Algorithmique Appliquée

BTS SIO SISR

Structures de données fondamentales en Python

Plan

- Notion de conteneur
- Notion d'opérations CRUD
- Tuples
- Ranges
- Lists
- Clonage et copie profonde
- Sets
- Dictionaries
- Technique "Pythonic": comprehensions
- Structure personnalisée

Correction du travail à la maison

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

DM : Retours sur les fonctions et le débogage

Lien vers le sujet de DM.

Notion de conteneur

Container 🇬🇧

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

Types utilisés jusqu'à présent

- `int` : nombre entier.
- `float` : nombre flottant.
- `str` : chaîne de caractères.

Conteneur de caractères

Une chaîne de caractères est un **conteneur de caractères**.

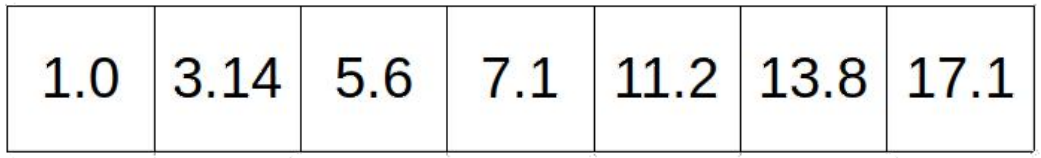

```
chaine = "Bonjour"
```

Conteneur d'entiers

On voudrait aussi pouvoir manipuler des conteneurs d'entiers.

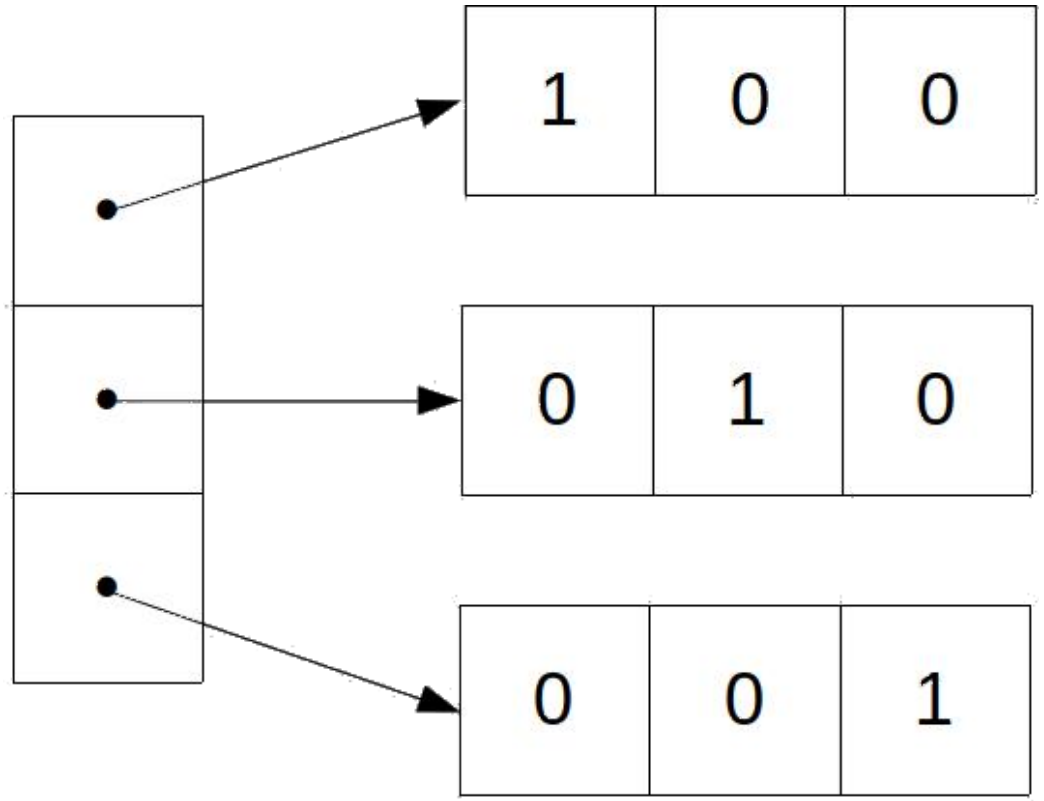
Conteneur de flottants

On voudrait aussi pouvoir manipuler des conteneurs de nombres flottants.



