On a vu en Python une structure pour stocker une collection de données, les tableaux. Cette structure a un certain nombre de spécifications :

— elle est **indexée** par des **entiers**, c'est à dire que chaque donnée est associée à un indice, ici une position entière dans le tableau.

— elle est **mutable**, c'est à dire que chaque donnée du tableau à une position donnée peut être modifiée.

— elle est **modifiable**, c'est à dire qu'on peut ajouter ou enlever des données.

— elle autorise les **doublons**, c'est à dire que la même donnée peut être présente plusieur fois dans le tableau.

Nous allons voir dans ce chapitre d'autres structures existant en python qui ont des spécifications différentes.

1 Ensembles

1.1 Création

Les ensembles sont des structures non indexée, non-mutables (car c'est une notion liée à l'indexation), modifiables et sans-doublon. Ils représentent les ensembles mathématiques.

Pour créer un ensemble, comme pour les tableaux, plusieurs options sont offertes par Python :

— par déclaration simple :

```
s = {1, 4, 9, 16, 25}
```

— par création d'un ensemble vide et ajout d'éléments :

```
s = set() #s est un ensemble vide
for i in range(1,6):
    s.add(i**2)
```

— par compréhension :

```
s = {i**2 for i in range(1,6)}
```

1.2 Itération et appartenance

On peut parcourir les éléments d'un ensemble comme avec les tableaux :

```
s = {1,2,9,16,25}
for x in s:
    print(x)
```

On peut tester si un élément `x` est présent dans dans un tableau `s` avec la commande `x in s` :

```
s = {1,4,9,16,25}
print(4 in s)
print(3 in s)
```

1.3 Non-indexation et non-mutabilité

Les ensembles ne sont pas indexés : il n'est pas possible d'accéder à un élément particulier de l'ensemble. En particulier, il est donc aussi impossible de modifier un élément précis.

1.4 Modification

On peut ajouter un élément `s` à un ensemble `s` via la commande `s.add(x)`.

```
s = {1,2,3}
s.add(4)
print(s)
```

Il est aussi possible de supprimer un élément avec `s.remove(x)`.

```
s = {1,2,3}
s.remove(2)
print(s)
```

Remarque : la méthode `remove` est également disponible sur les tableaux.

1.5 Absence de doublon

Les ensembles ont également la spécificité de ne jamais présenter de doublons, c'est à dire qu'une valeur n'apparaît toujours qu'une fois au plus dans un ensemble.

```
s = {1,1,2}
print(s)

s.add(2)
print(s)
```

1.6 Opérations

Soient A et B deux sous-ensembles un ensemble E :

— L'**union** de A et B est définie par :

$$A \cup B = \{x \in E \ / \ (x \in A) \vee (x \in B)\}$$

— L'**intersection** de A et B est définie par :

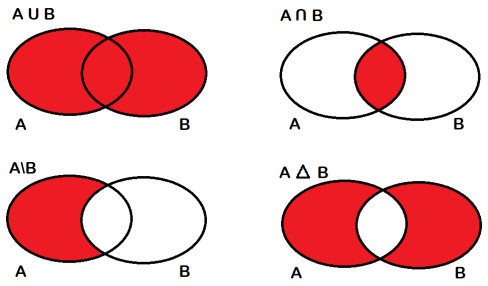
$$A \cap B = \{x \in E \ / \ (x \in A) \wedge (x \in B)\}$$

— La **défférence** de A et B est définie par :

$$A \backslash B = \{x \in E \ / \ (x \in A) \wedge (x \notin B)\}$$

— La **défférence symétrique** de A et B est définie par :

$$A \Delta B = (A \backslash B) \vee (B \backslash A) = \{x \in E \ / \ (x \in A) \oplus (x \in B)\}$$



Les opérations mathématiques d'union, intersection, etc. sont disponibles sur les ensembles Python :

```
s1 = {1,2,3}
s2 = {1,2,4}

print(s1 | s2) #union de s1 et s2
print(s1 & s2) #intersection de s1 et s2
print(s1 - s2) #difference de s1 et s2 (s1 prive de s2)
print(s1 ^ s2) #difference symetrique de s1 et s2
```

Remarque : Ces opérations renvoient de nouveaux ensembles et ne modifient pas les ensembles passés en argument.

