

Collections : tuples, dictionnaires et ensembles

Polytech Marseille, GII, 3A

Séverine Dubuisson, Simon Vilmin

`severine.dubuisson@univ-amu.fr`,

`simon.vilmin@univ-amu.fr`

2023 - 2024



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 est *épuisant* mais ...

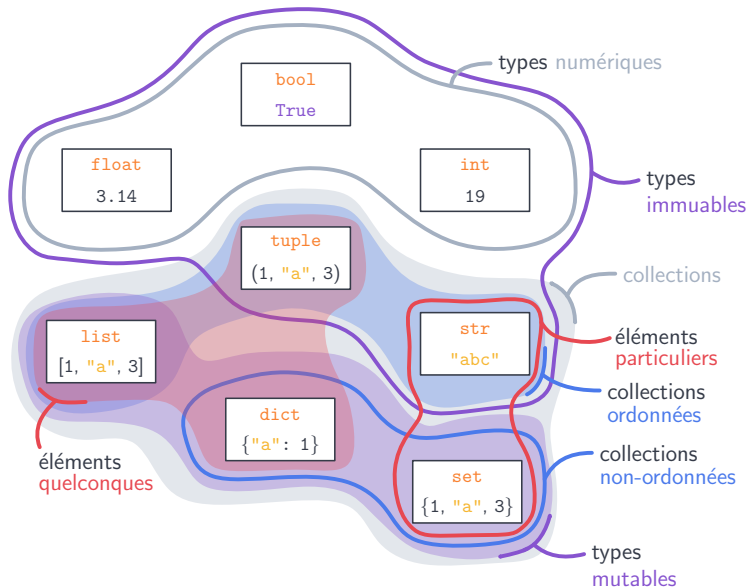
- cours *conceptuellement moins chargé* que celui sur les listes
- portrait *général* des autres collections

! 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, ordonnée/non-ordonnée, 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 *collection ordonnée mutable* d'objets *quelconques*.

Accès, parcours, recherche :

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

Modification, ajout, suppression :

- ajout avec `append`, `insert`
- 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* (type `dict`) est une *collection non-ordonnée 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

Illustration de la syntaxe

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

Hashabilité et unicité des clés :

```
In []: dico = {[0] : 1}
Out[]: TypeError: unhashable type: 'list'

In []: dico = {0: "a", 0: "b"}
In []: dico
Out[]: {0: 'b'}
```


Pourquoi faire

Listes :

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

Dictionnaires :

- permet de « stocker » des données accessibles par un « identifiant »
- on accède à une valeur par son « *nom / identifiant* » (sa clé)
- parfois appelés *tableaux associatifs* : à une clé on associe une valeur

i 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

Un *dictionnaire des synonymes*

- mot : liste de synonyme

Modélisation :

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

Exemple : livres

Une *collection de livres*.

- ISBN : info du livre (auteur.e, titre, ...)

Modélisation

```
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"]
```

Exemple : Unicode

Table des *correspondances entre caractères et leur code Unicode*


- caractère : code


Modélisation

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

i Remarque : en vrai, la fonction `ord()` donne directement le code d'un caractère

Table de hachage

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

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

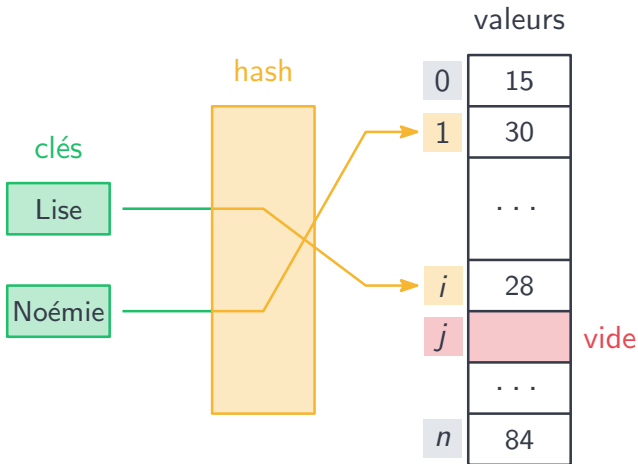
Objectif d'une table de hachage :

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

Pour y arriver :