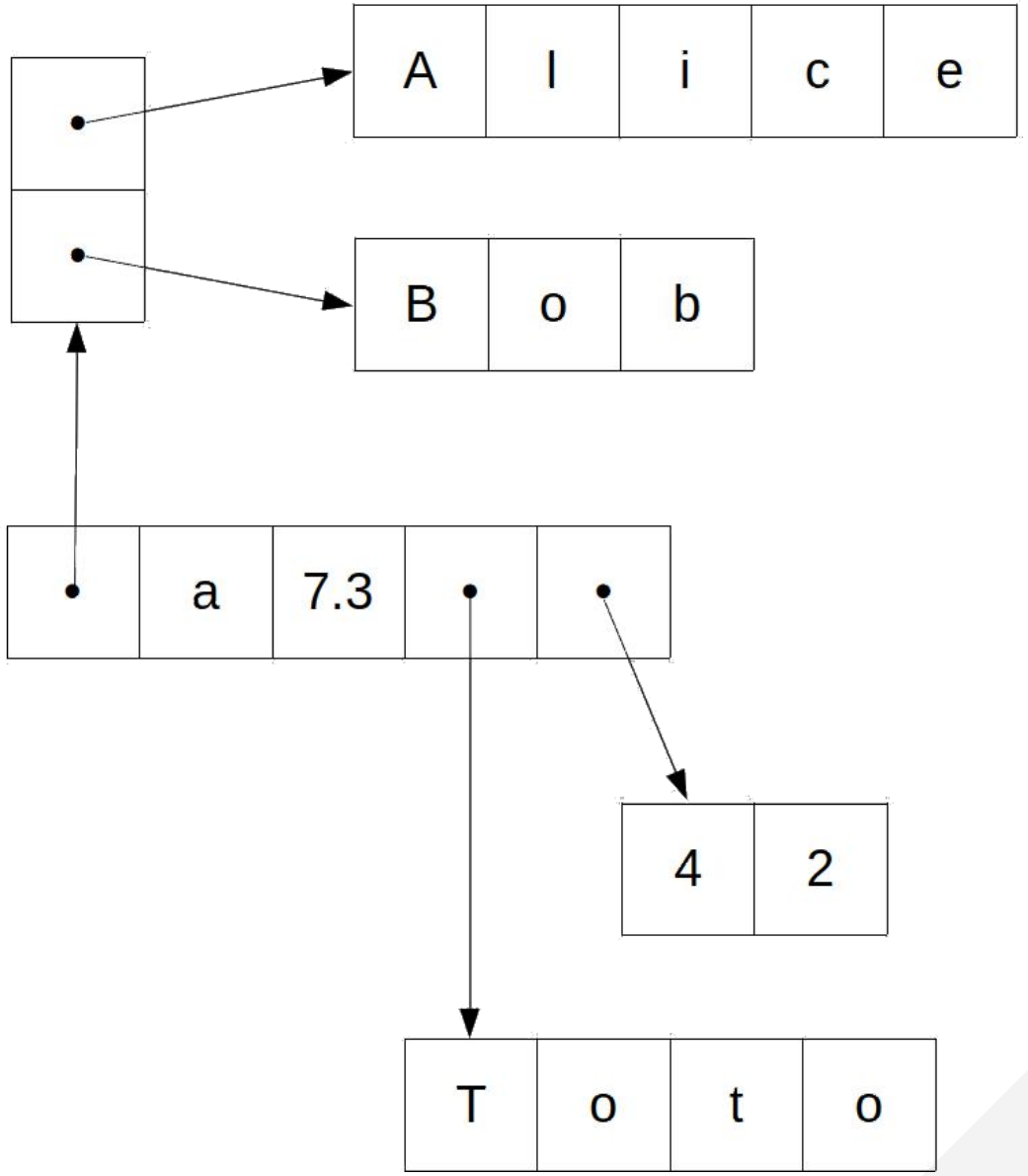
Tableaux à 2 dimensions

On voudrait aussi pouvoir manipuler des tableaux à 2, 3, N dimensions.



Données hétérogènes

On voudrait aussi pouvoir manipuler des données hétérogènes.



Collections

- Les conteneurs sont également appelés **collections**.
- Leur propriété principale est d'être **itérable**.
- Cela signifie que l'on peut itérer sur chaque élément de la collection.

