



Programmation Python

Séance 2 — Structures de données

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

<https://ebtoure.sn/cours/programmation-python/chapitres/02-structures-de-donnees.html>

Version du 11 octobre 2026



Version en ligne

Programmation Python

— Séance 2

Structures de données

Dr. El Hadji Bassirou TOURÉ

Département de Mathématiques et Informatique, Faculté des Sciences et Techniques, Université Cheikh Anta Diop de Dakar

Objectifs de la séance

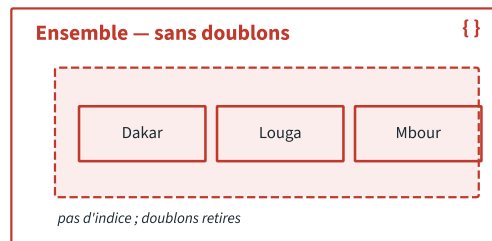
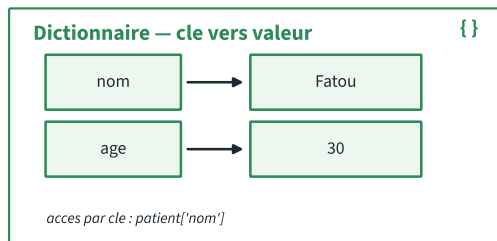
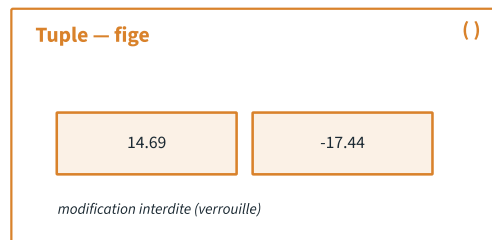
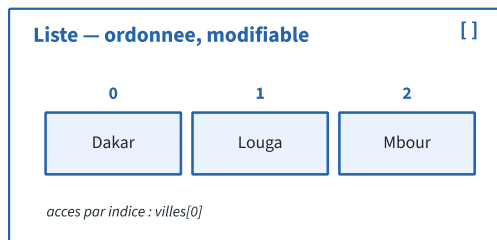
Contenu

Jusqu'ici, une variable contenait *une* valeur. On apprend à en regrouper **plusieurs** dans une même structure. Le choix de la bonne structure est une décision centrale en programmation : chacune a ses forces.

- **Listes** : collections ordonnées et modifiables.
- **Tuples** : collections figées.
- **Dictionnaires** : associations clé → valeur.
- **Ensembles** : collections de valeurs uniques.
- **Compréhensions** : construire une liste en une ligne.

L'entraînement se fait dans le **notebook**, avec une cellule « À vous » après chaque notion.

Panorama des quatre structures



Quatre façons de regrouper des valeurs : par position (liste, tuple), par clé (dictionnaire), ou en collection unique (ensemble).

Les listes

L'idée : une collection ordonnée et modifiable

L'idée en une phrase

Une **liste** regroupe des valeurs entre crochets `[ ]`, dans un **ordre**. On y accède par **position**, comme pour une chaîne (indice à partir de `0`, `-1` pour le dernier), et — nouveauté — on peut la **modifier** après création.

Une liste peut contenir n'importe quoi : des nombres, des textes, et même un mélange. C'est la structure la plus utilisée pour stocker une suite de données du même genre — une liste de villes, de notes, de patients.

Créer, accéder, découper

```
villes = ["Dakar", "Louga", "Mbour", "Kaolack"]  
print(villes)  
print("nombre :", len(villes))  
print("premiere :", villes[0])  
print("derniere :", villes[-1])  
print("tranche :", villes[1:3])
```

RÉSULTAT →

```
['Dakar', 'Louga', 'Mbour', 'Kaolack']  
nombre : 4  
premiere : Dakar  
derniere : Kaolack  
tranche : ['Louga', 'Mbour']
```

Modifier une liste

Remplacer, ajouter, insérer

`liste[i] = ...` remplace ; `.append(x)` ajoute à la fin ; `.insert(i, x)` insère à la position `i`.

```
villes[0] = "Thies"  
villes.append("Touba")  
villes.insert(1, "Saint-Louis")  
print(villes)
```

RÉSULTAT →

```
['Thies', 'Saint-Louis', 'Louga', 'Mbour', 'Kaolack', 'Touba']
```

Retirer un élément

Par valeur ou par position

`.remove(x)` retire la valeur `x` ; `.pop()` retire le dernier élément et le renvoie.

```
villes.remove("Touba")
dernier = villes.pop()
print("retire :", dernier)
print(villes)
```

RÉSULTAT →

```
retire : Kaolack
['Thies', 'Saint-Louis', 'Louga', 'Mbour']
```

Trier, compter, tester

```
nombres = [5, 2, 8, 1]
nombres.sort()
print(nombres)
print("taille :", len(nombres))
print(8 in nombres)
```

RÉSULTAT →

```
[1, 2, 5, 8]
taille : 4
True
```

`len()` compte les éléments ; `in` teste la présence d'une valeur et renvoie un booléen.

