Programmation en Python - Séance 4 : Structures de données

Hervé TALE KALACHI

ENSPY – Université de Yaoundé 1

Structures de données — introduction

Définition. Une **structure de données** est un moyen d'organiser et de stocker plusieurs valeurs dans une même variable afin de les manipuler efficacement.

Structures principales en Python :

- `list` — liste ordonnée et modifiable.
- `tuple` — liste ordonnée mais non modifiable.
- `dict` — ensemble de paires `clé : valeur`.
- `set` — ensemble non ordonné d'éléments uniques.

Avantages :

- Regrouper des données liées (ex. notes d'un étudiant, produits, résultats...)
- Itérer facilement avec des boucles.
- Simplifier le traitement de lots d'informations.

Les listes — définition et syntaxe

Définition. Une `liste` est une séquence ordonnée d'éléments, **modifiables**, délimitée par des crochets `[ ]`.

Syntaxe générale :

```
ma_liste = [val1, val2, val3, ...]
```

Exemples :

```
nombres = [10, 20, 30, 40]
fruits = ["pomme", "banane", "kiwi"]
melange = [1, "Python", 3.14, True]
```

Observation :

- Une liste peut contenir des éléments de types différents.
- On accède à chaque élément par son **indice** : `liste[0]`, `liste[1]`, etc.
- L'indice commence toujours à 0.

Les listes — accès et opérations

Accès par indice :

```
fruits = ["pomme", "banane", "kiwi"]
print(fruits[0]) # "pomme"
print(fruits[-1]) # "kiwi" (indice négatif = depuis la fin)
```

Modification :

```
fruits[1] = "mangue"
print(fruits) # ["pomme", "mangue", "kiwi"]
```

Opérations utiles :

```
# Ajout et suppression
fruits.append("orange")
fruits.remove("pomme")
# Longueur et présence
print(len(fruits)) # 3
print("kiwi" in fruits) # True
# Concaténation
liste1 + liste2
```

Les listes — parcours et copies

Parcours avec `for` :

```
fruits = ["pomme", "banane", "kiwi"]
for f in fruits:
    print(f)
```

Parcours avec indices :

```
for i in range(len(fruits)):
    print(i, fruits[i])
```

Copie d'une liste (à ne pas confondre avec référence) :

```
A = [1, 2, 3]
B = A # copie par référence
C = A.copy() # vraie copie
A[0] = 100
print(B, C) # [100, 2, 3] [1, 2, 3]
```

Les listes — tranches et méthodes utiles

Slicing (tranches) :

```
L = [0, 1, 2, 3, 4, 5]
print(L[1:4]) # [1, 2, 3]
print(L[:3]) # [0, 1, 2]
print(L[::2]) # [0, 2, 4]
```

Méthodes fréquemment utilisées :

```
L = [3, 1, 4, 1, 5]
L.append(9)
L.sort()
L.reverse()
print(L.count(1)) # nombre d'occurrences
print(L.index(4)) # position de 4
```

Remarque : les méthodes `sort()` et `reverse()` modifient la liste en place.

À vous de jouer — exercices sur les listes

```
# A) Créez une liste contenant les 10 premiers entiers.
# Affichez le 1er, le dernier et la longueur.

# B) Remplacez la 3e valeur par 99, puis ajoutez 100 a la fin.

# C) Parcourez la liste et affichez la somme des valeurs.

# D) Créez une liste de 5 notes, calculez la moyenne.

# E) Inversez la liste sans utiliser reverse() (indice : slicing)
    .
```

Les tuples — définition et syntaxe

Définition. Un `tuple` est une séquence ordonnée d'éléments, **non modifiable** (immuable), délimitée par des parenthèses `( )`.

Syntaxe générale :

```
mon_tuple = (val1, val2, val3, ...)
```

Exemples :

```
coord = (10, 20)
informations = ("Alice", 25, "Yaoundé")
singleton = (42,) # attention à la virgule finale !
```

Observation :

- Le tuple est similaire à une liste, mais **on ne peut pas le modifier**.
- Il est souvent utilisé pour représenter des données fixes ou des retours de fonctions multiples.

Les tuples — accès et parcours

Accès par indice :

```
coord = (4, 7, 9)
print(coord[0]) # 4
print(coord[-1]) # 9
```

Parcours :

```
for x in coord:
    print(x)
```

Immutabilité :

```
coord[0] = 100 # Erreur : tuple immuable
```

Avantages :

- Plus rapides et plus légers que les listes.
- Peuvent être utilisés comme clés de dictionnaire.
- Garantissent que les données ne seront pas modifiées accidentellement.

