

Support de cours

PROGRAMMATION EN PYTHON



CHAPITRE III : STRUCTURE DE DONNÉES ET COLLECTIONS

Plan

- **INTRODUCTION AUX STRUCTURES DE DONNÉES**
- **STRUCTURES DE DONNÉES INTÉGRÉES**
 - **LISTES (LIST)**
 - **TUPLES (TUPLE)**
 - **ENSEMBLES (SET)**
 - **DICTIONNAIRES (DICT)**
 - **PARCOURS ET ITÉRATIONS SUR LES COLLECTIONS**
- **STRUCTURES DE DONNÉES SPECIFIQUES DÉFINIES PAR L'UTILISATEUR**
 - **TABLEAUX AVANCÉS**
 - **PILE**
 - **FILE**
 - **LISTES CHAÎNÉES**
 - **ARBRES**
 - **GRAPHES**
- **EXERCICES**

Introduction

Une structure de données est un moyen d'organiser et de stocker des informations afin de faciliter leur manipulation et leur accès.

→ Ces structures se divisent en deux grandes catégories :

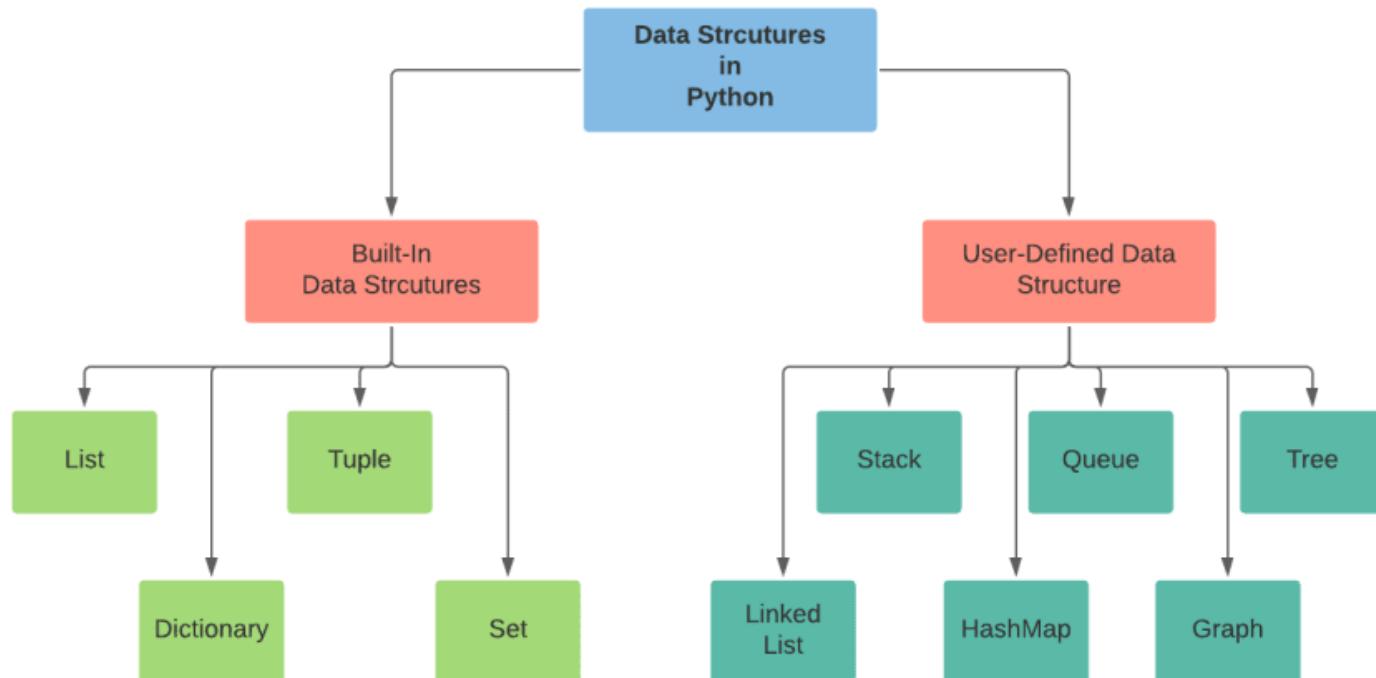
- **Structures intégrées (Built-in Data Structures)** : directement fournies par Python, elles sont optimisées et faciles à utiliser. Exemples : listes, tuples, ensembles et dictionnaires.
- **Structures définies par l'utilisateur (User-Defined Data Structures)** : créées manuellement pour répondre à des besoins spécifiques, elles incluent les piles, files, listes chaînées, arbres et graphes.

Chaque structure a ses propres avantages et limites, influençant la performance des algorithmes et l'optimisation des ressources. Ce chapitre explore en détail ces structures et leurs applications en Python.

Introduction

Les structures de données sont essentielles pour :

- **Stocker et organiser** efficacement les informations.
- Effectuer des **recherches** et des **modifications** rapidement.
- **Optimiser** l'exécution des programmes en choisissant la structure la plus adaptée.



Structures de données intégrées

Python propose plusieurs structures intégrées qui permettent de stocker et manipuler efficacement les données. Parmi elles, nous retrouvons :

- Listes (**list**) : Collection ordonnée et modifiable d'éléments.
- Tuples (**tuple**) : Collection ordonnée et immuable.
- Ensembles (**set**) : Collection non ordonnée sans doublons.
- Dictionnaires (**dict**) : Collection clé-valeur rapide et modifiable.



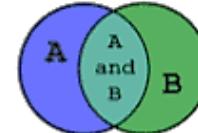
List



Tuple



Dictionary



Sets

→ Les listes

Une **liste** est une structure de données ordonnée et **modifiable** qui peut contenir des éléments de types différents. Les listes sont très flexibles et permettent d'ajouter, de modifier ou de supprimer des éléments facilement.