Différents conteneurs

- Il existe différents types de conteneurs, pour répondre à différents besoins.
- Nous allons étudier les principes offerts en Python.
- Avant cela, nous allons étudier les principales opérations sur les conteneurs.

Notion d'opérations CRUD

Create, **R**ead, **U**pdate, **D**elete

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

Opérations de base

- **Création** : Création d'un nouveau conteneur.
- **Lecture** : Lecture de tout ou parti du contenu du conteneur.
- **Mise à Jour** :
  - Modification d'une valeur.
  - Insertion d'une valeur.
  - Suppression d'une valeur.
- **Suppression** : Suppression du conteneur.

Création

```
chaine = "nouvelle chaine"
```

Lecture

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

print(type(chaine))

i = chaine.find("c")
print(i)
```

Mise à jour

```
nouvelle_chaine = chaine.replace("nouvelle", "nouveau")
print(chaine)
print(nouvelle_chaine)
```

```
nouvelle chaine
nouveau chaine
```

On ne peut pas modifier une chaîne de caractères.

La mise à jour est impossible.

Suppression

```
chaine = None
```

Immutabilité

- La **mutabilité** est la capacité à mettre à jour un objet.
- Cela signifie qu'il est possible de modifier sa valeur, d'insérer des éléments, ou d'en supprimer.
- L'**immutabilité** est donc l'impossibilité à mettre à jour un objet.

`str` est immutable

```
chaine = "oui"
chaine = "non"
```

Dans l'exemple ci-dessus, à la 2e ligne, on lie une nouvelle chaîne de caractères `"non"` à la variable `chaine`.

Tuples

Notion de tuple

- Comme les chaînes de caractères, les **tuples** sont des séquences ordonnées immutables d'éléments.
- La différence est que les éléments d'un tuple n'ont pas à être des caractères.
- Les éléments individuels peuvent être de **n'importe quel type**.
- Ils peuvent même être de **types différents**.

Création

```
tuple1 = () # le tuple vide
print(tuple1)

tuple2 = (1, "deux", 3.14)
print(tuple2)

tuple3 = tuple(range(3))
print(tuple3)
```

```
()
(1, 'deux', 3.14)
(0, 1, 2)
```

Valeur répétée

```
t = (1, 1, 1, 1, 1, 1)
print(t)
```

```
(1, 1, 1, 1, 1, 1)
```

Ordre conservé

```
t = (5, 4, 3, 2, 1, 0)
print(t)
```

```
(5, 4, 3, 2, 1, 0)
```

Création d'un tuple à 1 élément

```
t = (1,)
print(t)
```

```
(1,)
```

Tuple imbriqué

```
t = (1, ("deux", "trois"), 3.14)
print(t)
```

```
(1, ("deux", "trois"), 3.14)
```

Itération sur un tuple (1/2)

```
t = (1, 2, 3.14)
for i in range(len(t)):
    print(t[i])
```

```
1
2
3.14
```

Itération sur un tuple (2/2)

```
for element in (1, 2, 3.14):
    print(element)
```

```
1
2
3.14
```

Slicing

```
t = (0, 1, 2, 3, 4, 5, 6, 7)
t2 = t[1:6:2]
print(t2)
```

```
(1, 3, 5)
```

Bornes et itérateurs

Ranges & iterables

Range

```
print(range(10))
```

```
range(0, 10)
```

Comparaison de ranges

```
r1 = range(10)
r2 = range(0, 10, 2)
r3 = range(0, 9, 2)

similaires = (r1 == r2)
print(similaires) # False

similaires = (r2 == r3)
print(similaires) # True
```

Itérateur

- Tous les types **itérables** ont une méthode nommée `__iter__` .
- La méthode `__iter__` renvoie un **objet itérable**.
- Cet objet itérable est utilisé dans les boucles `for` .
- A chaque itération, le prochain élément de la séquence est renvoyée.
- Un `range` est itérable.

Evaluation paresseuse

Lazy evaluation 

- La séquence complète d'un `range` n'est jamais construite intégralement.
- A la place, on conserve uniquement les bornes et l'élément actuel.
- On reste ainsi capable de renvoyer toujours le prochain élément.
- Cette optimisation s'appelle l'**évaluation paresseuse**.

Intérêt de l'évaluation paresseuse (1/2)

