

# Collections : tuples, dictionnaires et ensembles

Polytech Marseille

Séverine Dubuisson, Simon Vilmin

[severine.dubuisson@univ-amu.fr](mailto:severine.dubuisson@univ-amu.fr),

[simon.vilmin@univ-amu.fr](mailto:simon.vilmin@univ-amu.fr)

2024 - 2025

**amU** Aix  
Marseille  
Université



# Plan

## Dictionnaires

- Définition

- Opérations

- Résumé

## Tuples

- Définition

- Opérations

- Remarques

- Résumé

## Ensembles

- Définition

- Opérations

- Résumé

Collections : le mot de la fin

## Remarque préliminaire

**i Remarque :** le plan et le nombre de slides sont *épuisants* mais ...

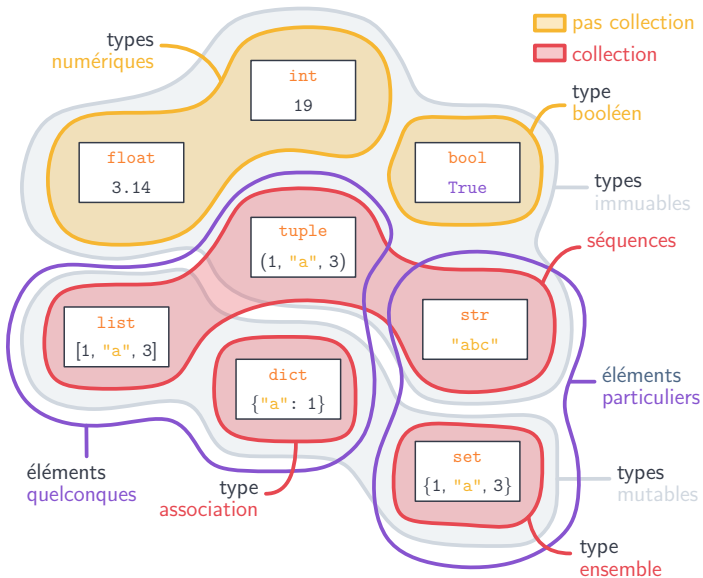
- cours *conceptuellement moins chargé* que celui sur les listes
- portrait *général* des autres collections

**! Attention :** beaucoup de ce qu'on a vu dans les listes *s'applique aux autres collections* en fonction de leurs propriétés !

- parcours, modifications, fonctions
- les subtilités sur la mémoire et les pointeurs, ...

**paw Astuce :** avec la *doc*, l'aide (*help*) et les différences mutable/immutable, séquences, etc, vous avez *tous les outils pour essayer et voir quoi marche quand* !

## Dans l'épisode précédent



# Listes

 **Définition :** une *liste* (type `list`) est une *séquence mutable* d'objets *quelconques*.

Accès, parcours, recherche :

- indexation, slicing
- parcours avec `range`, `in`, `enumerate`

Modification, ajout, suppression :

- ajout avec `append`, `insert`, `extend`, `+=`, ...
- modification avec indexation, slicing, `sort`, `reverse`, ...
- suppression avec `clear`, `pop`, `del`, ...

Opérations entre listes :

- concaténation (+) et répétition (\*)

# Dictionnaires

## Dictionnaires

- Définition

- Opérations


- Résumé

## Tuples

## Ensembles


## Collections : le mot de la fin

# Définition

 **Définition** : un *dictionnaire* (**dict**) est une *collection mutable* de *paires* ou *items* (clé, valeur) où

- *clé* est un objet *hashable*
- *valeur* est un objet *quelconque*

toutes les clés d'un dictionnaire sont *uniques*

 **Syntaxe** : un dictionnaire est défini avec des accolades « {} », on y sépare les éléments par des virgules « , », une clé est séparée de sa valeur par deux points « : »

```
dico = {  
    cle1: valeur1,  
    cle2: valeur2,  
    ...  
}
```

# Exemples

## Syntaxe

```
>>> dico = {}      # dict vide
>>> dico = dict()  # dict vide
>>> dico = {'a': 1, 'b': 2}
>>> dico = {
    (0, 0): "origine",
    "prenom": "Iris",
    "age": 27
}
```

## Hashabilité et unicité des clés

```
>>> dico = {[0] : 1}
TypeError: unhashable type: 'list'
>>> dico = {0: "a", 0: "b"}
>>> dico
{0: 'b'}
```



# Pourquoi faire ?

Listes :

- jouent le rôle de *tableau* de valeurs
- on accède à une valeur par sa *position*

Dictionnaires :