□ Création d'une Liste

En Python, on crée une liste en plaçant des éléments entre crochets [], séparés par des virgules.

```
# Liste vide
ma_liste = []

# Liste d'entiers
nombres = [1, 2, 3, 4, 5]

# Liste mixte (différents types de données)
mixte = [10, "Python", 3.14, True]
```

Structures de données intégrées

→ Les listes

□ Accès aux éléments

On accède aux éléments d'une liste en utilisant leur **indice (position)**.

```
nombres = [10, 20, 30, 40, 50]  
  
print(nombres[0]) # Premier élément  
print(nombres[-1]) # Dernier élément
```

```
10  
50
```

□ Modification des éléments

Les listes sont mutables, ce qui signifie que leurs éléments peuvent être modifiés.

```
nombres = [10, 20, 30]  
nombres[1] = 25 # Modifier l'élément à l'index 1  
print(nombres)
```

```
[10, 25, 30]
```

→ Les listes

□ Opérations de Base sur les Listes

→ Ajouter des éléments

- `append(x)` : ajoute un élément à la fin
- `insert(i, x)` : insère un élément à un index donné

Exemple

```
nombres = [1, 2, 3]
nombres.append(4) # Ajoute 4 à la fin
print(nombres)

nombres.insert(1, 10) # Insère 10 à l'index 1
print(nombres)

[1, 2, 3, 4]
[1, 10, 2, 3, 4]
```

→ Les listes

□ Opérations de base sur les Listes

→ Supprimer des éléments

- **remove (x)** : supprime la première occurrence de x
- **pop (i)** : supprime l'élément à l'index i (ou le dernier si non spécifié)

Exemple

```
nombres = [1, 2, 3, 2, 4]
nombres.remove(2) # Supprime la valeur 2
print(nombres)

nombres.pop() # Supprime le dernier élément
print(nombres)

[1, 3, 2, 4]
[1, 3, 2]
```

→ Les listes

□ Opérations de base sur les Listes

→ Trier et inverser

- `sort()` : Trie la liste en place (ordre croissant par défaut).
- `sorted(liste)` : Retourne une nouvelle liste triée (ordre croissant par défaut).
- `sort(reverse=True)` : Trie la liste en place en ordre décroissant.
- `sorted(liste, reverse=True)` : Retourne une nouvelle liste triée en ordre décroissant.
- `reverse()` : Inverse l'ordre des éléments en place.
- `reversed(liste)` : Retourne un itérateur inversé.

➤ *Différence clé*

- `sort()` et `reverse()` modifient la liste originale.
- `sorted()` et `reversed()` créent une nouvelle liste ou un itérateur.

Structures de données intégrées

→ Les listes

□ Opérations de base sur les Listes

→ Trier et inverser

Exemple 1

```
nombres = [5, 2, 9, 1]
nombres.sort()
print(f"Tri avec sort() : {nombres}")
# Remise à l'état original
nombres.reverse()
print(f"Liste après reverse() (retour à l'état original) : {nombres}")
nombres.sort(reverse=True)
print(f"Tri décroissant avec sort(reverse=True) : {nombres}")
# Remise à l'état original
nombres.reverse()
print(f"Liste après reverse() (retour à l'état original) : {nombres}")
```

```
Tri avec sort() : [1, 2, 5, 9]
Liste après reverse() (retour à l'état original) : [9, 5, 2, 1]
Tri décroissant avec sort(reverse=True) : [9, 5, 2, 1]
Liste après reverse() (retour à l'état original) : [1, 2, 5, 9]
```

Structures de données intégrées

→ Les listes

□ Opérations de base sur les Listes

→ Trier et inverser

Exemple 2

```
nombres = [5, 2, 9, 1]
# Tri avec sorted() (nouvelle liste)
triée = sorted(nombres)
print(f"Tri avec sorted() : {triée}")
print(f"Liste originale après sorted() : {nombres}")

# Tri décroissant avec sorted(reverse=True)
triée_decroissant = sorted(nombres, reverse=True)
print(f"Tri décroissant avec sorted(reverse=True) : {triée_decroissant}")
print(f"Liste originale après sorted(reverse=True) : {nombres}")

Tri avec sorted() : [1, 2, 5, 9]
Liste originale après sorted() : [5, 2, 9, 1]
Tri décroissant avec sorted(reverse=True) : [9, 5, 2, 1]
Liste originale après sorted(reverse=True) : [5, 2, 9, 1]
```

→ Les listes

□ Autres opérations courantes

- **Concaténation (+)** : Fusionne deux listes.
- **Répétition (*)** : Répète les éléments n fois.
- **Slicing** : `liste[start:stop:step]`
: Extrait une sous-liste.
- **Appartenance (in)** : Vérifie si x est présent.

Exemple

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]

fusion = liste1 + liste2
print(f"Concaténation : {fusion}")

répetition = liste1 * 2
print(f"Répétition : {répetition}")

sous_liste = fusion[1:5:2]
print(f"Slicing : {sous_liste}")

présence = 3 in fusion
print(f"Présence de 3 ? {présence}")

Concaténation : [1, 2, 3, 4, 5, 6]
Répétition : [1, 2, 3, 1, 2, 3]
Slicing : [2, 4]
Présence de 3 ? True
```

→ Les listes

❑ Autres Fonctions Utiles

- **len(liste)** : Retourne le nombre d'éléments dans la liste.
- **count(x)** : Retourne le nombre d'occurrences de x dans la liste.
- **index(x)** : Retourne l'index de la première occurrence de x.
- **extend(liste2)** : Ajoute les éléments de liste2 à la liste.
- **copy()** : Retourne une copie de la liste.
- **clear()** : Supprime tous les éléments de la liste.
- **copy()** : Retourne une copie de la liste.
- **any(liste)** : Retourne **True** si au moins un élément est vrai.
- **all(liste)** : Retourne **True** si tous les éléments sont vrais.
- **enumerate(liste)** : Retourne les index et valeurs sous forme de tuples.
- **zip(liste1, liste2, ...)** : Associe les éléments de plusieurs listes.

Structures de données intégrées

→ Les listes

```
l = [10, 20, 30, 20, 40, 50, 0]
print("Longueur de la liste:", len(l))
print("Occurrences de 20:", l.count(20))
print("Index de 30:", l.index(30))

l2 = [60, 70]
l.extend(l2)
print("Liste après extension:", l)

copie_liste = l.copy()
print("Copie de la liste:", copie_liste)

l.clear()
print("Liste après clear:", l)

l_test = [0, 0, 1, 0]
print("any(l_test):", any(l_test)) # True (car 1 est vrai)
print("all(l_test):", all(l_test)) # False (car il y a des 0)

l_enum = ["a", "b", "c"]
print("Enumerate:", list(enumerate(l_enum)))

noms = ["Imane", "Amina", "Asmaa"]
ages = [18, 19, 20]
couleurs = ["Rouge", "Bleu", "Vert"]

liste_zip = list(zip(noms, ages, couleurs))
print("Zip des listes:", liste_zip)

