La méthode *clear*, supprime tous les éléments de l'ensemble, qui devient un ensemble vide ;
- La méthode *copy* permet de réaliser la copie élément par élément d'un ensemble (sans copier la référence) ;

```
>>> s1 = {'a', 'b', 'c', 'd'}
>>> s2 = s1
>>> s3 = s1.copy()
>>> s1.add('e')
>>> s2
{'a', 'c', 'e', 'd', 'b'}
>>> s3
{'c', 'a', 'd', 'b'}
>>> s1.clear()
>>> s2
set()
>>> s3
{'c', 'a', 'd', 'b'}
```

Union

Les ensembles (set) python ont à leur disposition des méthodes spécifiques qui s'apparentent aux opérations mathématiques ensemblistes. L'**union** en est une.

- L'union de deux ensembles crée un nouvel ensemble regroupant tous les éléments des deux ensembles ;
- L'union s'invoque par la méthode *union* ou aussi par l'opérateur "|";

```
>>> s1 = {1, 2, 4, 7, 8}
>>> s2 = {7, 5, 2, 9, 4, 6}
>>> s3 = s1.union(s2)
>>> s3
{1, 2, 4, 5, 6, 7, 8, 9}
>>> s4 = s1 | s2
>>> s4
{1, 2, 4, 5, 6, 7, 8, 9}
```

Union

Les ensembles (set) python ont à leur disposition des méthodes spécifiques qui s'apparentent aux opérations mathématiques ensemblistes. L'**union** en est une.

- L'opérateur `|` est commutatif : `s1 | s2` et `s2 | s1` sont deux ensembles égaux (ils ont les mêmes éléments) ;
- `s1.union(s2)` et `s2.union(s1)` sont donc aussi égaux ;
- Si $s1 \leq s2$ (inclusion large), alors `s1 | s2` est égal à `s2` !
- En général, $\text{len}(s1 | s2) \leq \text{len}(s1) + \text{len}(s2)$;
- $\text{len}(s1 | s2)$ est égal à $\text{len}(s1) + \text{len}(s2)$ seulement si `s1` et `s2` sont "disjoints" (d'intersection vide) ;

Intersection

Une autre méthode ensembliste : l'**intersection**.

- L'intersection de deux ensembles crée un nouvel ensemble regroupant les éléments **communs** aux deux ensembles ;
- L'intersection s'invoque par la méthode *intersection* ou par l'opérateur **&** ;

```
>>> s1 = {1, 2, 4, 7, 8}
>>> s2 = {7, 5, 2, 9, 4, 6}
>>> s3 = s1.intersection(s2)
>>> s4 = s1 & s2
>>> s3
{2, 4, 7}
>>> s4
{2, 4, 7}
```

Intersection

Une autre méthode ensembliste : l'**intersection**.

- L'opérateur $\&$ est commutatif : $s1 \& s2$ et $s2 \& s1$ sont deux ensembles égaux (ils ont les mêmes éléments) ;
- $s1.intersection(s2)$ et $s2.intersection(s1)$ sont égaux ;
- Si $s1 \leq s2$ (inclusion large), alors $s1 \& s2$ est égal à $s1$!
- $len(s1 \& s2) \leq \min(len(s1), len(s2))$;
- $len(s1 \& s2)$ est égal à 0 seulement si $s1$ et $s2$ sont "disjoints" (d'intersection vide) ;

Différence

Une autre méthode ensembliste : la **différence**.

- La différence entre un ensemble `s1` et un ensemble `s2` (pris dans cet ordre) crée un nouvel ensemble regroupant les éléments de `s1` qui ne sont pas dans `s2` ;
- La différence s'invoque par la méthode *difference* ou par l'opérateur `-` ;

```
>>> s1 = {1, 2, 4, 7, 8}
>>> s2 = {7, 5, 2, 9, 4, 6}
>>> s3 = s1.difference(s2)
>>> s4 = s1 - s2
>>> s3
{8, 1}
>>> s4
{8, 1}
```

Différence

Une autre méthode ensembliste : la **différence**.

- Attention ! L'opérateur - n'est pas commutatif : $s1 - s2$ et $s2 - s1$ sont en général différents !
- Si $s1 \leq s2$ (inclusion large), alors $s1 - s2$ est vide !
- Si $s1$ et $s2$ sont disjoints alors $s1 - s2$ est égal à $s1$;

Bilan sur les ensembles

- Un ensemble est une structure muable, itérable ;
- Ce n'est pas une structure ordonnée ;
- Il n'y a aucune répétition d'éléments ;
- Il n'y a ni clé ni indice pour accéder à un élément particulier ;
- Un ensemble est adapté aux situations où les données ne sont pas figées, mais demeurent uniques ;
- Un ensemble n'est pas adapté s'il faut accéder à un élément particulier (dans ce cas, listes, tuples ou dictionnaires sont préférables) ;