```python
import time

depart = time.process_time()

for i in range(1000000000):
    if i == 300:
        break

fin = time.process_time()
temps = fin - depart
print(f"{temps:.6f}s")
```

```
0.000064s
```

Intérêt de l'évaluation paresseuse (2/2)

```python
import time

depart = time.process_time()

for i in tuple(range(1000000000)): # tuple ajouté ici
    if i == 300:
        break

fin = time.process_time()
temps = fin - depart
print(f"{temps:.6f}s")
```

```
60.011151s
```

Immutable et ordonné

- Par nature, un `range` est immutable.
- Par nature, un `range` conserve son ordre initial.

Listes

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

Notion de liste

- Comme les tuples, les **listes** sont une séquence ordonnée de valeurs, où chaque valeur peut être identifiée par un index.
- Une liste, contrairement à un tuple, est **mutable**.
- La liste, notée `list`, est très utilisée en Python.

Création

```
liste1 = [] # la liste vide
print(liste1)

liste2 = [1, "deux", 3.14]
print(liste2)

liste3 = list(range(3))
print(liste3)
```

```
[]
[1, 'deux', 3.14]
[0, 1, 2]
```

Valeur répétée

```
liste = [1, 1, 1, 1, 1, 1]
print(liste)
```

```
[1, 1, 1, 1, 1, 1]
```

Ordre conservé

```
liste = [5, 4, 3, 2, 1, 0]
print(liste)
```

```
[5, 4, 3, 2, 1, 0]
```

Création d'une liste à 1 élément

```
liste = [1]
print(liste)
```

```
[1]
```

Liste imbriquée

```
liste = [1, ["deux", "trois"], 3.14]
print(liste)
```

```
[1, ["deux", "trois"], 3.14]
```

Itération d'une liste (1/2)

```
liste = [1, 2, 3.14]
for i in range(len(liste)):
    print(liste[i])
```

```
1
2
3.14
```

Itération d'une liste (2/2)

```
for element in [1, 2, 3.14]:
    print(element)
```

```
1
2
3.14
```

Slicing

```
liste = [0, 1, 2, 3, 4, 5, 6, 7]
liste2 = liste[1:6:2]
print(liste2)
```

```
[1, 3, 5]
```

Liste de tuples

```
liste = [(1, 2), (3, 4)]
print(liste)
```

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

Tuple de listes

```
t = ([1, 2], [3, 4])
print(liste)
```

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

Liste à 2 dimensions

```
liste = [
    [1, 0, 0],
    [0, 1, 0],
    [0, 0, 1]
]
```

Liste à 3 dimensions

```
liste = [
    [
        [1, 0],
        [0, 1]
    ],
    [
        [1, 0],
        [0, 1]
    ]
]
```

Concaténation de listes

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste3 = liste1 + liste2
print(liste3)
```

```
[1, 2, 3, 4, 5, 6]
```

Modification d'une valeur

```
liste = [1, 2, 3]
liste[0] = 5
print(liste)
```

```
[5, 2, 3]
```

Insertion d'une valeur au début

```
liste = [1, 2, 3]
liste.insert(0, 42)
print(liste)
```

```
[42, 1, 2, 3]
```

Insertion d'une valeur au milieu

```
liste = [1, 2, 3]
liste.insert(2, 42)
print(liste)
```

```
[1, 2, 42, 3]
```

Insertion d'une valeur à la fin

```
liste = [1, 2, 3]
liste.append(42)
print(liste)
```

```
[1, 2, 3, 42]
```

Suppression d'une valeur

```
liste = [1, 2, 3]
liste.remove(liste[0])
print(liste)
```

```
[2, 3]
```

Suppression de la dernière valeur

```
liste = [1, 2, 3]
liste.pop()
print(liste)
```

```
[1, 2]
```

TD : Implémenter les opérations matricielles les plus classiques

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

TD : Opérations matricielles classiques

Lien vers le sujet de TD.