Longueur de la liste: 7
Occurrences de 20: 2
Index de 30: 2
Liste après extension: [10, 20, 30, 20, 40, 50, 0, 60, 70]
Copie de la liste: [10, 20, 30, 20, 40, 50, 0, 60, 70]
Liste après clear: []
any(l_test): True
all(l_test): False
Enumerate: [(0, 'a'), (1, 'b'), (2, 'c')]
Zip des listes: [('Imane', 18, 'Rouge'), ('Amina', 19, 'Bleu'), ('Asmaa', 20, 'Vert')]
```

→ Les listes

□ Avantages des Listes

- **Flexibilité** : Peut contenir différents types de données
- **Mutabilité** : Peut être modifiée après création
- Facilité **d'ajout et de suppression**

□ Inconvénients des Listes

- **Performances** : Moins efficace que certaines structures spécifiques pour certaines opérations
- **Occupation mémoire** : Peut être plus gourmande en mémoire que d'autres structures

→ Les tuples

Les **tuples** sont des structures de données similaires aux **listes**, mais **immuables** (on ne peut pas modifier leur contenu après leur création). Ils sont utilisés lorsqu'on veut stocker des données qui ne doivent pas être altérées.

□ Déclaration d'un Tuple

Un tuple se définit en plaçant des valeurs entre parenthèses (), séparées par des virgules :

```
mon_tuple = (1, 2, 3, 4, 5)
print(mon_tuple)
```

```
(1, 2, 3, 4, 5)
```

Remarque : Il est possible de créer un tuple sans parenthèses :

```
mon_tuple = 1, 2, 3, 4, 5
print(mon_tuple)
```

```
(1, 2, 3, 4, 5)
```

→ Les tuples

□ Déclaration d'un Tuple

Si le tuple contient un seul élément, il faut ajouter une virgule, sinon Python ne le reconnaît pas comme un tuple :

```
tuple_unique = (5,)  
print(type(tuple_unique))
```

```
tuple_faux = (5)  
print(type(tuple_faux))
```

```
<class 'tuple'>  
<class 'int'>
```

La fonction `type()` permet de connaître le type d'une variable en Python

Structures de données intégrées

→ Les tuples

□ Accès aux éléments

On accède aux éléments d'une liste en utilisant leur **indice (position)**.

```
mon_tuple = ('a', 'b', 'c', 'd')
```

```
print(mon_tuple[0])
print(mon_tuple[2])
```

```
a
c
```

On peut aussi utiliser l'**indexation négative** :

```
print(mon_tuple[-1]) # dernier élément
print(mon_tuple[-2]) # avant-dernier élément
```

```
d
c
```

→ Les tuples

□ Propriétés des tuples (Immutabilité)

Les tuples sont **immuables**, c'est-à-dire qu'une fois créés, on ne peut **ni ajouter, ni modifier, ni supprimer** un élément.

□ Pourquoi utiliser un tuple ?

- **Sécurité** : Les données ne peuvent pas être modifiées accidentellement.
- **Performance** : Les tuples sont plus rapides que les listes.
- **Utilisation en clés de dictionnaire** : Un tuple peut être une clé dans un dictionnaire (contrairement aux listes). *Exemple:*

```
coordonnees = (33.5731, -7.5898)
dico = {coordonnees: "Casablanca"}
print(dico[coordonnees])
```

```
Casablanca
```

Structures de données intégrées

→ Les tuples

□ Fonctions intégrées pour les tuples

- **len(tuple)** : Retourne le nombre d'éléments dans le tuple.
- **count(x)**: Retourne le nombre d'occurrences de x dans le tuple.
- **index(x)**: Retourne l'index de la première occurrence de x.
- **min(tuple)** :Retourne la plus petite valeur du tuple.
- **max(tuple)** : Retourne la plus grande valeur du tuple.
- **sum(tuple)**: Retourne la somme des éléments (si numériques).
- **reversed(tuple)** : Retourne un itérateur inversé du tuple (mais ne modifie pas le tuple d'origine).
- **sorted(tuple)** : Retourne une nouvelle liste triée (et non un tuple).

Remarque : Les tuples n'ont **pas** de méthodes comme **append()** , **remove()**, **sort()**, **reverse()**, contrairement aux listes.

Structures de données intégrées

→ Les tuples

□ Fonctions intégrées pour les tuples

Exemple

```
t = (5, 10, 15, 10, 20, 25)

print("Longueur du tuple:", len(t))
print("Occurrences de 10:", t.count(10))
print("Index de 15:", t.index(15))
print("Valeur minimale:", min(t))
print("Valeur maximale:", max(t))
print("Somme des éléments:", sum(t))

liste = [1, 2, 3]
tuple_converti = tuple(liste)
print("Tuple converti depuis une liste:", tuple_converti)

tuple_inverse = tuple(reversed(t))
print("Tuple inversé:", tuple_inverse)

tuple_trie = sorted(t)
print("Tuple trié (résultat sous forme de liste):", tuple_trie)

tuple_trie_tuple = tuple(sorted(t))
print("Tuple trié (converti en tuple):", tuple_trie_tuple)

Longueur du tuple: 6
Occurrences de 10: 2
Index de 15: 2
Valeur minimale: 5
Valeur maximale: 25
Somme des éléments: 85
Tuple converti depuis une liste: (1, 2, 3)
Tuple inversé: (25, 20, 10, 15, 10, 5)
Tuple trié (résultat sous forme de liste): [5, 10, 10, 15, 20, 25]
Tuple trié (converti en tuple): (5, 10, 10, 15, 20, 25)
```

Structures de données intégrées

→ Les tuples

□ Conversion entre tuple et liste

- **list(tuple)** : Convertit un tuple en liste (permet la modification des éléments).
- **tuple(liste)** : Convertit une liste en tuple (rend les éléments immuables).

Exemple

```
t = (10, 20, 30)
l = list(t) # Conversion en liste
l.append(40) # Ajout d'un élément
print("Liste après modification:", l)

Liste après modification: [10, 20, 30, 40]
```

```
l = [1, 2, 3, 4]
t = tuple(l) # Conversion en tuple
print("Tuple après conversion:", t)