Piège — `sort()` ne renvoie rien

Modification en place

`.sort()` trie la liste **sur place** et renvoie `None`. Écrire `nombres = nombres.sort()` efface donc la liste !

```
valeurs = [3, 1, 2]
resultat = valeurs.sort()
print("renvoie par sort() :", resultat)
print("liste triee      :", valeurs)
```

RÉSULTAT →

```
renvoie par sort() : None
liste triee       : [1, 2, 3]
```

Les tuples

L'idée : une collection figée

L'idée en une phrase

Un **tuple** ressemble à une liste — ordonné, accès par position — mais il est **figé** : une fois créé, on ne peut plus le modifier. On l'écrit entre parenthèses ().

On l'emploie pour un groupe de valeurs qui forment un tout cohérent et ne doivent pas changer : des coordonnées (*latitude, longitude*), une date (*jour, mois, annee*). Le caractère figé est une *garantie* : personne ne modifiera ces valeurs par erreur.

Accéder à un tuple, et son piège

```
coord = (14.69, -17.44)
print(coord[0])
print(coord[1])
```

RÉSULTAT →

```
14.69
-17.44
```

Tenter de modifier un tuple provoque une erreur :

```
coord[0] = 15.0
TypeError: 'tuple' object does not support item assignment
```

Déballer un tuple (unpacking)

Affecter plusieurs variables d'un coup

On peut répartir les éléments d'un tuple dans autant de variables, en une ligne.

```
latitude, longitude = coord  
print("lat =", latitude)  
print("lon =", longitude)
```

RÉSULTAT →

```
lat = 14.69  
lon = -17.44
```

Très pratique pour échanger deux variables (`a, b = b, a`) ou récupérer plusieurs valeurs à la fois.

Les dictionnaires

L'idée : associer une clé à une valeur

L'idée en une phrase

Un **dictionnaire** associe des **clés** à des **valeurs**, entre accolades `{ }`. On n'accède plus par position, mais par **clé** : `patient["nom"]`. C'est la structure idéale pour décrire un objet par ses attributs.

Chaque clé est unique et pointe vers une valeur. Là où une liste répond à « quel est le troisième élément ? », un dictionnaire répond à « quelle est la valeur du nom ? » — bien plus lisible quand les données ont une étiquette naturelle.

Créer et accéder

```
patient = {"nom": "Fatou", "age": 30, "ville": "Dakar"}  
print(patient)  
print("nom   :", patient["nom"])  
print("ville :", patient["ville"])
```

RÉSULTAT →

```
{'nom': 'Fatou', 'age': 30, 'ville': 'Dakar'}  
nom   : Fatou  
ville : Dakar
```

Ajouter et modifier

```
patient["tel"] = "77 000 00 00"  
patient["age"] = 31  
print(patient)
```

RÉSULTAT →

```
{'nom': 'Fatou', 'age': 31, 'ville': 'Dakar', 'tel': '77 000 00 00'}
```

Affecter une clé existante la **met à jour** ; une clé nouvelle est **créée**. Ici **age** passe à 31 et **tel** apparaît.

Clé absente : `.get` et `in`

Accès direct à une clé inexistante

`patient["email"]` sur une clé absente lève `KeyError: 'email'`. La méthode `.get(cle, default)` renvoie une valeur de repli au lieu de planter ; `in` teste la présence d'une clé.

```
print(patient.get("email", "non renseigné"))  
print("nom" in patient)  
print("email" in patient)
```

RÉSULTAT →

```
non renseigné  
True  
False
```

Parcourir clés, valeurs, couples

```
print(patient.keys())  
print(patient.values())  
print(patient.items())
```

RÉSULTAT →

```
dict_keys(['nom', 'age', 'ville', 'tel'])  
dict_values(['Fatou', 31, 'Dakar', '77 000 00 00'])  
dict_items([('nom', 'Fatou'), ('age', 31), ('ville', 'Dakar'), ('tel', '77 000 00 00')])
```

`.keys()`, `.values()`, `.items()` servent surtout à *parcourir* un dictionnaire — ce qu'on fera avec les boucles en Séance 3.

Les ensembles

L'idée : des valeurs uniques

L'idée en une phrase

Un **ensemble** (`set`) est une collection **non ordonnée** et **sans doublons**, entre accolades `{ }`. Il sert à éliminer les répétitions et à tester très rapidement l'appartenance.

Comme l'ordre n'est pas garanti, on n'accède *pas* par indice. Pour afficher le contenu de façon stable, on le trie avec `sorted()`, qui renvoie une liste ordonnée. Construire un ensemble à partir d'une liste est la façon la plus simple de supprimer les doublons.