- permet de « stocker » des données accessibles par un « identifiant »
- permet de représenter des fonctions (au sens maths) avec des items de la forme  $x: f(x)$
- implémentation des *tableaux associatifs* : à une clé on associe une valeur



**Remarque :** du coup, *insérer* et *accéder à une valeur* dans un dictionnaire est très rapide (en termes de complexité)

« principe » dans un dictionnaire : *tout passe par les clés*

## Exemple : dictionnaire de synonymes

```
synonymes = {  
    "flemme": ["faineantise", "flemmardise", "paresse"],  
    "radin": ["avare", "grigou", "pingre", "rapiat", "rat"],  
    "oseille": ["flouze", "pognon", "peze", "fric", "ble"],  
    ...  
}  
  
synonymes["flemme"] # liste des synonymes de flemme
```

## Exemple : livres

```
livres = {  
  2070300951: {  
    "titre": "Capitale de la Douleur",  
    "auteur.e": "Eluard",  
    "annee": 1926},  
  2207260380: {  
    "titre": "Il est difficile d'etre un dieu",  
    "auteur.e": ["Strougatski", "Strougatski"],  
    "annee": 1964},  
  ...  
}  
  
livres[2207260380]["titre"] # titre du livre avec ISBN 220...
```

## Exemple : Unicode


```
unicode = {  
    'A': 65,  
    ...  
    'a': 97,  
    'b': 98,  
}  
  
unicode['A']
```

**i Remarque :** en Python, la fonction `ord()` associe à un caractère son code Unicode

## Exemple : associer à un mot sa taille

```
>>> phrase = ["un", "ours", "fait", "ses", "courses"]
>>> taille = {}
>>> for mot in phrase:
...     taille[mot] = len(mot)
>>> taille
{"un": 2, "ours": 4, "fait": 4, "ses": 3, "courses": 7}
```

# Table de hachage

 **Question :** comment ça marche, un *dictionnaire*?

 **Réponse :** on utilise les *tables de hachage*

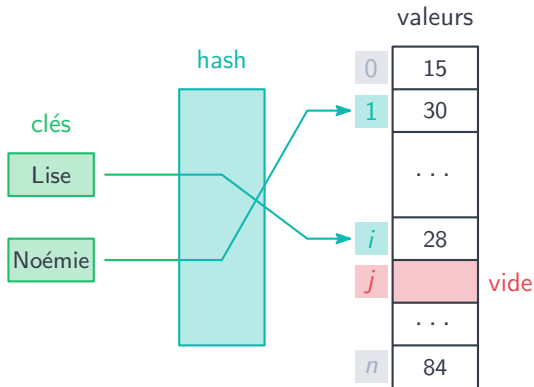
Objectif d'une table de hachage :

- remplacer l'accès aux valeurs d'un tableau via leur *position* par un *système plus flexible* de *clés*

Pour y arriver :

- on part d'un tableau
- on utilise une *fonction de hachage* qui sert à transformer une *clé* en l'*indice* où se trouve la valeur
- le tableau peut contenir beaucoup de *cases vides*

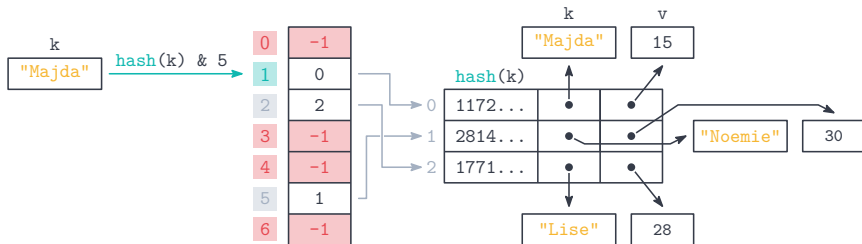
# Illustration



- à un prénom on associe un âge
- valeurs stockées dans un tableau (avec des cases potentiellement vides)
- une fonction de hash associe une clé  $k$  à la position où se trouve sa valeur  $v$

**!** **Attention :** représentation *abstraite* d'une table de hachage

## Les dict Python (simplifié)



- items (`hash(k)`, `k`, `v`) stockés *par ordre d'insertion* dans un tableau (droite)
- une table de hachage (gauche) associe à `k` la position de l'item correspondant dans le tableau de données



**Rappel :** CPython est *une implémentation* de Python *parmi d'autres*.  
D'autres implémentations pourraient utiliser d'autres structures de données



## Accès

On accède aux valeurs d'un dictionnaire par ses clés. On utilise les crochets [ ] ou la méthode `get`

```
>>> D = {"a": 1, "b": 3}
>>> D["a"]
1
>>> D.get("b")
3
```

