

Dictionnaires et Ensembles



Les ensembles

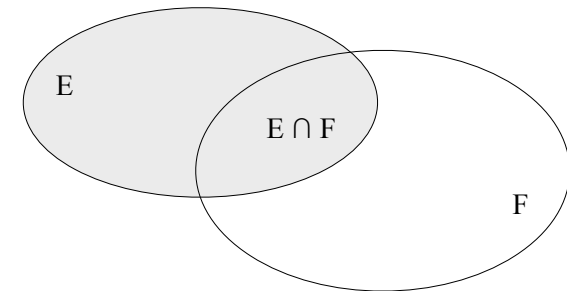


Diagramme de Venn

Origine :
Georg Cantor (1845-1918),
mathématicien allemand.

Où en sommes-nous des types de données ?

- Quels types de données avons-nous déjà rencontrés ?
- Quelques *types simples* : int, float, bool

```
>>> type(-45)
<class 'int'>
```

```
>>> type(23.7)
<class 'float'>
```

```
>>> type(False)
<class 'bool'>
```

- Quelques *types composés*, suites numérotées d'objets. Nous avons vu trois sortes de **séquences** : str, list, tuple

```
>>> type('foo')
<class 'str'>
```

```
>>> type([2,6,1])
<class 'list'>
```

```
>>> type((2,6,1))
<class 'tuple'>
```

- Nous allons découvrir deux nouveaux types composés, les **ensembles** et les **dictionnaires** : set, dict

```
>>> type({2,6,1})
<class 'set'>
```

```
>>> type({'prix':50, 'nom':'Eska'})
<class 'dict'>
```

Qu'est-ce qu'un ensemble ?

- Un **ensemble** en Python est une collection d'objets *sans répétition* et *sans ordre* (donc *sans numérotage*). Ce n'est PAS une séquence !
- On les note comme en maths avec des accolades {...}. Les éléments sont de types quelconques. Exemples :
- L'ensemble vide se note set() et non {} qui crée un dictionnaire !

{5,3,-8,2} {'o','e','y','u','a','i'} {5,'foo',(3,-2),7.4}

- Un ensemble est défini à l'ordre près :

```
>>> {3,1,2} == {2,3,1}
True
>>> {3,2,3,2} == {2,3}
True
```

- Un ensemble paraît trié en interne mais c'est uniquement pour accélérer la recherche d'un élément :

```
>>> e = {3,'b',2,'a',5}
>>> e
{2,3,'a',5,'b'}
```

Accès aux éléments d'un ensemble

- Les éléments d'un ensemble ne sont pas numérotés. On ne peut pas utiliser une notation comme `e[i]` puisque parler de l'élément numéro `i` n'a pas de sens !
- L'opérateur `in` permet de savoir si un objet **appartient** à un ensemble.
- L'opération `E < F` permet de tester si l'ensemble `E` est **strictement inclus** dans l'ensemble `F`.

```
>>> {2,4} < {1,2,3,4}
True
```

- Axiome du choix : il est possible d'obtenir un élément (lequel ?) d'un ensemble `E` et de le supprimer en même temps avec la méthode `pop()`.

```
>>> E = {5,2,3,1}
>>> x = E.pop()
>>> (x, E)
(1, {2,3,5})
```

donc un ensemble est mutable !

Comment construire un ensemble ?

- En extension :

```
>>> E = {5,2,3,1}
>>> E
{1, 2, 3, 5}
```

- En compréhension :

```
>>> E = {x*x for x in range(20) if x % 3 == 0}
>>> E
{0, 225, 36, 9, 144, 81, 324}
```

- Avec la fonction `set(...)` de la classe `set` :

```
>>> E = set('aeiouy')
>>> E
{'o', 'i', 'e', 'a', 'y', 'u'}

>>> E = set([5,2,5,6,2])
>>> E
{2, 5, 6}
```

- En ajoutant un élément à un ensemble `E` avec la méthode `E.add(x)`

```
>>> E = {5,3,2,1}
>>> E.add(8)

>>> E
{8,1,2,3,5} mutation !
```

Le nombre d'éléments d'un ensemble

- On parle du *cardinal* d'un ensemble (fini). Il s'agit de la fonction `len`.

```
>>> E = {5,2,3,1}
>>> len(E)
4
```

- Si elle n'existait pas, on pourrait la programmer. Notons au passage qu'un ensemble est un objet itérable.

```
def cardinal(E) :
    res = 0
    while E != set() :
        x = E.pop()
        res = res + 1
    return res
```

```
def cardinal(E) :
    res = 0
    for x in E :
        res = res + 1
    return res
```

```
def cardinal(E) :
    return sum(1 for x in E)
```