Tuple après conversion: (1, 2, 3, 4)
```

□ Opérations possibles sur les tuples

- **Concaténation (+)** : Combine deux tuples en un seul.
- **Répétition (*)** : Répète le tuple plusieurs fois.
- **Vérification d'appartenance (in)** : Vérifie si un élément est présent dans le tuple.
- **Slicing ([:])** : Extrait une partie du tuple.

Structures de données intégrées

→ Les tuples

□ Opérations possibles sur les tuples

Exemple

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)

t_concat = t1 + t2
print("Concaténation:", t_concat)
t_repetition = t1 * 3
print("Répétition:", t_repetition)
print("Vérification d'appartenance (2 dans t1) :", 2 in t1)
print("Vérification d'appartenance (10 dans t1) :", 10 in t1)

print("Slicing t_concat[1:4]:", t_concat[1:4]) # éléments de l'index 1 à 3
print("Slicing t_concat[:3]:", t_concat[:3]) # du début jusqu'à l'index 2
print("Slicing t_concat[3:]:", t_concat[3:]) # de l'index 3 jusqu'à la fin
print("Slicing t_concat[::-1]:", t_concat[::-1]) # inversion du tuple

Concaténation: (1, 2, 3, 4, 5, 6)
Répétition: (1, 2, 3, 1, 2, 3, 1, 2, 3)
Vérification d'appartenance (2 dans t1) : True
Vérification d'appartenance (10 dans t1) : False
Slicing t_concat[1:4]: (2, 3, 4)
Slicing t_concat[:3]: (1, 2, 3)
Slicing t_concat[3:]: (4, 5, 6)
Slicing t_concat[::-1]: (6, 5, 4, 3, 2, 1)
```

Structures de données intégrées

→ Tuples Vs. Listes

Caractéristique	Tuple	Liste
Mutabilité	Immutable	Modifiable
Performance	Plus rapide	Moins rapide
Mémoire	Moins gourmand	Plus gourmand
Clé de dictionnaire	Possible	Impossible
Méthodes disponibles	Peu (count, index)	NOMBREUSES (append, remove, sort...)

→ Les ensembles

Un **ensemble** (« **set** ») en Python est une collection non ordonnée d'éléments uniques. Cela signifie que :

- Un ensemble ne peut pas contenir de doublons.
- Les éléments d'un ensemble ne sont pas indexés et n'ont pas d'ordre fixe.
- Les ensembles sont mutables, mais leurs éléments doivent être immuables (ex. nombres, chaînes, tuples, mais pas des listes ou d'autres ensembles).

□ Déclaration d'un Ensemble

Un ensemble se définit en plaçant des valeurs entre accolades {}, séparées par des virgules :

```
mon_ensemble = {1, 2, 3, 4, 5}  
print(mon_ensemble)  
  
{1, 2, 3, 4, 5}
```

➤ Si un ensemble contient des **doublons**, ils sont **automatiquement supprimés** :

```
ensemble_avec_doublons = {1, 2, 2, 3, 4, 4}  
print(ensemble_avec_doublons)  
  
{1, 2, 3, 4}
```

Structures de données intégrées

→ Les ensembles

□ Déclaration d'un Ensemble

Avec la fonction set() (alternative aux accolades) :

```
ensemble2 = set([1, 2, 3, 4]) # Conversion d'une liste en ensemble
print(ensemble2)

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

➤ Cas d'un ensemble vide :

```
ensemble_vide = set()
print(ensemble_vide)

set()
```

Ne pas utiliser {} car cela crée un dictionnaire vide

Attention : {} tout seul crée un dictionnaire vide, pas un ensemble. Pour un ensemble vide, il faut **obligatoirement** utiliser **set()**

Structures de données intégrées

→ Les ensembles

□ Accès aux éléments

Les ensembles **ne supportent pas l'indexation ni le slicing** comme les listes. Cependant, il est possible d'itérer sur un ensemble avec une boucle **for** :

```
mon_ensemble = {10, 20, 30, 40}
for element in mon_ensemble:
    print(element)
```

```
40
10
20
30
```

➤ Si l'on veut vérifier la présence d'un élément, on utilise l'opérateur **in** :

```
print(20 in mon_ensemble)
print(50 in mon_ensemble)
```

```
True
False
```

→ Les ensembles

□ Opérations de base

Les ensembles étant **non indexés**, on ne peut pas modifier un élément spécifique directement. Cependant, on peut :

→ Ajouter des éléments et les mettre à jour

- **add(x)** : ajoute un élément à l'ensemble.
- **update(iterable)** : ajoute plusieurs éléments.

Exemple

```
mon_ensemble = {1, 2, 3}
mon_ensemble.add(4) # Ajoute un seul élément
mon_ensemble.update([5, 6, 7]) # Ajoute plusieurs éléments
print(mon_ensemble)

{1, 2, 3, 4, 5, 6, 7}
```

→ Les ensembles

□ Opérations de base

→ Supprimer des éléments

- **remove(x)** : supprime un élément (erreur si absent).
- **discard(x)** : supprime un élément (ne génère pas d'erreur si absent).

Exemple

```
mon_ensemble = {1, 2, 3, 4, 5, 6, 7}
mon_ensemble.remove(2)
mon_ensemble.discard(8) # Ne fait rien si l'élément 8 n'existe pas
print(mon_ensemble)

{1, 3, 4, 5, 6, 7}
```

Structures de données intégrées

→ Les ensembles

□ Opérations ensemblistes

- **union()** : permet de retourner un nouvel ensemble contenant tous les éléments des deux ensembles.

Exemple

```
ensemble1 = {1, 2, 3}
ensemble2 = {3, 4, 5}
uniEn = ensemble1.union(ensemble2)
print(uniEn)

{1, 2, 3, 4, 5}
```

- **intersection()** : permet de retourner un nouvel ensemble contenant les éléments communs.

Exemple

```
interEn = ensemble1.intersection(ensemble2)
print(interEn)

{3}
```

→ Les ensembles

□ Opérations ensemblistes

- **difference()** : permet de retourner un ensemble contenant les éléments de *ensemble1* qui ne sont pas dans *ensemble2*

Exemple

```
diffEn = ensemble1.difference(ensemble2)
print(diffEn)

{1, 2}
```

- **symmetric_difference()** : permet de retourner un ensemble contenant les éléments qui ne sont pas communs aux deux ensembles.

Exemple

```
diff_sym = ensemble1.symmetric_difference(ensemble2)
print(diff_sym)