1.7 Conclusion

On utilise les d'ensembles lorsqu'on a besoin d'une structure se comportant de la même manière que les ensembles mathématiques. Cela est intéressant lorsqu'on n'a pas besoin d'ordonner les données et que celles-ci doivent être sans doublon.

Spécification	Indéxé	Clé	Mutable	Modifiable	Doublons
Tableaux	oui	indices (entiers)	oui	oui	oui
Ensembles	non	-	-	oui	non

2 n-uplets

2.1 Création

Les n-upplets sont des structures rigides, indexées comme les tableaux (par des indices), non-modifiables et non-mutables et autorisant les doublons.

Pour créer un n-uplet, on procède généralement par déclaration simple :

```
u = ("toto", 12, 1.34, "Tourcoing")
```

Remarque : Les n-uplets ont rarement vocation à être de grande taille, on les utilise généralement pour stocker des données de natures différentes. Dans l'exemple précédent on a utilisé un n-uplet pour stocker des informations sur une personne.

Les n-uplet sont souvent utilisés également dans les fonctions pour renvoyer plusieurs réponses.

```
def plus_moins(a,b):
    return (a+b, a-b)
```

2.2 Itération et appartenance

Bien qu'on ne le fasse généralement pas, on peut parcourir les éléments d'un u-uplet comme avec les tableaux :

```
u = ("toto", 12, 1.34, "Tourcoing")
for x in u:
    print(x)
```

On peut tester si un élément `x` est présent dans dans un n-uplet `u` avec la commande `x in s` :

```
u = ("toto", 12, 1.34, "Tourcoing")
print("toto" in u)
print(3 in u)
```

2.3 Indexation, non-mutabilité, non-modifiabilibé

Les données stockées dans un n-uplet ont une posision (comme dans les tableaux). On peut y accéder selon la même syntaxe :

```
u = ("toto", 12, 1.34, "Tourcoing")
print(u[0])
print(u[2])
```

Il est cependant impossible de modifier une donnée particulière, le code suivant renvoyant une erreur :

```
u = ("toto", 12, 1.34, "Tourcoing")
u[0] = "titi"
```

Les n-uplet ne proposent afin pas de commande permettant d'ajouter ou d'enlever des éléments (pas de `append`, `add`, `remove`)

2.4 Doublons

Les n-uplets autorisent les doublons :

```
u = (1,1,1,1,1)
print(u)
```

2.5 Opérations

Les n-uplets peuvent être utilisés afin de simplifier grandement l'écriture de code, voici quelques astuces autorisées :

```
#echange de valeurs
a = 1
b = 2
a, b = b, a
print(a,b)

#recuperation d'elements
u = ("toto", 12, 1.34, "Tourcoing")
nom, age, taille, ville = toto
print(ville)
```

2.6 Conclusion

Les n-uplets sont utilisé pour stocker temporairement un petit nombre de données, par exemple lors d'un retour de fonction. Les données stockées ne sont pas modifiables et la taille d'un n-uplet est fixe.

Spécification	Indéxé	Clé	Mutable	Modifiable	Doublons
Tableaux	oui	indices (entiers)	oui	oui	oui
Ensembles	non	-	-	oui	non
n-uplets	oui	indices (entiers)	non	non	oui

3 Dictionnaires

3.1 Création

Les dictionnaires sont des structures permettant de stocker des valeurs en associant à chacune une clé ayant le même rôle que les indices dans les tableau mais pouvant être de type quelconque, c'est donc une structure indexée. De plus elle est mutable, modifiable et autorise les doublons.

