# Chapitre : Les collections


- *Author* : **Ibrahima SY**
- *Email* : syibrahima31@gmail.com
- *Github* : Cliquez
- *Linkdin* : Cliquez
- *School* : Institut Supérieur Informatique (**ISI**)
- *Spécialité* : Licence 2 **GL**

# PLAN

1. Les tuples

2. Table de Hash

3. Les dictionnaires

4. Les ensembles

5. Les Exceptions

# Les Tuples

- Le tuple est une séquence, on peut donc appliquer les opérations comme le test d'appartenance avec in, accéder aux différents éléments avec un crochet, faire du slicing dessus.
- un tuple peut référencer des objets complètement hétérogènes. C'est donc très proche de la liste.
- le tuple est un objet immuable. Ça veut dire qu'une fois qu'on a créé le tuple, on ne peut plus le modifier.

# La syntaxe des tuples

- On crée un tuple en utilisant simplement des parenthèses ouvrantes et fermantes .
- On peut créer un tuple avec un élément avec la syntaxe suivante : `t = ( 4, )`.
- On peut créer un tuple de plusieurs éléments, qui contient des objets complètement hétérogènes
- Une caractéristique importante du tuple c'est que les parenthèses sont facultatives.

```
>>> t =(1,)
>>> t
(1,)
>>> t = (1,2,3,4)
>>> t
(1,2,3,4)
>>> t = 1,2,3,4
>>> t
(1,2,3,4)
```

# Tuple unpacking

- Le **tuple unpacking** fonctionne de la manière suivante : nous avons dans un tuple deux variables, a et b, et on dit que ces variables sont égales à une séquence qui doit avoir le même nombre d'éléments que l'on a dans notre tuple.
- En Python, il existe également la notion de extended **tuple unpacking**.

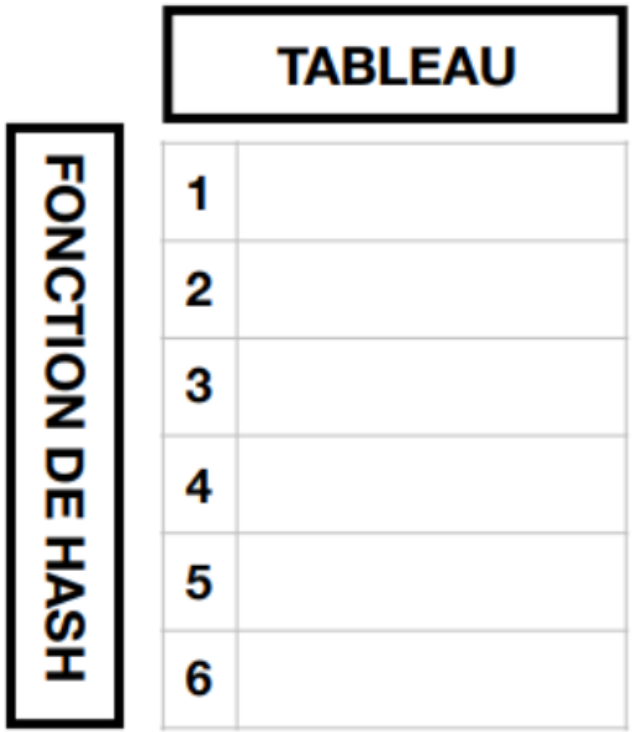
```
# notion de tuple unpacking
>>> a = (2,3)
>>> a
2
>>> b
3
```

```
# Extended tuple unpacking unpacking
>>> *x, y = [1,2,3,4]
>>> x
[1,2,3]
>>> y
4
```

# Tables de hash

La **tables de hash**, une structure de données qui permet de répondre à certaines limitations des types séquence.

**Ci-contre :** Un tableau avec 6 éléments et une fonction de hash. Lorsqu'on lui passera un objet, la fonction calculera une valeur comprise entre 1 et 6. Le but de cette fonction est de créer une correspondance entre un objet quelconque et une case du tableau.
**f(x) appartient à [1,6]**
Cet ensemble **fonction de hash** et **tableau** constitue **une table de hash**.