Clonage et copie profonde

Shallow and deep copy 🇬🇧

Le problème

```
liste1 = [1, 2, 3]
liste2 = liste1

liste2.append(4)
print(liste1)
```

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

Impact imbriqué

```
def f(liste):
    liste.append(4)

liste1 = [1, 2, 3]
f(liste1)
print(liste1)
```

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

Egalité de listes

```
liste1 = [1, 2, 3]
liste2 = [1, 2, 3]

egaux = (liste1 == liste2)
print(egaux)
```

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

Egalité d'objets : opérateur `is`

```
liste1 = [1, 2, 3]
liste2 = [1, 2, 3]
liste3 = liste2

egaux = (liste1 is liste2)
print(egaux) # False

egaux = (liste2 is liste3)
print(egaux) # True
```

Clonage (1/2)

```
liste1 = [1, 2, 3]
liste2 = liste1.copy()
liste2.append(4)
print(liste1)
```

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

Clonage (2/2)

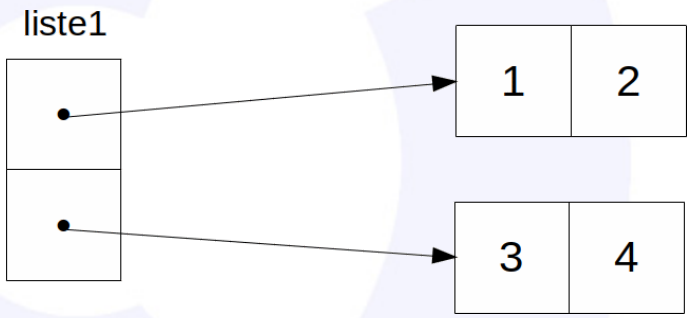
```
liste1 = [1, 2, 3]
liste2 = liste1[:]
liste2.append(4)
print(liste1)
```

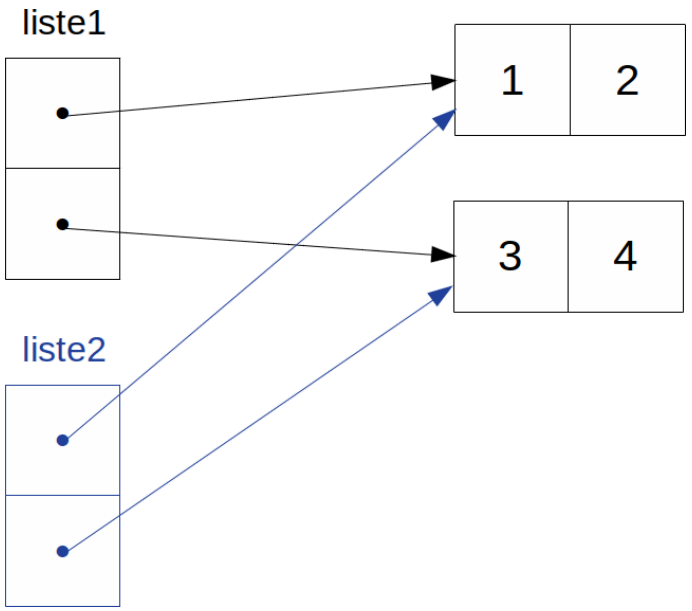
```
[1, 2, 3]
```

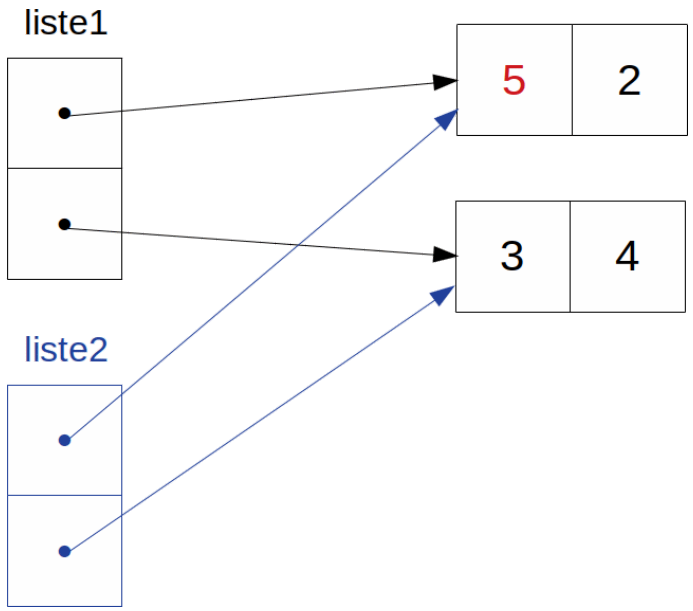
Limites du clonage

```
liste1 = [[1, 2], [3, 4]]
liste2 = liste1[:]
liste2[0][0] = 5
print(liste1)
```