{1, 2, 4, 5}
```

→ Les ensembles

□ Suppression des Doublons avec un Ensemble

Une des utilisations courantes des ensembles est l'élimination des doublons d'une liste. Cela est possible grâce à la fonction **set()**, qui permet de convertir une liste en ensemble, supprimant ainsi automatiquement les doublons.

Exemple

```
liste_avec_doublons = [1, 2, 2, 3, 4, 4, 5]
liste_sans_doublons = list(set(liste_avec_doublons))
print(liste_sans_doublons)

[1, 2, 3, 4, 5]
```

→ Les ensembles

□ Autres fonctions utiles

- **len(set)**: retourne le nombre d'éléments dans un ensemble.
- **copy()**: crée une copie de l'ensemble.
- **clear()**: vide complètement un ensemble.
- **issubset(set)** : vérifie si un ensemble est inclus dans un autre.
- **issuperset(set)** : vérifie si un ensemble contient tous les éléments d'un autre.
- **isdisjoint(set)** : vérifie si deux ensembles n'ont aucun élément en commun.

Structures de données intégrées

→ Les ensembles

□ Autres fonctions utiles

Exemple

```
A = {1, 2, 3}
B = {3, 4, 5}
C = {6, 7}

print("Taille de A :", len(A))
copie_A = A.copy()
print("Copie de A :", copie_A)

A.clear()
print("A après clear() :", A)

print("B est un sous-ensemble de {1, 2, 3, 4, 5} ?", B.issubset({1, 2, 3, 4, 5}))
print("{1, 2, 3, 4, 5} est un sur-ensemble de B ?", {1, 2, 3, 4, 5}.issuperset(B))
print("B et C sont-ils disjoints ?", B.isdisjoint(C))

Taille de A : 3
Copie de A : {1, 2, 3}
A après clear() : set()
B est un sous-ensemble de {1, 2, 3, 4, 5} ? True
{1, 2, 3, 4, 5} est un sur-ensemble de B ? True
B et C sont-ils disjoints ? True
```

Structures de données intégrées

→ Les dictionnaires

Un dictionnaire est une structure de données qui associe des **clés** à des **valeurs**. Il se définit avec des accolades {} et des paires **clé : valeur** séparées par des virgules.

```
mon_dict = {"nom": "Ilham", "age": 23, "ville": "Beni Mellal"}  
print("Dictionnaire :", mon_dict)  
  
Dictionnaire : {'nom': 'Ilham', 'age': 23, 'ville': 'Beni Mellal'}
```

- Accéder à une valeur par sa clé :

```
print("Nom :", mon_dict["nom"])  
print("Age :", mon_dict.get("age"))  
  
Nom : Ilham  
Age : 23
```

- Si une clé n'existe pas, **dict[key]** lève une erreur, tandis que **get(key, valeur_defaut)** retourne une valeur par défaut :

```
print("Pays :", mon_dict.get("pays", "Non défini"))  
  
Pays : Non défini
```

Structures de données intégrées

→ Les dictionnaires

□ Ajouter une clé-valeur

```
mon_dict["profession"] = "Ingénieur"
print("Après ajout :", mon_dict)
```

```
Après ajout : {'nom': 'Ilham', 'age': 23, 'ville': 'Beni Mellal', 'profession': 'Ingénieur'}
```

□ Modifier une valeur existante

```
mon_dict["age"] = 26
print("Après modification :", mon_dict)
```

```
Après modification : {'nom': 'Ilham', 'age': 26, 'ville': 'Beni Mellal', 'profession': 'Ingénieur'}
```

Structures de données intégrées

→ Les dictionnaires

□ Supprimer une clé

- **pop(key)** : supprime la clé et retourne sa valeur.
- **del dict[key]** : supprime la clé sans retour de valeur.

Exemple

```
age = mon_dict.pop("age")
print("Age supprimé :", age)
print("Dictionnaire après suppression :", mon_dict)

Age supprimé : 26
Dictionnaire après suppression : {'nom': 'Ilham', 'ville': 'Beni Mellal', 'profession': 'Ingénieur'}
```

```
del mon_dict["ville"]
print("Dictionnaire après suppression de 'ville' :", mon_dict)

Dictionnaire après suppression de 'ville' : {'nom': 'Ilham', 'profession': 'Ingénieur'}
```

→ Les dictionnaires

□ Fonctions utiles

- **keys()** : retourne une vue des clés.
- **values()** : retourne une vue des valeurs.
- **items()** : retourne une vue des paires (clé, valeur).
- **copy()** : crée une copie du dictionnaire.
- **clear()** : vide le dictionnaire.

Exemple

```
print("Clés :", mon_dict.keys())
print("Valeurs :", mon_dict.values())
print("Paires clé-valeur :", mon_dict.items())

Clés : dict_keys(['nom', 'profession'])
Valeurs : dict_values(['Ilham', 'Ingénieur'])
Paires clé-valeur : dict_items([('nom', 'Ilham'), ('profession', 'Ingénieur')])
```

Structures de données intégrées

→ Les Parcours et Itérations sur les Collections

□ Parcours d'une Liste

```
ma_liste = [1, 2, 3, 4]
for element in ma_liste:
    print("Élément de la liste :", element)
```

```
Élément de la liste : 1
Élément de la liste : 2
Élément de la liste : 3
Élément de la liste : 4
```

□ Parcours d'un Tuple

```
mon_tuple = ("a", "b", "c")
for valeur in mon_tuple:
    print("Valeur du tuple :", valeur)
```

```
Valeur du tuple : a
Valeur du tuple : b
Valeur du tuple : c
```

Structures de données intégrées

→ Les Parcours et Itérations sur les Collections

□ Parcours d'un Ensemble

```
mon_set = {10, 20, 30}
for valeur in mon_set:
    print("Valeur de l'ensemble :", valeur)
```

```
Valeur de l'ensemble : 10
Valeur de l'ensemble : 20
Valeur de l'ensemble : 30
```

□ Parcours d'un Dictionnaire

```
mon_dict = {"nom": "Imane", "age": 23, "ville": "Casablanca"}
for cle, valeur in mon_dict.items():
    print(f"{cle} : {valeur}")
```