**i Remarque :** si la clé `k` n'existe pas, `D[k]` donne une erreur, alors que `D.get(k)` renvoie `None`

```
>>> D = {"a": 1, "b": 3}
>>> D["c"]
KeyError: 'c'
>>> print(D.get("c"))
None
```

# Appartenance



**Rappel** : les dictionnaires sont pensés « clés »



**Syntaxe** : on teste l'existence d'une clé avec `in`

```
>>> animal = {"nom": "chien", "mammifere": True, "pattes": 4}
>>> "pattes" in animal
True
>>> "ecailles" in animal
False
>>> "chien" in animal
False
```



**Remarque** : si on voulait chercher une *valeur* plutôt qu'une *clé*, il faudrait utiliser `D.values()` qui renvoie une *vue* (pas une *copie*!) sur les valeurs du dictionnaire

## Parcours

```
# ==== Prog
D = {"a": 1, "b": 2, "c": 3}
for i in range(len(D)):
    print((i, D[i]), end=" ")
```

```
# ==== Resultat
KeyError
```

```
# ==== Prog
D = {"a": 1, "b": 2, "c": 3}
for k in D:
    print((k, D[k]), end=" ")
```

```
# ==== Resultat
('a', 1) ('b', 2) ('c', 3)
```

```
# ==== Prog
D = {"a": 1, "b": 2, "c": 3}
for (k, v) in D.items():
    print((k, v), end=" ")
```

```
# ==== Resultat
('a', 1) ('b', 2) ('c', 3)
```

```
# ==== Prog
D = {"a": 1, "b": 2, "c": 3}
for v in D.values():
    print(v, end=" ")
```

```
# ==== Resultat
1, 2, 3
```

# Parcours en résumé

## Parcours par les clés

```
for k in D:  
    print(k, D[k])  
  
for k in D.keys():  
    print(k, D[k])
```

## Parcours des valeurs

```
for v in D.values():  
    print(v)
```

## Parcours des items (clé, valeur)

```
for (k, v) in D.items():  
    print(k, v)
```

**i** **Remarque :** `keys()`, `values()` et `items()` ne *renvoient pas une copie* des éléments, mais *une vue*  $\Rightarrow$  coût en mémoire *faible* !

## Ajout d'un item

pour ajouter un item (k, valeur), on utilise directement les crochets

```
D[k] = valeur
```

**!** **Attention :** *différent des listes* où cette opération *ne fonctionne pas* !

```
>>> gateau = {}
>>> gateau["farine"] = 500
>>> gateau["oeufs"] = 4
>>> gateau["beurre"] = 7560
>>> gateau["sucre"] = 10
>>> gateau
{"farine": 500, "oeufs": 4, "beurre": 7560, "sucre": 10}
```

## Suppression d'un item

- `pop` *supprime* un item (cle, valeur) donnée par *clé* et renvoie *valeur*
- `popitem` *supprime* et *renvoie* le dernier item (cle, valeur) ajouté au dictionnaire
- `del` permet de supprimer un item (cle, valeur) à partir de la *clé*

```
>>> D = {"a": 1, "b": 2, "c": 3, "d": 4}
>>> a = D.pop("a")
>>> del D["b"]
>>> item = D.popitem()
>>> print(a, item, D)
1 ("d", 4) {"c": 3}
```

# Opérations

**!** **Attention** : pas de + ni de \* pour les dictionnaires !

Opération principale : *la mise à jour*

- $D3 = D1 \mid D2$  renvoie l'agrégation de D1 et D2 dans D3
- $D1 \mid= D2$  ou `D1.update(D2)` met à jour les valeurs de D1 avec D2.

**i** **Remarque** : si D1 et D2 ont des *clés en commun*, ce sont les valeurs de D2 qui sont gardées  $\Rightarrow D1 \mid D2$  est différent de  $D2 \mid D1$

```
>>> D1, D2 = {"a": 1, "c": 3}, {"a": 2, "b": 3}
>>> D1 |= D2
>>> D1
{"a": 2, "b": 3, "c": 3}
>>> D4 = D1 | {"c": 84, "d": 5}
>>> D4
{"a": 2, "b": 3, "c": 84, "d": 5}
```

## Définition en compréhension

**i Remarque :** Comme pour les listes, on peut définir un dictionnaire en *compréhension*

```
>>> {str(i): i for i in range(3)}  
{'0': 0, '1': 1, '2': 2}
```

```
>>> {m: len(m) for m in ["la", "patate", "au", "pate", "pateux"]}  
{"la": 2, "patate": 6, "au": 2, "pate": 4, "pateux": 6}
```

```
>>> {i: i**2 for i in range(10) if i % 3 != 0}  
{1: 1, 2: 4, 4: 16, 5: 25, 7: 49, 8: 64}
```