Opérations sur les ensembles

- L'**ensemble vide** se note `set()`. Définissons-le : `empty = set()`

- La **réunion** $E \cup F = \{x : x \in E \text{ ou } x \in F\}$ se note `E | F` en Python.

```
>>> {3,2,5,4} | {1,7,2,5}
{1, 2, 3, 4, 5, 7}
```

- L'**intersection** $E \cap F = \{x : x \in E \text{ et } x \in F\}$ se note `E & F` en Python.

```
>>> {3,2,5,4} & {1,7,2,5}
{2, 5}
```

- La **différence** $E - F = \{x : x \in E \text{ et } x \notin F\}$ se note `E - F` en Python.

```
>>> {3,2,5,4} - {1,7,2,5}
{3, 4}
```

- Ne pas confondre `E - {x}` qui construit un nouvel ensemble, avec la méthode `E.remove(x)` qui supprime `x` de l'ensemble `E`.

- Nous avons déjà vu la construction d'un ensemble par **compréhension** :

```
>>> E = {x*x for x in range(20) if x % 3 == 0}
>>> E
{0, 225, 36, 9, 144, 81, 324}
```

- La primitive `filter(function, iterable)` permet de parcourir un objet itérable (str, list, tuple, range, set, dict) et d'obtenir un itérateur sur les objets retenus. Quitte à le parcourir, à le transformer en liste, en ensemble, etc. Attention, une fois utilisé, il est épuisé ! Style de programmation très pythoniste...

```
>>> F = filter(lambda x : x % 2 == 0, {3,2,5,4,1,6,9,0})
>>> F
<filter object at 0x102f46f50>
>>> set(F)          # ou bien : for x in F : ...
{0, 2, 4, 6}
>>> set(F)          # F a été épuisé par une itération !
set()
```

One of the exercises on our Python course asks the user to take the following tuple:

```
m1en = (31,28,31,30,31,30,31,31,30,31,30,31)
```

and produce two lists with 365 elements each. The first list is to contain the number 1 31 times, then the number 2 28 times, and so on - so that a user of the list can look up a day number of the year and (in a single reference to the list) know what month that day is in. A second list is also produced, containing 1 through 31, then 1 through 28, then 1 through 31, and so on - thus giving the ability to look up any day number of the year and tell you which day number it is within a month.

Les ensembles gelés (frozenset)

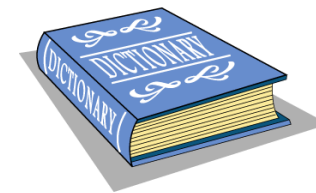
```
>>> {(1,2),(3,4)}      # un ensemble de tuples
{(1, 2), (3, 4)}
>>> E = {[1,2],[3,4]}  # un ensemble de listes ?
TypeError: unhashable type: 'list'
```

- Un ensemble construit par `set(...)` est mutable : axiome du choix. Mais ses éléments ne peuvent pas être mutables. On peut faire des ensembles de tuples (par exemple des ensembles de points du plan) mais pas d'ensembles dont les éléments sont des listes ou des ensembles !

- Pour faire des ensembles de listes ou d'ensembles, il faut utiliser le constructeur `frozenset(...)`. Il n'est PAS AU PROGRAMME de L1.



Les dictionnaires



Qu'est-ce qu'un dictionnaire ?

• Un **dictionnaire** est une collection non numérotée de couples var:val où var est un objet *non mutable* (la **clé**) et où val est n'importe quelle valeur. Toutes les clés doivent être **distinctes** !

```
>>> stock = {'poires':51, 'pommes':243}
>>> stock
{'pommes': 243, 'poires': 51}          # aucun ordre !
```

```
>>> len(stock)    >>> 'pommes' in stock    >>> stock['pommes']
2                True                      243
```

• Le dictionnaire vide se note {} ou dict().

• On peut modifier la valeur associée à une clé, ou rajouter un nouveau couple clé/valeur :

```
>>> stock['pommes'] = 100
>>> stock['bananes'] = 18    =>    >>> len(stock)
                               3
```

Accès aux éléments d'un dictionnaire

• On peut TRES VITE **accéder à la valeur associée à une clé**. C'est le principal intérêt des dictionnaires (techniquement : des *tables de hash-code*). La recherche est pratiquement instantanée !

```
>>> stock['pommes']    >>> stock[243]
243                    KeyError: 243  pas une clé !
```

• La recherche est donc **unidirectionnelle**, comme dans un dictionnaire Français-->Anglais. On va de la clé vers la valeur !