## Les limitations des types séquences

Les types séquence ont été optimisés pour **l'accès**, la **modification** et **l'effacement** en fonction d'un **numéro de séquence**. Cependant ces types n'ont pas été optimisés pour le **test d'appartenance**.

```
>>> %timeit  "x" in range(1000)
>>> %timeit  "x" in range(10_000)
>>> %timeit  "x" in range(100_000)
```

Mais supposons que nous avons des âges dans notre liste : `t = [18, 35]`. On pourrait vouloir écrire `t['alice'] = 35` pour lier un nom à un âge, donc indicer notre séquence non plus avec des entiers mais avec des chaînes de caractères. Et bien ça on ne peut pas le faire.

- La structure de données tables de hash permet de répondre à ces deux limitations
- Une table de hash est constituée d’un tableau et d’une fonction dont le but est : quand on passe à notre fonction un objet, elle calcule une valeur comprise entre le nombre de lignes du tableau.

# Les dictionnaires

- Les dictionnaires sont des tables de hash. On a donc un temps d'accès, d'insertion, d'effacement et un test d'appartenance qui sont indépendants du nombre d'éléments.
- Les dictionnaires sont des objets mutables
- Dans un dictionnaire, on peut avoir comme clef n'importe quel objet qui est hashable, c'est-à dire un objet sur lequel on peut calculer la fonction de hash.
- En python tous les objets immuables sont hashables et que tous les objets mutables ne sont pas hashables

## Création d'un dictionnaire

Pour initialiser un dictionnaire , on utile la syntaxe suivante:

```
>>> d = dict()
>>> d =  {}
```

## Ajouter des valeurs dans un dictionnaire

Pour ajouter des valeurs à un dictionnaire il faut indiquer une clé ainsi qu'une valeur:

```
>>> a = {}
>>> a["nom"] = "curry"
>>> a["prenom"] = "Stephen"
>>> a
{"nom":"curry", "prenom":"Stephen"}
```

La méthode `get` vous permet de récupérer une valeur dans un dictionnaire et si la clé est introuvable, vous pouvez donner une valeur à retourner par défaut:

```
>>> data = {"name": "wayne", "age":45 }
>>> data.get("name")
"wayne"
>>> data.get("adresse", "Adresse Inconnue")
>>> "Adresse Inconnue"
```

## Existence d'une cle dans un dictionnaire

Vous pouvez utiliser la méthode `has_key` pour vérifier la présence d'une clé que vous cherchez( a supprimer)

```
>>> data = {"name": "wayne", "age":45 }
>>> data.has_key("name")
True
```

## Supprimer une entrée dans un dictionnaire

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes:

```
>>> data = {"name": "wayne", "age":45 }
>>> del data["name"]
>>> data
{"age":45 }
```

## Utilisation des tuple des tuples

Une des forces de python est la combinaison tuple/dictionnaire qui fait des merveilles dans certains cas comme lors de l'utilisation de coordonnées.

```
>>> d ={}
>>> d[(3,2)] = 12
>>> d[(4,5)] = 13
>>> d
{(4,5):13, (3,2):12}
```

## Fusionner des dictionnaires

La méthode `update` permet de fusionner deux dictionaires .

```
>>> dic1 = {"nom":"curry"}
>>> dic2 = {"prenom":"Stephen"}
>>> dic1.update(dict2)
>>> dic1
{"nom":"curry", "prenom": "Stephen"}
```

## Supprimer une entrée dans un dictionnaire

Comme pour toute variable, vous ne pouvez pas copier un dictionnaire en faisant `dic1 = dic2` :

```
>>> dict1 = {"k1":"value1", "k2":"value2"}
>>> dict2 = dict1
>>> dict1["k1"] = "XXX"
>>> dict2
{"k1":"XXX", "k2":"value2"}
```

Pour créer une copie indépendante vous pouvez utiliser la méthode `copy` :

```
```python
>>> dict1 = {"k1":"value1", "k2":"value2"}
>>> dict2 = dict1.copy()
>>> dict1["k1"] = "XXX"
>>> dict2
{"k2":"value2","k1":"value1"}
```
```

# Les dictionnaires

## Les vues

- Les méthodes `.keys()`, `.values()` et `.items()` retournent un objet particulier qu'on appelle une vue.
- En Python, une vue est un objet sur lequel on peut itérer. On peut donc faire une boucle for sur cet objet. Et on peut également faire un test d'appartenance, donc faire par exemple in directement sur cette vue.
- La caractéristique principale des vues, c'est qu'elles sont mises à jour en même temps que le dictionnaire.

```
>>> L = [("a",1),("b",2)]
>>> d = dict(L)
>>> valeurs = d.values()
>>> valeurs
dict_values([1,2])
>>> d["a"] = 100
>>> valeurs
dict_values([100,2])
```