Pour créer un dictionnaire, plusieurs méthodes sont possibles :

— par déclaration simple :

```
moyenne = {"toto":14.5 , "titi": 14.5, "tutu": 9.5}
moyenne = dict(toto = 14.5, titi = 14.5, tutu = 9.5) #ne fonctionne que si les cles sont des textes
```

— par création d'un dictionnaires vide et ajout d'éléments :

```
moyenne = {} #s est un dictionnaire vide
moyenne["toto"] = 14.5
moyenne["titi"] = 14.5
moyenne["tutu"] = 9.5
```

— par compréhension :

```
carre = {i:i**2 for i in range(1,6)}
```

3.2 Clés et valeurs, accès

Prenons l'exemple du dictionnaire `moyenne` donné en exemple. Ici on distingue deux types de données :

— les noms : `"toto"`, `"titi"`, `"tutu"`. Ce sont les **clés** du dictionnaires.

— les moyennes : `14.5`, `14.5`, `9.5`. Ce sont les **valeurs** du dictionnaire.

Les clés d'un dictionnaires sont à voir comme les indices d'un tableau. À chaque clé d'un dictionnaire est associé une valeur.

Remarque : On appelle les dictionnaires dont les clés sont des chaînes de caractères des n**-uplet nommés**.

On peut, étant donné une clé, connaître et modifier la valeur associée :

```
print(moyenne["toto"])
moyenne["toto"] = 13
print(moyenne["toto"])
```

3.3 Itérations

Il est possible de parcourir un dictionnaire de deux façons : selon les clés ou selon les valeurs.

```
for eleve in moyenne.keys(): #parcours par cles
    print(eleve)

for eleve in moyenne: #parcours par cles, notation plus simple
    print(eleve)

for note in moyenne.values(): #parcours par valeurs
    print(note)
```

Remarque : Si on a besoin à la fois des clés et des valeurs, il faudra parcourir par clé :

```
for eleve in moyenne:
    print(eleve, ":", moyenne[eleve])
```

3.4 Appartenance

Pour savoir si une clé `k` est présente dans un dictionnaire `d`, on peut utiliser la syntaxe : `k in d.keys()` ou plus simplement `k in d`.

```
print("toto" in moyenne)
print(14.5 in moyenne)
```

Pour savoir si une valeur `v` est présente dans un dictionnaire `d`, on peut utiliser la syntaxe : `v in d.values()`.

```
print("toto" in moyenne.values())
print(14.5 in moyenne.values())
```

3.5 Mutabilité et modifiabilité

On ajoute et modifie un couple clé, valeur `k:v` dans un dictionnaire `d` de la même manière : `d[k] = v`.

— Si la clé `k` n'est pas déjà présente dans `d`, les données sont ajoutées.

— Si la clé `k` est déjà présente dans `d`, la valeur associée) `k` est modifiée en `v`.

```
moyenne["toto"] = 13 #on modifie la moyenne de toto a 13
moyenne["tata"] = 12 #on ajoute un eleve tata de moyenne 12
```

Il est cependant impossible de modifier une clé particulière.

Pour supprimer une donnée de clé `k` dans un dictionnaire `d` on utilise la synatxe `del d[k]`.

```
del moyenne["toto"] #on retire toto (et sa note) du dictionnaire
```

3.6 Doublons

Les dictionnaires autorisent les doublons de valeurs mais pas de clé :

```
moyenne = {} #s est un dictionnaire vide
moyenne["toto"] = 14.5
moyenne["toto"] = 13.5
moyenne["titi"] = 14.5
moyenne["tutu"] = 9.5
moyenne["tata"] = 9.5
print(moyenne)
```

3.7 Dictionnaires : conclusion

Les dictionnaires sont des structures plus souples que les tableaux mais fonctionnent de la même manière. On les utilise pour organiser des données sous la forme d'une association clé, valeur (un peu comme une fonction ayant un nombre fini d'antécédents possibles).

Spécification	Indéxé	Clé	Mutable	Modifiable	Doublons
Tableaux	oui	indices (entiers)	oui	oui	oui
Ensembles	non	-	-	oui	non
n-uplets	oui	indices (entiers)	non	non	oui
Dictionnaires	oui	ce qu'on veut	oui	oui	oui