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


liste1
1
2
3
4


liste1
1
2
3
4
liste2


liste1
5
2
3
4
liste2

Copie profonde

```
import copy

liste1 = [[1, 2], [3, 4]]
liste2 = copy.deepcopy(liste1)
liste2[0][0] = 5
print(liste1)
```

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

Ensembles

Sets

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

Notion d'ensemble

- Comme en mathématiques, un ensemble, noté `set`, est une collection **non-ordonnée** d'**éléments uniques**.
- Les éléments individuels peuvent être de **types différents**.
- Ils doivent être **hashables**, c'est-à-dire fournir une définition pour les méthodes :
  - `__hash__` : génère un `int` unique pour un objet.
  - `__eq__` : égalité entre 2 objets.

Création

```
set1 = {} # l'ensemble vide
print(set1)

set2 = {1, "deux", 3.14}
print(set2)

set3 = set(range(3))
print(set3)
```

```
{}
{'deux', 1, 3.14}
{0, 1, 2}
```

Valeur répétée

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

```
{1}
```

Ordre non conservé

```
s = {5, 4, 3, 2, 1, 0}
print(s)
```

```
{0, 1, 2, 3, 4, 5}
```

Création d'un set à 1 élément

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

```
{1}
```

Set imbriqué ?

```
s = {1, {"deux", "trois"}, 3.14}
print(s)
```

```
TypeError: unhashable type: 'set'
```

Itération sur un set (1/2)

```
s = {"Mona Lisa", "La Scapigliata", "La Belle Ferronnière"}
for i in range(len(s)):
    print(s[i])
```

```
TypeError: 'set' object is not subscriptable
```

Itération sur un set (2/2)

```
s = {"Mona Lisa", "La Scapigliata", "La Belle Ferronnière"}
for element in s:
    print(element)
```

```
La Belle Ferronnière
La Scapigliata
Mona Lisa
```

Union

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7}
s3 = s1 | s2
print(s3)
```

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

Intersection

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7}
s3 = s1 & s2
print(s3)
```

```
{4, 5}
```

Différence

```
s1 = {1, 2, 3, 4, 5}
s2 = {4, 5, 6, 7}
s3 = s1 - s2
print(s3)
```

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

Sous-ensemble

```
s1 = {1, 2, 3, 4, 5}
s2 = {1, 2, 3}
sous_ensemble = (s2 <= s1)
print(sous_ensemble) # True
```

Dictionnaires

Dictionaries

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

Notion de dictionnaire

- Un dictionnaire, noté `dict`, est une collection de **paires clé/valeur**.
- Une clé doit être **hashable**.
- Une valeur peut avoir **n'importe quel type**.
- Les clés et les valeurs peuvent avoir des types différents.
- Les dictionnaires sont très utilisés en Python.

Création

```
dico1 = dict() # dictionnaire vide
print(dico1)

dico2 = {"un": 1, "deux": 2, "trois": 3}
print(dico2)
```

```
{}
{'un': 1, 'deux': 2, 'trois': 3}
```

Valeur répétée

```python
d = {"un": 1, "un": 1}
print(d)
```

```
{'un': 1}
```

Ordre conservé

≥ **Python 3.8**

```
d = {"trois": 3, "deux": 2, "un": 1}
print(d)
```

```
{'trois': 3, 'deux': 2, 'un': 1}
```

Création d'un dictionnaire à 1 élément

```
d = {"un": "ein"}
print(d)
```

```
{'un': 'ein'}
```

Dictionnaire imbriqué

```
d = {
    "fr_en": {"un": "one", "deux": "two", "trois": "three"},
    "fr_de": {"un": "ein", "deux": "zwei", "trois": "drei"}
}
print(d)
```

```
{'fr_en': {'un': 'one', 'deux': 'two', 'trois': 'three'},
 'fr_de': {'un': 'ein', 'deux': 'zwei', 'trois': 'drei'}}
```

Itération sur un dictionnaire (1/3)

```
d = {"Janvier": 1, "Février": 2, "Mars": 3}
for cle in d:
    print(f"Le numéro du mois de {cle} est {d[cle]}.")
```

```
Le numéro du mois de Janvier est 1.
Le numéro du mois de Février est 2.
Le numéro du mois de Mars est 3.
```