Les tuples — déballage (*unpacking*)

Définition. On peut extraire directement les valeurs d'un tuple dans plusieurs variables.

```
coord = (3, 4)
x, y = coord
print(x) # 3
print(y) # 4
```

Astuce : le déballage marche aussi dans les boucles :

```
for nom, age in [("Ada", 18), ("Bob", 21)]:
    print(nom, "a", age, "ans")
```

Liste vs Tuple — comparaison rapide

Liste (`list`)

- Modifiable (mutable)
- Délimitée par `[ ]`
- Taille variable
- Plus lente, plus flexible
- Exemple : `L = [1, 2, 3]`

Tuple (`tuple`)

- Non modifiable (immuable)
- Délimité par `( )`
- Taille fixe
- Plus rapide, plus sûr
- Exemple : `T = (1, 2, 3)`

Conseil : Utilisez des tuples pour des données constantes, des listes pour des données évolutives.

À vous de jouer — exercices sur les tuples

```
# A) Créez un tuple coord = (5, 8, 12)
# et affichez la somme des trois valeurs.

# B) Essayez de modifier coord[1] et notez le message d'erreur.

# C) Définissez une fonction infos_etudiant(nom, age, moyenne)
# qui retourne un tuple (nom, age, mention)
# selon la moyenne de l'étudiant.

# D) Déballage : pour le tuple t = ("Python", 3.11, "Langage"),
# affectez ses trois valeurs àtrois variables et affichez-les.

# E) Créez une liste de tuples (nom, age) pour 3 étudiants,
# puis parcourez cette liste pour afficher : "Nom a X ans".
```

Les dictionnaires — définition et syntaxe

Définition. Un **dictionnaire** est une collection non ordonnée de paires `clé : valeur`. Chaque clé est unique et permet d'accéder rapidement à sa valeur.

Les dictionnaires — définition et syntaxe

Définition. Un **dictionnaire** est une collection non ordonnée de paires `clé : valeur`. Chaque clé est unique et permet d'accéder rapidement à sa valeur. **Syntaxe générale :**

```
mon_dict = {
    "clé1": valeur1,
    "clé2": valeur2,
    "clé3": valeur3
}
```

Exemples :

```
etudiant = {
    "nom": "Alice",
    "age": 21,
    "moyenne": 14.5
}
print(etudiant["nom"]) # "Alice"
```

Les dictionnaires — définition et syntaxe

Syntaxe générale :

```
mon_dict = {
    "clé1": valeur1,
    "clé2": valeur2,
    "clé3": valeur3
}
```

Exemples :

```
etudiant = {
    "nom": "Alice",
    "age": 21,
    "moyenne": 14.5
}
print(etudiant["nom"]) # "Alice"
```

Remarques :

- Les clés peuvent être de type `str`, `int`, ou `tuple` (mais pas `list`).
- L'ordre d'insertion est conservé depuis Python 3.7.

Les dictionnaires — opérations courantes

Accès et modification :

```
etudiant = {"nom": "Alice", "age": 21, "moyenne": 14.5}

# Accéder àune valeur
print(etudiant["age"]) # 21

# Ajouter ou modifier
etudiant["ville"] = "Yaoundé"
etudiant["moyenne"] = 15.2

print(etudiant)
```

Suppression et vérification :

```
del etudiant["age"]
print("nom" in etudiant) # True
print("age" in etudiant) # False
```

Observation : Un dictionnaire peut évoluer dynamiquement : ajout, suppression, mise à jour.

Les dictionnaires — parcours

Parcourir les clés :

```
etudiant = {"nom": "Alice", "age": 21, "moyenne": 15.2}
for cle in etudiant:
    print(cle, "->", etudiant[cle])
```

Parcourir les paires clé–valeur :

```
for cle, valeur in etudiant.items():
    print(f"{cle} : {valeur}")
```

Autres méthodes utiles :

```
print(etudiant.keys()) # dict_keys([...])
print(etudiant.values()) # dict_values([...])
print(len(etudiant)) # nombre d'elements
```

Les dictionnaires — exemples pratiques

Dictionnaire de dictionnaires :

```
etudiants = {
    "E001": {"nom": "Alice", "moyenne": 15},
    "E002": {"nom": "Bob", "moyenne": 12},
    "E003": {"nom": "Claire", "moyenne": 17}
}

print(etudiants["E002"]["nom"]) # "Bob"
```

Application concrète :

```
for matricule, infos in etudiants.items():
    print(f"{infos['nom']} a obtenu {infos['moyenne']}/20")
```

Observation :

- Structure idéale pour représenter des bases de données simples.
- Combine puissance des clés et souplesse des structures Python.

Les dictionnaires — fonctions et méthodes utiles

Méthodes fréquemment utilisées :

```
notes = {"Alice": 15, "Bob": 12, "Claire": 17}

# Obtenir une valeur avec valeur par défaut
print(notes.get("David", 0)) # 0

# Ajouter ou fusionner un autre dictionnaire
notes.update({"David": 14, "Eve": 13})

# Supprimer et recuperer un element
val = notes.pop("Bob")
print("Bob retire :", val)

# Effacer tout le contenu
# notes.clear()
```

Astuce : `get()` évite les erreurs de clé manquante, `update()` permet des fusions rapides.