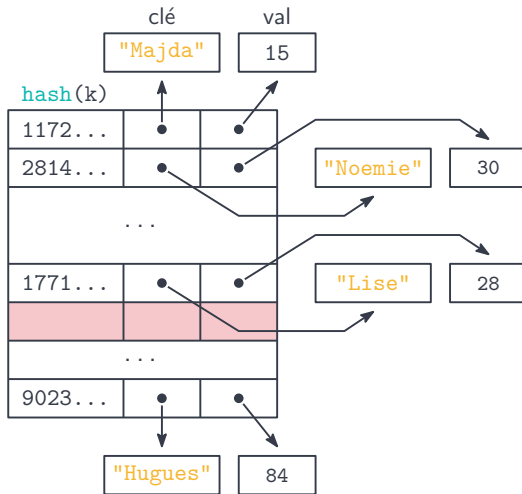
- on part d'un tableau
- et on utilise une *fonction de hachage* qui transforme une *clé* en l'*indice* où se trouve la valeur
- le tableau peut contenir beaucoup de *cases vides*

Illustration



! **Attention :** représentation de la *structure de données abstraite*

Les dict Python



i Remarque : tableau où chaque case contient : `hash(k)` (`k` la clé),
pointeur sur la clé, pointeur sur la valeur

Accès

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

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

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

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


Appartenance



Rappel : les dictionnaires sont pensés « clés »



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

```
In [] animal = {"nom": "chien", "mammifere": True, "pattes": 4}
In [] "pattes" in animal
Out[]: True
In [] "ecailles" in animal
Out[]: False
In []: "chien" in animal
Out[]: 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])
```

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 : `values()` et `items()` ne *renvoient pas une copie* des éléments du dictionnaire, mais *une vue* \implies coût en mémoire *faible* !

Ajout d'un item

pour ajouter une paire (k, valeur), on utilise directement les crochets

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

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

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

Suppression d'un item

- `pop` *supprime* une paire (cle, valeur) donnée par la *clé* et renvoie la *valeur*
- `popitem` *supprime* et *renvoie* la dernière paire (cle, valeur) ajoutée au dictionnaire
- `del` permet de supprimer une paire (cle, valeur) à partir de la *clé*

```
In []: D = {"a": 1, "b": 2, "c": 3, "d": 4}
In []: a = D.pop("a")
In []: del D["b"]
In []: paire = D.popitem()
In []: print(a, paire, D)
Out []: 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 $\implies D1 \mid D2 \neq D2 \mid D1$

```
In []: D1, D2 = {"a": 1, "c": 3}, {"a": 2, "b": 3}
```

```
In []: D1 |= D2
```

```
In []: D1
```

```
Out []: {"a": 2, "b": 3, "c": 3}
```

```
In []: D4 = D1 | {"c": 84, "d": 5}
```

```
In []: D4
```

```
Out []: {"a": 2, "b": 3, "c": 84, "d": 5}
```

Résumé



Rappel : un dictionnaire

- est de type `dict`
- est *non-ordonné*, *mutable*,
- est constitué d'*items* de la forme `cle: valeur`
- indexe les valeurs dans un tableau *autrement que par leurs positions*



Attention :

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

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

Exercice

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

```
In []: D = {"a": 1, "b": 2}
In []: Di = invertKV(D)
In []: Di
Out []: {1: "a", 2: "b"}
```

i Remarque : on considère que les valeurs de `D` sont toutes hashables

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.

```
In []: D1 = {"a": 1, "b": 2}
In []: D2 = {"a": "AAAAAAH"}
In []: D1.update(D2)
In []: D1
Out []: {"a": "AAAAAAH", "b": 2}
```

Écrire 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.

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

Exercice

⚙️ **Exercice** : Écrire une fonction `getLivrebyAuteur` qui prend en paramètre un dictionnaire de livres (comme vu précédemment) et qui construit le dictionnaire auteur ou les items ont la forme

```
nom_auteur: listes des livres
```

```
# exemple d'item du dico livre
```

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

```
# exemple d'item du dico auteur
```

```
{  
    "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* (type **tuple**) est une *collection ordonnée immuable* d'objets *quelconques*.