Unicité et appartenance

```
liste = ["Dakar", "Louga", "Dakar", "Mbour", "Louga"]
uniques = set(liste)
print("nombre d'uniques :", len(uniques))
print("contenu trie      :", sorted(uniques))
print("Dakar present ?  :", "Dakar" in uniques)
```

RÉSULTAT ↪

```
nombre d'uniques : 3
contenu trie      : ['Dakar', 'Louga', 'Mbour']
Dakar present ?  : True
```

Union et intersection

Combiner deux ensembles

| réunit (tout) ; & garde les éléments **communs**.

```
a = {"Dakar", "Thies", "Louga"}  
b = {"Louga", "Mbour", "Dakar"}  
print("union      :", sorted(a | b))  
print("intersection :", sorted(a & b))
```

RÉSULTAT →

```
union      : ['Dakar', 'Louga', 'Mbour', 'Thies']  
intersection : ['Dakar', 'Louga']
```

Les compréhensions

L'idée : construire une liste en une ligne

L'idée en une phrase

Une **compréhension de liste** crée une liste de façon concise à partir d'une autre collection, selon le modèle `[ expression for element in collection ]`.

On lit : « pour chaque `element` de la `collection`, calcule `expression` et range-la dans la nouvelle liste ». La boucle `for` complète est détaillée en Séance 3 ; ici, elle apparaît seulement dans cette écriture compacte, très courante en science des données.

Transformer chaque élément

```
nombres = [1, 2, 3, 4]  
doubles = [n * 2 for n in nombres]  
print(doubles)
```

RÉSULTAT →

```
[2, 4, 6, 8]
```

Chaque élément de `nombres` est multiplié par 2 ; le résultat est une **nouvelle** liste, sans toucher à l'originale.

Filtrer avec une condition

Ajouter un `if` à la fin

Seuls les éléments qui vérifient la condition sont conservés.

```
villes = ["Dakar", "Saint-Louis", "Mbour", "Ziguinchor", "Louga"]  
longues = [v for v in villes if len(v) > 5]  
print(longues)
```

RÉSULTAT →

```
['Saint-Louis', 'Ziguinchor']
```

On garde les villes dont le nom dépasse 5 caractères. Transformer et filtrer en une ligne rend le code court et lisible.

Synthèse

Quelle structure choisir ?

Structure	Syntaxe	Accès	Modifiable	Doublons
Liste	[]	par indice	oui	autorisés
Tuple	()	par indice	non (figé)	autorisés
Dictionnaire	{c: v}	par clé	oui	clés uniques
Ensemble	{ } / set()	sans indice	oui	interdits

Repère : une suite ordonnée → liste ; un groupe figé → tuple ; des données étiquetées → dictionnaire ; des valeurs uniques → ensemble.

À retenir — Séance 2

L'essentiel

Liste `[ ]` : ordonnée, modifiable ; `.append`, `.insert`, `.remove`, `.pop`, `.sort` (en place, renvoie `None`), `len`, `in`.

Tuple `( )` : figé ; accès par indice ; *unpacking* `a, b = t`.

Dictionnaire `{ }` : clé $\rightarrow$ valeur ; `d[cle]`, `.get(cle, default)`, `in`, `.keys/.values/.items`.

Ensemble `set()` : valeurs uniques, non ordonnées ; `|` union, `&` intersection ; `sorted()` pour afficher.

Compréhension : `[expression for element in collection if condition]`.

Pièges fréquents

- `nombres = nombres.sort()` : `.sort()` renvoie `None` et efface la liste.
- Modifier un **tuple** : impossible (`TypeError`) — il est figé.
- Accéder à une clé absente d'un dictionnaire : `KeyError` ; préférer `.get`.
- Attendre un **ordre** dans un ensemble : il n'y en a pas ; trier avec `sorted()`.
- Confondre les délimiteurs : `[ ]` liste, `( )` tuple, `{ }` dictionnaire ou ensemble.

Prochaine séance

Séance 3 — Contrôle du flux

On sait stocker des collections. La Séance 3 apprend à les **parcourir** et à **décider** : les conditions `if/elif/else`, et les boucles `for` et `while`. C'est là que la compréhension de liste révélera toute sa logique, et qu'on pourra traiter une liste élément par élément.

À faire d'ici là : refaire les cellules « À vous » du notebook de la séance.

Références

- Severance, C. — *Python for Everybody*, 2016.
- Downey, A. — *Think Python*, 2e éd., O'Reilly, 2015.
- Matthes, E. — *Python Crash Course*, 3e éd., No Starch Press, 2023.
- Python Software Foundation — *The Python Tutorial* (Data Structures), 2024.

Ressources de la séance

- **Notebook** · [TP 2 — Structures de données](#) (16 Ko)