À vous de jouer — exercices sur les dictionnaires

```
# A) Creez un dictionnaire etudiant = {"nom": "Paul", "age": 22,
    "note": 14.0}
# puis ajoutez une cle "ville" avec la valeur "Yaoundé".

# B) Parcourez le dictionnaire pour afficher chaque clé et sa
    valeur.

# C) Creez un dictionnaire notes = {"Alice": 15, "Bob": 12, "
    Claire": 17}
# et affichez la moyenne generale.

# D) Creez un dictionnaire etudiants avec 3 clés (E001, E002,
    E003)
# contenant chacune un sous-dictionnaire avec nom et moyenne.

# E) Trouvez et affichez l'etudiant ayant la meilleure moyenne.
```

Les ensembles (set) — définition et syntaxe

Définition. Un **ensemble** (set) est une collection **non ordonnée**, **non indexée**, et qui ne contient **aucun doublon**.

Syntaxe générale :

```
mon_ensemble = {val1, val2, val3, ...}
# ou
mon_ensemble = set([val1, val2, val3])
```

Exemples :

```
A = {1, 2, 3, 3, 2}
print(A) # {1, 2, 3} (doublons supprimés)

B = set("PYTHON")
print(B) # {'O', 'H', 'T', 'N', 'P', 'Y'} (ordre aléatoire)
```

Remarques :

- Pas d'accès par indice : on ne peut pas faire `A[0]`.
- Les éléments doivent être immuables (ex : int, str, tuple).

Les ensembles — opérations de base

Ajout et suppression :

```
fruits = {"pomme", "banane"}
fruits.add("kiwi")
fruits.remove("banane") # Erreur si absent
fruits.discard("mangue") # ne lève pas d'erreur
print(fruits)
```

Test d'appartenance :

```
print("pomme" in fruits) # True
print("orange" not in fruits) # True
```

Longueur et effacement :

```
print(len(fruits))
fruits.clear()
```

Les ensembles — opérations ensemblistes

Opérations principales :

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

print(A | B) # union -> {1, 2, 3, 4, 5, 6}
print(A & B) # intersection -> {3, 4}
print(A - B) # différence -> {1, 2}
print(A ^ B) # différence symétrique -> {1, 2, 5, 6}
```

Méthodes équivalentes :

```
A.union(B)
A.intersection(B)
A.difference(B)
A.symmetric_difference(B)
```

Remarque : ces opérations sont très efficaces pour filtrer ou comparer des données.

Les ensembles — inclusion et égalité

Tests d'inclusion :

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

print(A < B) # True (A est sous-ensemble strict)
print(B > A) # True (B est sur-ensemble strict)
print(A.issubset(B)) # True
print(B.issuperset(A)) # True
```

Test d'égalité :

```
C = {3, 2, 1}
print(A == C) # True (l'ordre n'a pas d'importance)
```

Astuce : les ensembles sont très utiles pour éliminer les doublons d'une liste :

```
liste = [1, 2, 2, 3, 1, 4]
unique = set(liste)
print(unique) # {1, 2, 3, 4}
```

Les ensembles — applications pratiques

Exemples d'utilisation :

```
# 1) Trouver les mots uniques d'une phrase
phrase = "le python aime le python"
mots_uniques = set(phrase.split())
print(mots_uniques) # {'le', 'aime', 'python'}

# 2) Vérifier si deux listes ont des éléments communs
L1 = [1, 2, 3, 4]
L2 = [3, 4, 5]
print(set(L1) & set(L2)) # {3, 4}

# 3) Éliminer les doublons d'une liste
notes = [12, 15, 12, 17, 15]
notes_sans_doublons = list(set(notes))
print(notes_sans_doublons)
```

Conclusion : les ensembles sont parfaits pour les tests d'appartenance, la détection de doublons et les comparaisons de groupes d'éléments.

À vous de jouer — exercices sur les ensembles

```
# A) Créez deux ensembles A et B de 5 nombres.
# Affichez leur union, intersection et différence.

# B) Àpartir de la liste noms = ["Alice","Bob","Alice","Eve"],
# affichez les prénoms uniques.

# C) Testez si l'ensemble {1, 2} est inclus dans {1, 2, 3, 4}.

# D) Àpartir de deux phrases, affichez les mots communs.

# E) Créez une fonction elements_communs(L1, L2)
# qui renvoie l'intersection de deux listes sous forme d'ensemble
    .
```

Astuce : pensez à utiliser les opérateurs |, &, -, ^.