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

```
tupl = (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)

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

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

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

Accès, appartenance

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

```
In []: T = {"je", "suis", "un", "tuple"}
```

```
In []: T[2]
```

```
Out []: un
```

```
In []: T[0:2]
```

```
Out []: je suis
```

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

```
In []: "suis" in T
```

```
Out []: True
```

```
In []: "serpent" in T
```

```
Out []: 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-affectation, 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*

Exercice

⚙️ **Exercice :** qu'affiche le programme suivant

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

⚙️ **Exercice :** qu'affiche le programme suivant

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

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

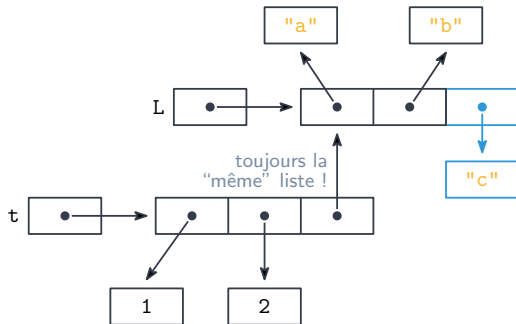
Le retour de la mémoire, 1

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

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

```
L.append('c')
```

```
print(t)
```



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

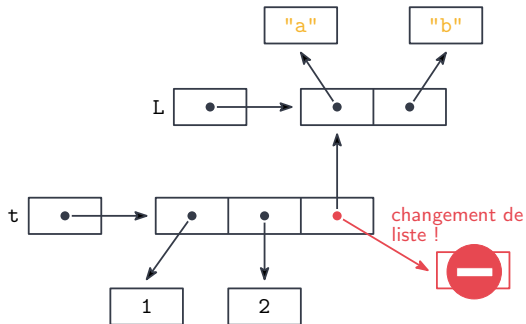
Le retour de la mémoire, 2

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

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

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

```
print(t)
```



i Remarque : Ici on essaye de changer de liste *dans le tuple*, ça coince

Résumé



Rappel : un tuple est

- de type `tuple`
- *ordonné, immuable,*
- constitué 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'immuabilité*!

```
In []: T = ("je", "suit", "en", "cours")
```

```
In []: T[1] = "suis"
```

```
Out []: 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.

```
In []: T1 = ("a", "b", "c")
In []: T2 = (1, 2, 3)
In []: merge(T1, T2)
Out []: [("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 (x_p, y_p)

```
In []: L = [(1, 2), (5, -1), (3, 2)]
In []: distance(L)
Out []: 2.0
```



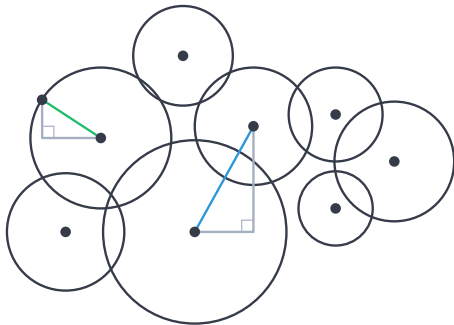
Rappel : la distance entre deux points $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$ est $d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

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", (True, False)}
```

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

```
In []: E1 = {i for i in range(5)}
```

```
In []: E1
```

```
Out []: {0, 1, 2, 3, 4}
```

```
In []: E2 = {c for c in "patapon" if c not in "aeiouy"}
```

```
In []: E2
```

```
Out []: {'p', 'n', 't'}
```

```
In []: E3 = {i + j for i in "abc" for j in "xyz"}
```


```
In []: E3
```

```
Out []: {'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`

```
In []: E = {1, 2, 3}
```

```
In []: 1 in E
```

```
Out []: True
```

```
In []: 4 in E
```


```
Out []: False
```

Mais ils ne sont pas ordonnés ...

```
In []: E = {1, 2, 3}
```

```
In []: E[0]
```

```
Out []: TypeError: 'set' object is not subscriptable
```

 **Attention** : *pas d'indexation* dans les ensembles, donc *pas d'accès direct* à un élément !

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

```
In []: E = {1, 2, 3}
```

```
In []: E.add(3)
```

```
In []: E.add(4)
```

```
In []: E
```

```
Out []: {1, 2, 3, 4}
```

Remarque :

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

Suppression d'un élément

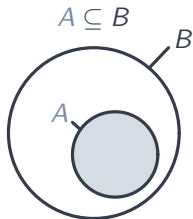
- `pop` *supprime* et *renvoie* un élément *arbitraire*
- `discard` *supprime* un élément
- `remove` *supprime* un élément, ou donne `KeyError` si l'élément n'est pas présent

```
In []: E = {1, 2, 3, 4}
In []: E.pop()
Out []: 1
In []: E.discard(2)
In []: E.remove(3)
In []: E
Out []: {4}
In []: E.remove(2)
Out []: KeyError: 2
```