# Résumé



**Rappel :** un dictionnaire (`dict`) est une collection *mutable* d'items de la forme `cle: valeur`




## Attention :


- les clés doivent être *hashables* !
- le parcours, l'accès, la modification se font essentiellement *par les clés*

```
>>> film = {"titre": "solaris", "annee": 2002, "note10": 6.2}
>>> D["pays"] = "Etats-Unis"
>>> D.pop("note10")
6.2
```

## Exercice

 **Exercice** : Écrivez une fonction `invertKV` qui prend en paramètre un dictionnaire `D` et qui renvoie un dictionnaire `Di` qui contient « l'inverse » des items de `D`.

```
>>> D = {"a": 1, "b": 2}
>>> Di = invertKV(D)
>>> Di
{1: "a", 2: "b"}
```

 **Remarque** : on considère que les valeurs de `D` sont *toutes hashables* et que chaque valeur n'apparaît qu'une fois

## Exercice

⚙️ **Exercice** : la méthode `update` met à jour un dictionnaire D1 avec les données d'un dictionnaire D2. Si D2 et D1 ont des clés en commun, c'est la valeur de D2 qui est conservée.

```
>>> D1 = {"a": 1, "b": 2}
>>> D2 = {"a": "AAAAAAH"}
>>> D1.update(D2)
>>> D1
{"a": "AAAAAAH", "b": 2}
```

Écrivez une fonction `cautiousUpdate` qui prend en paramètres 2 dictionnaires D1, D2 et qui renvoie un dictionnaire D3 qui agrège D1 et D2. Si D1 et D2 ont une clé en commun, D3 gardera la liste des valeurs associées.

```
>>> D1 = {"a": 1, "b": 2}
>>> D2 = {"a": "AAAAAAH"}
>>> D3 = cautiousUpdate(D1, D2)
>>> D3
{"a": [1, "AAAAAAH"], "b": 2}
```

## Exercice

⚙️ **Exercice** : Écrivez une fonction `getLivrebyAuteure` qui prend en paramètre un dictionnaire de livres et qui construit le dictionnaire auteure où les items ont la forme

```
nom_auteure: listes des livres
```

Le dictionnaire de livres a des items de la forme

```
20700300951: {  
  "titre": "Capitale de la Douleur",  
  "auteur.e": "Eluard",  
  "annee": 1926,  
}
```

Et le dictionnaire de sortie aura des items de la forme

```
{  
  "Eluard": ["Capitale de la Douleur", "Poesie Ininterrompue",  
    "L'Amoureuse"]  
}
```

# Tuples

## Dictionnaires

## Tuples

- Définition

- Opérations

- Remarques

- Résumé

## Ensembles

## Collections : le mot de la fin



**Remarque :** en une phrase, *tuple* = liste immuable

# Définition

 **Définition** : un *tuple* (**tuple**) est une *séquence immuable* d'objets *quelconques*.

 **Syntaxe** : un tuple est défini avec des parenthèses « () », on y sépare les éléments par des virgules « , »


```
tuple = (element1, element2, ...)
```

```
t = ()          # tuple vide  
t = tuple()    # tuple vide  
t = ([1, 2], (3, 4), "citron")
```

## Tuples à 1 élément

```
# ==== Prog principal
t = (1)
print(t, type(t))


# ==== Resultat
1 <class 'int'>
```

 **Problème :** comment faire pour créer un tuple à 1 élément ?


```
# ==== Prog principal
t = (1,) # virgule sans rien derriere
print(t, type(t))

# ==== Resultat
(1,) <class 'tuple'>
```

## Définition en compréhension

 **Question :** peut-on définir un tuple en *compréhension*? (on aimerait bien en tout cas)

```
>>> T = (i for i in range(5))
>>> T
<generator object <genexpr> at 0x0000021ADFB61930>
>>> T = (i for i in range(5),)
SyntaxError: invalid syntax
```

 **Réponse :** on peut passer par une *liste* que l'on *convertit en tuple*

```
>>> T = tuple([i for i in range(5)])
>>> T
(0, 1, 2, 3, 4)
```



## Accès, appartenance

**i Remarque :** les tuples sont *des séquences*, on a accès aux *indices*

```
>>> T = ("je", "suis", "un", "tuple")
>>> T[2]
un
>>> T[0:2]
je suis
```