```
nom : Imane
age : 23
ville : Casablanca
```

Structures de données définies par l'utilisateur

Les structures de données définies par l'utilisateur ne sont pas directement intégrées à Python, mais elles peuvent être créées en **utilisant des modules spécialisés** pour des performances optimisées. Il est aussi possible d'utiliser des **listes**, bien que leur efficacité puisse être limitée selon les opérations effectuées.

→ Les tableaux avancés

Un **tableau** est une structure qui stocke plusieurs éléments du même type dans une mémoire contiguë. Contrairement aux listes Python, un tableau impose un type unique pour ses éléments.

En Python, il n'existe pas de type natif **Array** comme en C ou Java, mais plusieurs alternatives sont possibles :

□ Avec la bibliothèque **array** (pour un tableau typé)

```
import array
tableau = array.array('i', [1, 2, 3, 4, 5]) # Tableau d'entiers ('i' signifie integer)
```

□ Avec **NumPy** (pour des tableaux plus performants)

```
import numpy as np
tableau = np.array([1, 2, 3, 4, 5])
```

→ Les tableaux avancés

□ Array (`array.array`)

- **Avantages** : Stocke uniquement un type spécifique d'éléments.
- **Inconvénients** : Moins utilisé que `list` en Python.

□ NumPy (`numpy.array`)

- **Avantages** : Optimisé pour les calculs scientifiques et les grandes données.
- **Inconvénients** : Nécessite l'installation de NumPy.

Structures de données définies par l'utilisateur

→ Les piles

Une **pile** est une structure de données qui suit le principe **LIFO** (Last In, First Out), c'est-à-dire que le dernier élément ajouté est le premier à être retiré. Python ne possède pas de type **stack** intégré, mais on peut l'implémenter de différentes manières :

- ❑ **collections.deque** : Fournit une pile performante avec **append()** et **pop()**.

```
from collections import deque

# Création d'une pile avec deque
stack = deque()
stack.append(1) # Ajout d'un élément
stack.append(2)
print(stack.pop()) # Suppression et affichage du dernier élément

2
```

- ❑ **queue.LifoQueue** : Offre une gestion sécurisée des piles pour les applications multi-threads.

```
from queue import LifoQueue

# Création d'une pile avec LifoQueue
stack = LifoQueue()
stack.put(1) # Ajout d'un élément
stack.put(2)
print(stack.get()) # Suppression et affichage du dernier élément

2
```

Structures de données définies par l'utilisateur

→ Les piles

- ❑ `collections.deque` : Fournit une pile performante avec `append()` et `pop()`.

```
from collections import deque

# Création d'une pile avec deque
stack = deque()
stack.append(1) # Ajout d'un élément
stack.append(2)
print(stack.pop()) # Suppression et affichage du dernier élément

2
```

- ❑ `queue.LifoQueue` : Offre une gestion sécurisée des piles pour les applications multi-threads.

```
from queue import LifoQueue

# Création d'une pile avec LifoQueue
stack = LifoQueue()
stack.put(1) # Ajout d'un élément
stack.put(2)
print(stack.get()) # Suppression et affichage du dernier élément

2
```

Structures de données définies par l'utilisateur

→ Les piles

- **Les listes** : On peut également implémenter les piles en utilisant simplement les listes, avec **append()** pour ajouter un élément et **pop()** pour le retirer

```
pile = []
# Empiler des éléments
pile.append(10)
pile.append(20)
pile.append(30)
# Dépiler un élément
élément = pile.pop() # Retire et retourne le dernier élément ajouté (30)
print(f"Après suppression de {élément}, la pile devient : {pile}")
# Voir le sommet de la pile
sommet = pile[-1] if pile else "Pile vide"
print(f"Sommet de la pile : {sommet}")
# Vérifier si la pile est vide
est_vide = len(pile) == 0
print(f"La pile est-elle vide ? {est_vide}")
# Obtenir la taille de la pile
taille = len(pile)
print(f"Taille de la pile : {taille}")

Après suppression de 30, la pile devient : [10, 20]
Sommet de la pile : 20
La pile est-elle vide ? False
Taille de la pile : 2
```

- **append(x)** : Ajoute x à la fin de la liste (sommet de la pile).
- **pop()** : Retire et retourne l'élément au sommet.
- **pile[-1]** : Permet d'accéder au sommet sans le retirer.
- **len(pile)** : Retourne le nombre d'éléments dans la pile.

Une liste convient bien aux petites piles, mais pour de meilleures performances, **collections.deque** est recommandé

Structures de données définies par l'utilisateur

→ Les files

Une file suit le principe **FIFO** (**First In, First Out**), où le premier élément ajouté est le premier à être retiré. Python ne possède pas de type queue intégré, mais on peut l'implémenter avec :

- ❑ `collections.deque` : est une alternative rapide pour une file.

```
from collections import deque
queue = deque()
queue.append("A")
queue.append("B")

# Retrait d'un élément
print(queue.popleft())
```

A

- ❑ `queue.Queue` est adapté aux applications multi-threadées.

```
from queue import Queue
# Création d'une file
queue = Queue()
# Ajout d'éléments
queue.put("A")
queue.put("B")
# Retrait d'un élément
print(queue.get())
```

A

Structures de données définies par l'utilisateur

→ Les files

- ❑ `queue.PriorityQueue` : permet de gérer des priorités.

```
from queue import PriorityQueue

# Création d'une file avec priorités
queue = PriorityQueue()

# Ajout d'éléments avec une priorité
queue.put((3, "Tâche normale"))
queue.put((1, "Tâche urgente"))
queue.put((2, "Tâche importante"))

# Retrait de l'élément le plus prioritaire
print(queue.get())

(1, 'Tâche urgente')
```

Structures de données définies par l'utilisateur

→ Les files

- **Les listes:** Avec une liste, on peut créer une file en utilisant append() pour ajouter un élément à la fin de la liste, et pop(0) pour retirer le premier élément.