# Les dictionnaires

## Parcourir les dictionnaires

Une manière classique de parcourir les dictionnaires est d'utiliser une boucle `for` en utilisant la notation de tuple unpacking :

```
>>> d = {"a":1, "b":2}
>>> for k,v in d.items():
        print(k,v)
a  1
b  2
```

# Les dictionnaires

## Parcourir les dictionnaires

L'itérateur sur les dictionnaires sans spécification de vue fonctionne directement sur les clefs

```
>>> for k in d:
        print(k)
a
b
```

# Les ensembles

- Les sets sont très proches des dictionnaires. Comme les dictionnaires, ils permettent de faire des tests d'appartenance, d'accéder / effacer / modifier des éléments indépendamment du nombre d'éléments.
- Les `sets` sont également des objets mutables. Mais à la différence des dictionnaires, les sets ne stockent qu'une clef. Il n'y a pas de valeur correspondante.
- La raison c'est que le set a été optimisé et créé pour des opérations spécifiques. Par exemple pour garder uniquement le nombre d'éléments uniques d'une séquence.
- Une autre opération pour laquelle le set est très utilisé : c'est pour faire des tests d'appartenance sur les éléments d'une séquence.

# Les ensembles

On peut créer un set vide avec la fonction built-in `set()` : `s = set()`. Ce qui nous donne un objet de type set qui est vide.

- On peut également créer un set avec des accolades : `s = {1, 2, 3, 'a', True}`. La différence entre les notations d'un set et d'un dictionnaire, c'est que dans le set il n'y a pas la notation deux points (:) qui sépare la clef et la valeur.

On peut aussi créer un set à partir d'une séquence.

```
>>> set()
>>> {1,2,3}
>>> L = [1,2,2]
>>> set(L)
{1,2}
```

- La fonction built-in `len()` pour obtenir le nombre d'éléments
- Le test d'appartenance est fait avec l'instruction `in`.
- Dans l'ensemble `s`, on peut ajouter des éléments avec la méthode `.add()`
- On peut aussi ajouter une séquence d'éléments avec la méthode `.update()`
- On peut également faire des opérations d'ensemble classiques sur un set. Pour `s1 : {1, 2, 3}` et `s2 = {3, 4, 5`
- une différence entre deux ensemble `s1 - s2`
- Union : `s1 | s2`
- Intersection : `s1 & s2` renvoie `{3}`

# Les exceptions

## Les erreurs de syntaxe

- Python lève des exceptions quand il trouve une erreur dans le code (erreur de syntaxe ou exécution
  d'une opération) et s'interrompt.
- Pour éviter de faire planter le programme on utilise alors les mécanismes d'exceptions. Nous allons mettre les instructions à tester dans un premier bloc et les instructions à exécuter en cas
  d'erreur dans un autre bloc.

```
try:
# Bloc à essayer
except:
# Bloc qui sera exécuté en cas d'erreur
```

# Les exceptions

## Les exceptions proprement dit

Dans l'ordre, nous trouvons :

- Le mot-clé try suivi des deux points « : » : bloc d'instructions à essayer.
- Le mot-clé except suivi des deux points « : » : bloc d'instructions qui sera exécuté si une erreur est trouvée dans le premier bloc. Il se trouve au même niveau d'indentation que try.

```
try:
 resultat = numerateur / denominateur

except:
 print ("on peut pas diviser pae zéro ")
```

- On peut préciser le type de l'exception à traiter au niveau du bloc except.

```
try:
 resultat = numerateur / denominateur

except ZeroDivisionError  :
 print ("on peut pas diviser pae zéro ")
```

- Dans un bloc try, le mot -clé else va permettre d'exécuter une action si aucune erreur ne survient dans le bloc

```
try:
 resultat = numerateur / denominateur

except ZeroDivisionError  :
 print ("on peut pas diviser pae zéro ")

else :
        print(f"les resultat est {resultat}" )
```

- le mot-clé finally permet d'exécuter du code après un bloc try, quelle que soit le résultat de l'exécution du bloc `try`.

```
try:
 resultat = numerateur / denominateur

except ZeroDivisionError  :
 print ("on peut pas diviser pae zéro ")

else :
        print(f"les resultat est {resultat}" )

finally:
        print("fin de l' opérarion")
```