 **Syntaxe :** on peut tester l'appartenance avec `in`

```
>>> "suis" in T
True
>>> "serpent" in T
False
```

# Parcours

par les *indices*

```
T = ('k', 'i', 'w', 'i')
for i in range(len(T)):
    print(T[i], end=" ")
```

par les *valeurs*


```
T = ('k', 'i', 'w', 'i')
for e in T:
    print(e, end=" ")
```

par les *paires (indices, valeurs)*

```
T = ('k', 'i', 'w', 'i')
for (i, e) in enumerate(T):
    print((i, e), end=" ")
```

**i** Remarque : en bref, *pareil que les listes* quoi

# Modifications

 **Question** : est-ce qu'on peut faire des modifications type `append`, `insert`, `pop`, etc ?

 **Réponse** : *NON*

 **Remarque** : parce que les tuples sont *immuables*

## Multi-affectations, multi-retours

```
# ==== Prog principal
t = 1, 2, 3
print(type(t))

# ==== Resultat
<class 'tuple'>
```

```
def division(a, b):
    return a // b, a % b

# ==== Prog principal
res = division(20, 5)
print(res, type(res))

# ==== Resultat
(4, 0) <class 'tuple'>
```

**i Remarque :** en interne, Python utilise des tuples pour gérer les *multi-affectations* et les *multi-retours*

## Des modifications ?

```
L = ['a', 'b']  
t = (1, 2, L)  
t[2].append('c')  
print(t)  
  
# ==== Resultat  
(1, 2, ['a', 'b', 'c'])
```

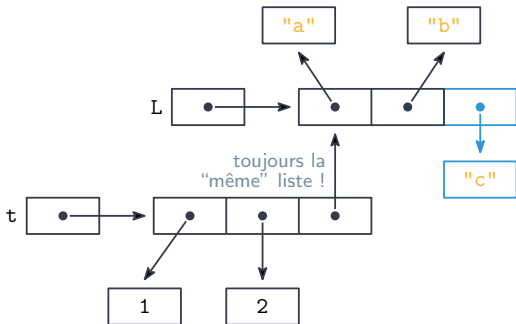
```
L = ['a', 'b']  
t = (1, 2, L)  
t[2] = [ ]  
print(t)
```

```
# ==== Resultat  
TypeError: 'tuple' objet does not support item assignment
```

❌ **Problème :** Mais on a pas dit qu'un tuple était *immuable* ?

## Le retour de la mémoire, 1

```
L = ['a', 'b']  
t = (1, 2, L)  
t[2].append('c')  
print(t)
```



**i Remarque :** on ne change pas *de liste*, on change *la liste*

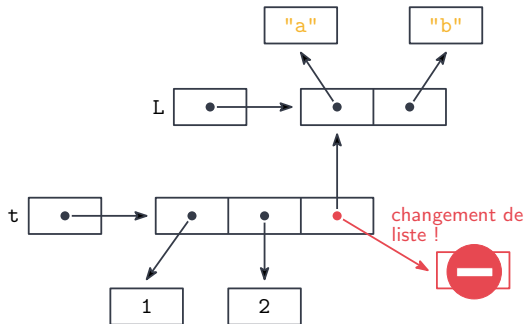
## Le retour de la mémoire, 2

```
L = ['a', 'b']
```

```
t = (1, 2, L)
```

```
t[2] = [ ]
```

```
print(t)
```



**i Remarque :** ici on essaye de changer de liste *dans la tuple*, ça coince

# Résumé



**Rappel :** un tuple (**tuple**) est une *séquence immuable* d'*objets quelconques*



## Attention :

- un tuple n'est *hashable* que s'il contient des *éléments hashables*
- on peut modifier une liste dans un tuple *sans casser l'immutabilité*!

```
>>> T = ("je", "suit", "en", "cours")
```

```
>>> T[1] = "suis"
```

```
TypeError: 'tuple' object does not support item assignment
```



## Exercices

⚙️ **Exercice** : écrire une fonction `merge` qui prend en paramètres deux tuples T1, T2 de même taille et qui renvoie la liste L des paires des  $i$ -èmes éléments de T1 et T2.

```
>>> T1 = ("a", "b", "c")
>>> T2 = (1, 2, 3)
>>> merge(T1, T2)
[("a", 1), ("b", 2), ("c", 3)]
```

⚙️ **Exercice** : écrire une fonction `distance` qui prend en paramètre une liste L de points dans  $\mathbb{R}^2$  et qui calcule la plus petite distance possible entre deux points de L. Un point p est représenté par un tuple (xp, yp)

```
>>> L = [(1, 2), (5, -1), (3, 2)]
>>> distance(L)
2.0
```