```
file = []
# Ajouter des éléments à la file
file.append(10)
file.append(20)
file.append(30)
# Retirer un élément de la file
element = file.pop(0) # Retire et retourne le premier élément ajouté
print(f"Après suppression de {element}, la file devient : {file}")

# Voir le premier élément de la file
premier = file[0] if file else "File vide"
print(f"Premier élément de la file : {premier}")

# Vérifier si la file est vide
est_vide = len(file) == 0
print(f"La file est-elle vide ? {est_vide}")

# Obtenir la taille de la file
taille = len(file)
print(f"Taille de la file : {taille}")

Après suppression de 10, la file devient : [20, 30]
Premier élément de la file : 20
La file est-elle vide ? False
Taille de la file : 2
```

- **append(x)** : Ajoute x à la fin de la liste (à la fin de la file).
- **pop(0)** : Retire et retourne le premier élément de la liste (le premier élément ajouté, donc selon le principe FIFO).
- **file[0]** : Permet d'accéder au premier élément sans le retirer.
- **len(file)** : Retourne la taille actuelle de la file.

→ Les listes chainées

Une **liste chaînée** est une structure de données composée de nœuds, où chaque nœud contient :

- **Une donnée** (valeur).
- **Un lien vers le nœud suivant.**

Elle suit généralement le principe FIFO (First In, First Out) et permet une gestion dynamique de la mémoire.

□ Types de Listes Chaînées

→ **Liste chaînée simple:** Chaque nœud contient un lien vers le suivant.

Exemple : `node1 -> node2 -> node3 -> None.`

→ **Liste chaînée double :** Chaque nœud contient un lien vers le suivant et un lien vers le précédent.

Exemple : `None <- node1 <-> node2 <-> node3 -> None.`

→ **Liste circulaire** : Le dernier nœud pointe vers le premier, formant une boucle.

Exemple : `node1 -> node2 -> node3 -> node1 (boucle) .`

→ Les listes chainées

□ Déclaration d'une liste chaînée

Il existe différentes manières de déclarer une liste chaînée en Python :

- **Avec une classe** : Chaque nœud de la liste est une instance d'une classe avec un attribut pour la valeur et un pointeur vers le nœud suivant. Cette approche est flexible et utilisée dans des listes chaînées dynamiques.
- **Avec un dictionnaire** : On représente chaque nœud par un dictionnaire avec la valeur et un pointeur vers le nœud suivant. C'est une méthode simple, mais moins flexible qu'une classe.
- **Avec des listes imbriquées** : Chaque nœud est une liste contenant la valeur et un pointeur (index) vers le nœud suivant. C'est simple mais difficile à gérer dans de grandes structures.
- **Avec des tuples** : Chaque nœud est un tuple contenant la valeur et un pointeur vers le nœud suivant. C'est similaire aux listes imbriquées mais plus rigide à cause de l'immuabilité des tuples.

Structures de données définies par l'utilisateur

→ Les listes chainées

Exemple: Déclaration d'une liste chaînée avec un dictionnaire

```
node1 = {"value": 10, "next": None} # Premier nœud (valeur 10, pas de suivant)
node2 = {"value": 20, "next": None} # Deuxième nœud (valeur 20)
node3 = {"value": 30, "next": None} # Troisième nœud (valeur 30)

# Chaînage des nœuds
node1["next"] = node2 # Le premier nœud pointe vers le deuxième
node2["next"] = node3 # Le deuxième nœud pointe vers le troisième

# Affichage de la liste chaînée
current_node = node1
while current_node:
    print(current_node["value"], end=" -> " if current_node["next"] else "\n")
    current_node = current_node["next"]

10 -> 20 -> 30
```

- Chaque nœud est représenté par un dictionnaire avec deux clés :
 - "value" : la valeur du nœud.
 - "next" : le pointeur vers le nœud suivant (initialement None).

→ Les arbres

Un **arbre** est une structure de données hiérarchique composée de nœuds reliés par des arêtes.

→ Chaque nœud peut avoir plusieurs enfants, sauf le nœud racine, qui est le point d'entrée de l'arbre.

□ Concepts de base

- **Nœud** : Un élément de l'arbre contenant une valeur.
- **Racine** : Le nœud principal, sans parent.
- **Enfant** : Un nœud connecté à un autre nœud supérieur (parent).
- **Parent** : Un nœud qui a un ou plusieurs enfants.
- **Feuille** : Un nœud sans enfant.
- **Hauteur** : Nombre de niveaux dans l'arbre.

→ Les arbres

□ Types d'arbres

- **Arbre général** : Chaque nœud peut avoir plusieurs enfants.
- **Arbre binaire** : Chaque nœud a au plus deux enfants (gauche et droit).
- **Arbre binaire de recherche (BST)** : Un arbre binaire où les valeurs sont organisées (gauche < racine < droite).
- **Arbre équilibré** : Un arbre où la différence de hauteur entre sous-arbres est minimale (ex : AVL, Rouge-Noir).

□ Déclaration d'un arbre

Il existe différentes manières de déclarer un arbre en Python :

→ **Avec une classe (approche orientée objet)** : Cette méthode utilise des objets pour représenter les nœuds d'un arbre. Chaque nœud est une instance d'une classe, avec des attributs pour la valeur du nœud, l'enfant gauche et l'enfant droit. Cette approche est idéale pour les arbres dynamiques, où l'on souhaite ajouter, supprimer ou modifier des nœuds de manière flexible.

→ Les arbres

□ Déclaration d'un arbre

Il existe différentes manières de déclarer un arbre en Python :

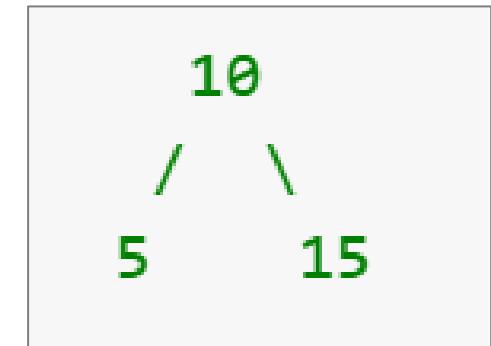
- **Avec un dictionnaire** : Un arbre peut aussi être représenté à l'aide d'un dictionnaire, où chaque nœud est une entrée contenant la valeur, l'enfant gauche et l'enfant droit. Cette méthode est plus simple et ne nécessite pas de classes, mais elle est moins flexible que l'approche orientée objet pour des arbres complexes.
- **Avec une liste imbriquée** : Cette méthode consiste à utiliser des listes pour représenter les nœuds de l'arbre. Chaque élément de la liste représente un nœud avec sa valeur et ses enfants sous forme de sous-listes. C'est une approche simple, mais elle peut devenir difficile à gérer lorsque l'arbre devient plus complexe.
- **Avec des tuples** : Un arbre peut aussi être représenté à l'aide de tuples, où chaque nœud est représenté par un tuple contenant la valeur du nœud et ses enfants. Cette méthode est similaire à la liste imbriquée, mais elle est souvent plus rigide car les tuples sont immuables.

Structures de données définies par l'utilisateur

→ Les arbres

Exemple 1: Représentation d'un arbre avec un dictionnaire