• La **clé doit être non mutable**. Donc essentiellement des chaînes et des nombres mais pas de listes. On peut utiliser des tuples comme clés à condition qu'ils ne contiennent aucun objet mutable.

```
>>> dico = {1:'one', 2:'two', 3:'three'}
>>> dico[2]
'two'
```

Clé inexistante : exception ?

• Si l'on demande la valeur associée à une clé inexistante, une exception **KeyError** est levée. On peut l'attraper au vol.

```
>>> x = dico[8]
KeyError: 8
```

```
try :
    x = dico[8]
except KeyError :
    x = 'inconnu'
```

• Il est aussi possible de demander si la clé existe, avec l'opérateur in :

```
>>> 2 in dico
True
```

```
>>> 8 not in dico
True
```

Comment construire un dictionnaire ?

• En extension : on fournit tous les couples, dans un ordre quelconque.

```
dico = {1:'one', 2:'two', 3:'three'}
```

• Un dictionnaire est **mutable**. La méthode update(d) permet d'ajouter un dictionnaire d à un dictionnaire existant.

```
>>> dico.update({4:'four', 5:'five'})
>>> dico
{1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
```

• La méthode pop(clé) retourne la valeur *val* associée à la clé et supprime le couple *clé:val* du dictionnaire.

```
>>> dico.pop(3)    # ne pas confondre avec dico[3]
'three'
>>> dico
{1: 'one', 2: 'two', 4: 'four', 5: 'five'}
```

Voir les clés et les valeurs

- On peut vouloir obtenir la liste de toutes les clés ou bien la liste de toutes les valeurs. Les méthodes `keys()` et `values()` retournent des **vues** (views) sur les clés et les valeurs.

```
>>> dico.keys()
dict_keys([1, 2, 4, 5])
>>> dico.values()
dict_values(['one', 'two', 'four', 'five'])
```

des objets bizarres...

- Ces vues sont des objets itérables que l'on peut parcourir avec une boucle `for`, ou transformer en listes/ensembles.

```
cpt = 0
for k in dico.keys():
    if k % 2 == 0:
        cpt = cpt + 1

>>> list(dico.keys())
[1, 2, 4, 5]
>>> set(dico.values())
{'five', 'one', 'four', 'two'}
```

```
sum(k for k in dico.keys() if k % 2 == 0)
```

Itérer dans un dictionnaire

- Un dictionnaire étant un *objet itérable*, on peut... itérer dessus !

```
>>> dico
{1:'one', 2:'two', 4:'four', 5:'five'}
```

- Par défaut, on itère sur les clés :

```
cpt = 0
for k in dico:
    if k % 2 == 0:
        cpt = cpt + 1
```

⇔

```
cpt = 0
for k in dico.keys():
    if k % 2 == 0:
        cpt = cpt + 1
```

⇕

```
cpt = sum(1 for k in dico if k % 2 == 0)
```

>>> cpt
2

- Mais on peut aussi itérer sur les valeurs :

```
for v in dico.values():
    if len(v) == 3: print(v, end=' ')
```

Application : une mémo-fonction

- Programmer une fonction qui se souvient des calculs déjà effectués !
- Exemple de la factorielle. Je calcule $100!$ qui demande 99 multiplications. Je calcule ensuite $101!$ qui demande 100 multiplications au lieu d'une seule puisque $101! = 100! * 101$. Je veux que ma fonction `fac(n)` se souvienne qu'elle a déjà calculé $100!$. Comment ?
- Nous allons gérer un **dictionnaire privé** à la fonction qui va contenir tous les couples `n:v` tels que `fac(n) == v` ait déjà été calculé !

Pour calculer `n!` :

- si `n` est une clé du dictionnaire, fini.
- sinon :
calculer `v = (n-1)! * n` ; stocker `n:v` ; retourner `v`

- On dit que le dictionnaire joue le rôle de **mémoire cache**.

```
def memo_fac():
    h = dict()
    h[0] = 1
    def calcul(n):
        if n < 0:
            print('la table est de longueur', len(h))
            return
        try:
            essai = h[n]
        except KeyError:
            essai = calcul(n-1) * n
            h[n] = essai
        return essai
    return calcul
```

retourne une mémo-fonction
la mémoire cache
je mémorise 0! = 1
une fonction interne
juste pour le fun...
déjà calculé ?
non, je calcule !
et je le stocke
et je le rends en résultat
je rends une fonction !

```
>>> fac = memo_fac()
>>> fac(50)
30414093201713378043612608166...000000
>>> fac(-1)
'la table est de longueur 101'
```

```
>>> fac(20)
2432902008176640000
immédiat !
```