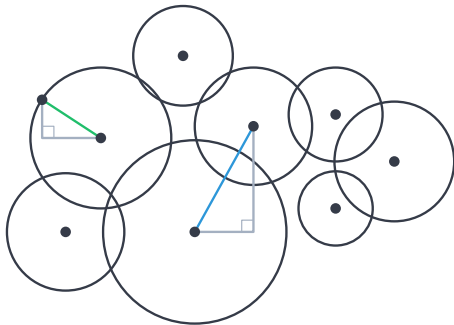
**Rappel** : la distance entre deux points  $p_1 = (x_1, y_1)$  et  $p_2 = (x_2, y_2)$  est  $d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

## Exercice : et rond et rond petit patapon

⚙ **Exercice** : on peut représenter un cercle  $C$  dans  $\mathbb{R}^2$  de (au moins) deux manières :

1. son centre  $(x_c, y_c)$  et un point sur le cercle  $(x_r, y_r)$
2. son centre  $(x_c, y_c)$  et son rayon  $r$

Étant donnée une liste de cercles, on veut trouver les cercles qui s'intersectent.



## Exercice

### Exercice : (cercles, suite)

1. Écrire une fonction `reprRayon` qui prend en paramètre un cercle sous la forme `((xc, yc), (xr, yr))`, et qui renvoie un tuple `((xc, yc), r)` où `(xc, yc)` est le centre du cercle et `r` son rayon.
2. Écrire une fonction `cercleInter` qui prend en paramètres 2 cercles sous la forme `((xc, yc), r)` et qui renvoie `True` si les cercles s'intersectent, `False` sinon.
3. Écrire une fonction `trouverInter` qui prend en paramètres une liste de cercles (sous la 1ère forme) et qui renvoie la liste des paires (`tuple`) d'indices des cercles qui s'intersectent deux à deux.



**Astuce :** on peut utiliser la fonction `sqrt` du module `math`

# Ensembles

## Dictionnaires

## Tuples

## Ensembles

- Définition

- Opérations

- Résumé


## Collections : le mot de la fin



**Remarque :** sur le principe : *set = dictionnaire avec que des clés*

# Définition

 **Définition :** un *ensemble* (type `set`) est une *collection non-ordonnée mutable* d'objets *hashables distincts*.

 **Syntaxe :** un ensemble est défini avec des accolades `{ }` (comme les dictionnaires), on y sépare les éléments par des virgules `,`

```
ens = {elt1, elt2, ...}
```

```
ens = set() # ensemble vide  
ens = {2, 3, 4, 8, "ouille"}
```

## Attention :

- ne pas confondre ensemble Python et ensemble en maths
- les ensembles ne sont *pas hashables*  $\implies$  pas d'*ensemble d'ensembles* !

## Définition en compréhension

```
>>> E1 = {i for i in range(5)}
```

```
>>> E1
```

```
{0, 1, 2, 3, 4}
```

```
>>> E2 = {c for c in "patapon" if c not in "aeiouy"}
```

```
>>> E2
```

```
{'p', 'n', 't'}
```

```
>>> E3 = {i + j for i in "abc" for j in "xyz"}
```

```
>>> E3
```

```
{'cz', 'cx', 'cy', 'bx', 'ay', 'ax', 'az', 'by', 'bz'}
```


**i Remarque :** la dernière formulation est équivalente à :

```
for i in "abc":
```

```
    for j in "xyz":
```

```
        # ajouter x + y a E
```

## Accès, appartenance

 **Syntaxe** : les ensembles sont des collections, on peut tester l'appartenance avec `in`

```
>>> E = {1, 2, 3}
>>> 1 in E
True
>>> 4 in E
False
```

Mais ce ne sont pas des séquences

```
>>> E = {1, 2, 3}
>>> E[0]
TypeError: 'set' object is not subscriptable
```

 **Attention** : *pas d'indexation*, donc *pas d'accès direct* !

 **Remarque** : pour la recherche, *meilleure complexité que les listes*.

## Parcours

```
# ==== Prog
E = {"m", "i", "t", "e"}
for i in range(len(E)):
    print(E[i], end=" ")

# ==== Resultat
TypeError
```

```
# ==== Prog
E = {"m", "i", "t", "e"}
for e in E:
    print(e, end=" ")

# ==== Resultat
m i e t
```

Parcours par les éléments :

```
for e in E:
    print(e)
```



## Ajout d'un élément

- `add` *ajoute* un élément s'il n'est pas déjà présent

```
>>> E = {1, 2, 3}
>>> E.add(3)
>>> E.add(4)
>>> E
{1, 2, 3, 4}
```


### Remarque :

- sans ordre, pas d'insertion indexée !
- complexité d'insertion meilleure que pour une liste