```
arbre = {
    "valeur": 10, # Racine de l'arbre avec la valeur 10
    "gauche": { # Branche gauche
        "valeur": 5, # Noeud gauche avec la valeur 5
        "gauche": None, # Pas d'enfant à gauche (feuille)
        "droite": None # Pas d'enfant à droite (feuille)
    },
    "droite": { # Branche droite
        "valeur": 15, # Noeud droit avec la valeur 15
        "gauche": None, # Pas d'enfant à gauche (feuille)
        "droite": None # Pas d'enfant à droite (feuille)
    }
}
```



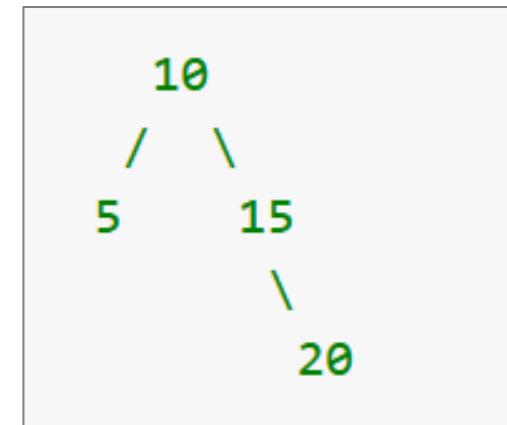
- C'est un **arbre binaire** car chaque nœud a au maximum 2 enfants (gauche et droite).
- C'est un **arbre binaire de recherche (BST)** car :
 - Le sous-arbre gauche contient des valeurs < 10.
 - Le sous-arbre droit contient des valeurs > 10.

Structures de données définies par l'utilisateur

→ Les arbres

Exemple 2: Représentation d'un arbre avec un dictionnaire

```
arbre = {
    "valeur": 10, # Racine de l'arbre avec la valeur 10
    "gauche": { # Branche gauche
        "valeur": 5, # Noeud gauche avec la valeur 5
        "gauche": None, # Pas d'enfant à gauche (feuille)
        "droite": None # Pas d'enfant à droite (feuille)
    },
    "droite": { # Branche droite
        "valeur": 15, # Noeud droit avec la valeur 15
        "gauche": None, # Pas d'enfant à gauche (feuille)
        "droite": { # Ajout d'un enfant à droite du nœud 15
            "valeur": 20,
            "gauche": None,
            "droite": None
        }
    }
}
```



→ Les graphes

Un **graphe** est une structure de données composée de nœuds (ou *sommets*) et de liens (ou *arêtes*) qui les relient.

➔ Il peut être orienté (les arêtes ont une direction) ou non orienté (les arêtes n'ont pas de direction). Un graphe peut également être pondéré si chaque arête a un poids ou un coût associé.

□ Types de graphes

→ **Graphe orienté** : Les arêtes ont une direction. Par exemple, si A pointe vers B, cela signifie qu'il y a une relation de A à B mais pas nécessairement de B à A.

→ **Graphe non orienté** : Les arêtes n'ont pas de direction. La relation entre A et B est bidirectionnelle.

→ **Graphe pondéré** : Chaque arête porte un poids, représentant souvent le coût ou la distance entre deux nœuds.

→ Les graphes

□ Déclaration d'un graphe

Il existe différentes manières de déclarer un graphe en Python :

- **Avec une liste de listes** (adjacence sous forme de liste) : Chaque sommet est représenté par un index de la liste, et les voisins sont stockés dans des sous-listes à cet index.
- **Avec un dictionnaire** (adjacence sous forme de dictionnaire) : Chaque sommet est une clé du dictionnaire, et ses voisins sont stockés dans une liste associée à cette clé.
- **Avec un dictionnaire de dictionnaires** (pour les graphes pondérés) : Chaque sommet est une clé, et les voisins sont représentés par un dictionnaire interne où la clé est le sommet voisin et la valeur est le poids de l'arête.
- **Avec des ensembles** (graphes non orientés): Chaque sommet est une clé dans un dictionnaire, et les voisins sont stockés dans un ensemble associé à cette clé.

Exercice 1

Créer une liste de nombres et effectuer les opérations suivantes :

- Ajouter un élément
- Supprimer un élément
- Afficher la liste mise à jour

Exercice 2

Créer un ensemble de nombres et effectuer les opérations suivantes :

- Ajouter un élément
- Supprimer un élément
- Afficher l'ensemble mis à jour

Exercice 3

Créer un dictionnaire avec des informations personnelles et :

- Ajouter une nouvelle clé-valeur
- Modifier une valeur existante
- Supprimer une clé
- Afficher les résultats

Exercices

Exercice 4

Écrire un programme en Python permettant de gérer une liste de courses en utilisant différentes structures de données intégrées. L'utilisateur pourra interagir avec le programme via un menu interactif. Le programme doit permettre d'effectuer les actions suivantes

1. **ajouter_article**(*liste_courses*, *article*, *quantite*): L'utilisateur entre le nom d'un article et une quantité. Chaque article doit être unique, si l'article est déjà présent, un message doit l'indiquer
2. **supprimer_article**(*liste_courses*, *article*): L'utilisateur entre le nom d'un article à supprimer. Si l'article n'existe pas, un message d'erreur doit être affiché
3. **afficher_liste**(*liste_courses*): Le programme doit afficher tous les articles et leurs quantités. Si la liste est vide, un message approprié doit s'afficher
4. **valider_liste**(*liste_courses*): Une fois la liste validée, elle ne pourra plus être modifiée. Les articles validés seront stockés dans une structure non modifiable
5. **quitter_programme**(*liste_courses*, *liste_validee*): Avant de quitter, le programme affichera un récapitulatif final des courses

Exercice 5

Créer un programme en Python pour la gestion des résultats d'un tournoi sportif. Les informations concernant les équipes et leurs scores sont stockées dans un dictionnaire, où chaque clé représente le nom d'une équipe et la valeur est un tuple contenant le nom de l'équipe ainsi que son score.

1. Trier les équipes par score de manière croissante (du score le plus bas au score le plus élevé).
2. Trier les équipes par score de manière décroissante (du score le plus élevé au score le plus bas).