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

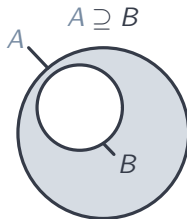
Comparaisons d'ensembles



`A.issubset(B)`

ou

`A <= B`



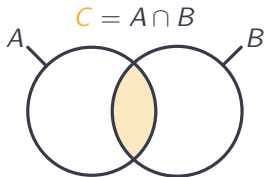
`A.issuperset(B)`

ou

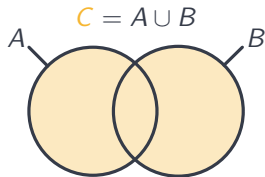
`A >= B`

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

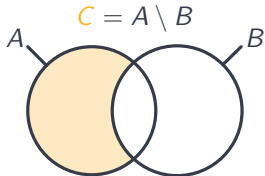
Opérations



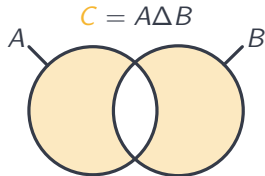
$C = A.\text{intersection}(B)$
ou
 $C = A \ \& \ B$



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

```
In []: A = {1, 2, 3}
```

```
In []: print(A.union("bonjour"))
```

```
Out[]: {'b', 1, 2, 3, 'u', 'r', 'o', 'n', 'j'}
```

```
In []: A = {1, 2, 3}
```

```
In []: print(A + "bonjour")
```

```
Out[]: 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

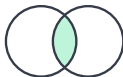
- de type `set`
- *non-ordonné*, *mutable*,
- constitué d'*objets hashables distincts*



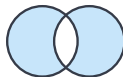
Attention : pas d'ordre explicite \implies pas d'*indexation*, de *slicing*, etc !



`issubset`



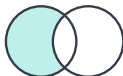
`intersection`



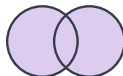
`symmetric_difference`



`issuperset`



`difference`



`union`

Exercice

⚙️ **Exercice** : un *hypergraphe* est une paire $\mathcal{H} = (X, \mathcal{E})$ où :

- X est un ensemble (fini) de *sommets*
- $\mathcal{E} = \{E_1, \dots, E_m\}$ est une collection (non-vide) de sous-ensembles de X , appelés *arêtes*

On dit que \mathcal{H} :

- a un *élément universel* s'il existe $x \in X$ t.q. $x \in E_i$ pour tout $E_i \in \mathcal{E}$
- est *Sperner* si $E_i \not\subseteq E_j$ et $E_j \not\subseteq E_i$ pour tout $E_i, E_j \in \mathcal{E}$ (E_i, E_j distincts)
- est une *couverture* de X si $\bigcup_{1 \leq i \leq m} E_i = X$

On modélise \mathcal{H} comme un **tuple** contenant le **set** X et la **list** \mathcal{E} des **set** E_i .
Écrire les fonctions **universal**, **sperner** et **cover** qui prennent en argument \mathcal{H} teste les propriétés décrites ci-dessus.

```
In []: H = ({1, 2, 3, 4}, [{1, 2}, {2, 3}, {2, 3, 4}])
```

```
In []: print(universal(H), sperner(H), cover(H))
```

```
Out []: True False True
```

Collections : le mot de la fin

Dictionnaires

Tuples

Ensembles

Collections : le mot de la fin

Conversions

On peut *convertir* une collection en une autre collection ... avec quelques *précautions*!

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

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

i Remarque : parenthèse objet, ces fonctions sont les *constructeurs* des collections

Mais, et les dictionnaires ?

```
In []: mot = "kiwi"
```

```
In []: print(dict(mot))
```

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

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

```
In []: print(dict(L))
```

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

Cas des dictionnaires

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

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



```
In []: t = ((0, 1), (0, 2), (1, "a"))
In []: d = dict(t)
In []: print(d)
Out[]: {0: 2, 1: 'a'}
```

i Remarque : on rappelle que 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)
- `la doc, mais en livre !` (oui.)
- `les internets`
- Python directement :

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