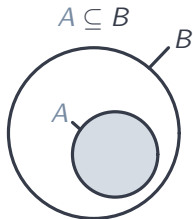
## Suppression d'un élément

- `pop` *supprime* et *renvoie* un élément *arbitraire*
- `discard` *supprime* un élément
- `remove` *supprime* un élément, ou donne `KeyError` si l'élément n'est pas là

```
>>> E = {1, 2, 3, 4}
>>> E.pop()
1
>>> E.discard(2)
>>> E.remove(3)
>>> E
{4}
>>> E.remove(2)
KeyError: 2
```

 **Attention :** si E est vide, `E.pop()` donne une `KeyError`

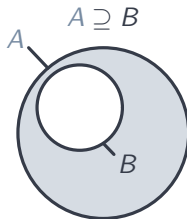
## Comparaisons d'ensembles



`A.issubset(B)`

ou

`A <= B`



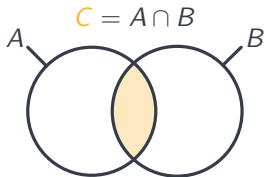
`A.issuperset(B)`

ou

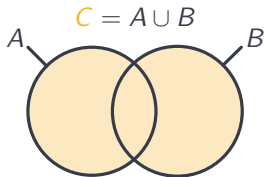
`A >= B`

**i** **Remarque** :  $A == B$  ssi  $A <= B$  et  $B <= A$  (ou `issubset`, `issuperset`)

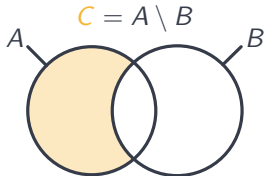
# Opérations



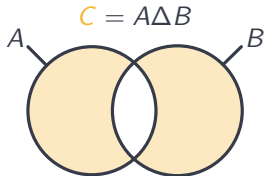
$C = A.\text{intersection}(B)$   
ou  
 $C = A \ \& \ B$



$C = A.\text{union}(B)$   
ou  
 $C = A \ | \ B$



$C = A.\text{difference}(B)$   
ou  
 $C = A - B$



$C = A.\text{symmetric\_difference}(B)$   
ou  
 $C = A \ \wedge \ B$

**?** Question : pourquoi *deux* versions de chaque opération ?

```
>>> A = {1, 2, 3}
>>> print(A.union("bonjour"))
{'b', 1, 2, 3, 'u', 'r', 'o', 'n', 'j'}
```

```
>>> A = {1, 2, 3}
>>> print(A + "bonjour")
TypeError: unsupported operand type(s) for +: 'set' and 'str'
```

**✓ Réponse :**

- les opérateurs +, &, |, ... ne s'appliquent *que sur des set* !
- les méthodes `union`, `intersection`, ... peuvent utiliser des *collections* !

## Modification d'un ensemble


**? Question :** les opérations qu'on a vues renvoient un nouvel ensemble C, comment *mettre à jour* A directement ?

**✓ Réponse :** il existe une version *update* de ces opérations

une fonction par opération :

- $A.\text{update}(B)$  : A devient  $A \cup B$  ( $\simeq$  *extend* des listes)
- $A.\text{intersection\_update}(B)$  : A devient  $A \cap B$
- $A.\text{difference\_update}(B)$  : A devient  $A \setminus B$
- $A.\text{symmetric\_difference\_update}(B)$  : A devient  $A \Delta B$

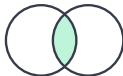
# Résumé

 **Rappel** : un ensemble est une collection *non-ordonnée mutable* d'objets *hashables distincts*

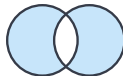
 **Attention** : pas d'*indexation*, de *slicing*, etc !



issubset



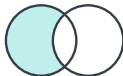
intersection



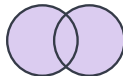
symmetric\_difference



issuperset



difference



union

## Exercice

⚙️ **Exercice** : Écrivez une fonction qui prend en paramètres deux ensembles et qui vérifie que ces deux ensembles ont plus d'éléments en commun qu'ils n'en ont de différents.

⚙️ **Exercice** : Écrivez une fonction `doublons` qui prend en paramètre une liste `L` et qui renvoie une nouvelle liste qui correspond à `L` à laquelle les doublons ont été enlevés. Par exemple, on doit avoir :

```
>>> L = [1, 1, 3, 2, 3, 4, 5, 7, 8, 7]
>>> doublons(L)
[1, 3, 2, 4, 5, 7, 8]
```

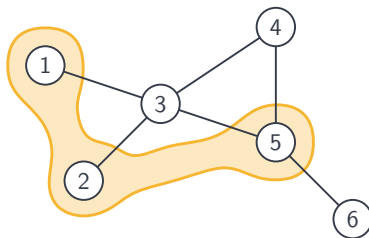


## Exercice

⚙️ **Exercice** : Écrivez une fonction `stable` qui prend en paramètre un ensemble  $E$  de paires d'entiers compris entre 0 et 9, et qui renvoie un sous-ensemble  $S$  de  $\{0, \dots, 9\}$  tel que :

- pour tout  $i, j$  de  $S$ , ni  $(i, j)$  ni  $(j, i)$  ne sont dans  $E$
- ajouter un seul élément à  $S$  casse la propriété juste au dessus

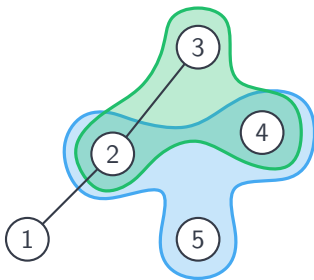
```
>>> E = {(1, 3), (3, 2), (3, 4), (3, 5), (5, 6), (4, 5)}  
>>> S = stable(E)  
>>> S  
{1, 2, 5}
```



## Exercice

⚙️ **Exercice** : dans cet exercice on considère une structure que l'on va appeler « hypergraphe ». Un hypergraphe est une liste contenant des ensembles, chacun des ensembles contenant des entiers entre 1 et 5. Par exemple

$$H = [\{1, 2\}, \{2, 3\}, \{2, 3, 4\}, \{2, 4, 5\}]$$



## Exercice

Écrivez les trois fonctions suivantes, chacune prend en paramètre un hypergraphe  $H$  et renvoie un booléen :

1. une fonction `universel` qui teste s'il existe un entier entre 1 et 5 qui soit contenu dans tous les ensembles de  $H$
2. une fonction `sperner` qui teste que tous les ensembles de  $H$  soient incomparables par inclusion
3. une fonction `couverture` qui teste que tous les entiers entre 1 et 5 sont dans au moins un ensemble de  $H$

Par exemple, on doit avoir :

```
>>> H = [{1, 2}, {2, 3}, {2, 3, 4}, {2, 4, 5}]
>>> universel(H)
True # 2 est dans tous les ensembles
>>> sperner(H)
False # {2, 3} est inclus dans {2, 3, 4}
>>> couverture(H)
True # tout le monde apparaît dans un ensemble au moins
```

# Collections : le mot de la fin

Dictionnaires

Tuples

Ensembles

Collections : le mot de la fin

# Conversions

On peut *convertir* une collection en une autre... avec *précaution* !

on utilise les fonctions `str()`, `list()`, `tuple()`, `set()`, `dict()`

```
>>> mot = "kiwi"
>>> L = list("kiwi") # list() constructeur de liste
>>> T = tuple(L)     # tuple() constructeur de tuple
>>> E = set(T)
>>> mot = str(E)
>>> print(L, T, E, mot)
['k', 'i', 'w', 'i'] ('k', 'i', 'w', 'i')
{'w', 'i', 'k'} "{w', 'i', 'k'}"
```

**i Remarque :** ces fonctions sont les *constructeurs* des collections (objet)

## Mais, et les dictionnaires ?

```
>>> mot = "kiwi"
```

```
>>> print(dict(mot))
```

```
ValueError: dictionary update sequence element #0  
has length 1; 2 is required
```

```
>>> L = [i for i in range(5)]
```

```
>>> print(dict(L))
```

```
TypeError: cannot convert dictionary update sequence  
element #0 to a sequence
```

## Cas des dictionnaires

**! Attention :** un dictionnaire est un ensemble d'items (clé, valeur) ... pour utiliser `dict()` il faut une *collection de paires* !

```
>>> L = ["a", "b", "c"]
>>> d = dict(enumerate(L))
>>> print(d)
{0: 'a', 1: 'b', 2: 'c'}

>>> t = ((0, 1), (0, 2), (1, "a"))
>>> d = dict(t)
>>> print(d)
{0: 2, 1: 'a'}
```



**Rappel :** dans un dictionnaire, chaque clé est *unique* !

Ce qu'on a vu, et le reste ...

**i Remarque :** On a fait un tour *rapide* des différentes collections.

Les collections ont des myriades d'autres fonctions / méthodes

- `clear`, `copy`, `sort`, `reverse`, ...

Pour savoir ce qu'il est possible de faire avec une collection

- `la doc` (encore)
- `les internets`
- Python directement :

```
help(collec) # collec = list, dict, set, str ou tuple
```