Itération sur un dictionnaire (2/3)

```
d = {"Janvier": 1, "Février": 2, "Mars": 3}
for cle, valeur in d.items():
    print(f"Le numéro du mois de {cle} est {valeur}.")
```

```
Le numéro du mois de Janvier est 1.
Le numéro du mois de Février est 2.
Le numéro du mois de Mars est 3.
```

Itération sur un dictionnaire (3/3)

```
d = {"Janvier": 1, "Février": 2, "Mars": 3}
for valeur in d.values():
    print(f"Le numéro du mois est {valeur}.")
```

```
Le numéro du mois est 1.
Le numéro du mois est 2.
Le numéro du mois est 3.
```

Modification d'une valeur

```
taille = {"petit": 140, "moyen": 170, "grand": 190}
taille["moyen"] = 165
print(taille)
```

```
{'petit': 140, 'moyen': 165, 'grand': 190}
```

Insertion d'une valeur

```
calories = {"eau": 0, "jus de fruit": 100, "coca": 100000}
calories["vin"] = 1000
print(calories)
```

```
{'eau': 0, 'jus de fruit': 100, 'coca': 100000, 'vin': 1000}
```

Suppression d'une valeur

```
heros = {
    "Catwoman": 300,
    "Batman": 400,
    "Wonderwoman": 900,
    "Robin": 2
}
del heros["Robin"]
print(heros)
```

```
{'Catwoman': 300, 'Batman': 400, 'Wonderwoman': 900}
```

Technique "Pythonic" : compréhensions

Notion de compréhension

- Une **compréhension** est une syntaxe alternative **plus compacte** pour créer des conteneurs.
- Cette syntaxe est surtout utilisée pour les **listes** et les **dictionnaires** mais peut aussi être utilisée pour les sets.

Compréhension avec une liste

```
liste = [i ** 2 for i in range(5)]
print(liste)
```

```
[0, 1, 4, 9, 16]
```

Compréhension avec condition

```
liste = [i ** 2 for i in range(10) if i % 2 == 0]
print(liste)
```

```
[0, 4, 16, 36, 64]
```

Compréhension imbriquée

```
liste = [(i, j) for i in range(2) for j in range(3)]
print(liste)
```

```
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

⚠️ Trop complexe ⚠️

```
liste = [[i * j + k for i in range(3)]
             for j in range(6) if j % 3 == 0
                 for k in range(2)]
print(liste)
```

```
[[0, 0, 0], [1, 1, 1], [0, 3, 6], [1, 4, 7]]
```

Compréhension avec un set

```
s = {i ** 3 for i in range(1, 4)}
print(s)
```

```
{8, 1, 27}
```

Compréhension avec un dictionnaire

```
dico = {i: i ** 2 for i in range(2, 11)}
print(dico)
```

```
{2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Compréhension avec un dictionnaire et une condition

```
dico = {i: i ** 2 for i in range(2, 11) if i % 2 == 0}
print(dico)
```

```
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

TP : Utiliser un dictionnaire pour gérer un hôpital avec des patients, des médecins et des soins à apporter

CHAMBRE DE COMMERCE ET D'INDUSTRIE

1er ACCÉLÉRATEUR DES ENTREPRISES

Yvo Solutions

TP : Gestion d'un hôpital

Lien vers le sujet de TP.

Structure personnalisée

Notion de classe comme Tuple avancé

Limites des conteneurs (1/2)

- Modélisation d'un point 3D avec une liste :

```
point = [0, 0, 0]
x, y, z = point[0], point[1], point[2]
```

- Il faut garder le modèle mental de correspondance :
  - 0 pour l'abscisse x,
  - 1 pour l'ordonnée y,
  - 2 pour l'ordonnée z.
- Cette correspondance complexifie le code.

Limites des conteneurs (2/2)

- Modélisation d'un point 3D avec un dictionnaire :

```
point = {"x": 0, "y": 0, "z": 0}
x, y, z = point["x"], point["y"], point["z"]
```

- Cette solution implique une empreinte mémoire supérieure.

Structure de données

- Python nous permet de définir nos **propres structures de données**.

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float = 0.
    y: float = 0.
    z: float = 0.

point = Point()
x, y, z = point.x, point.y, point.z
```

Avantages

- Plus **explicite** que l'emploi des indexes d'une liste.
- Code plus précis et **plus simple** à comprendre et maintenir.
- **Moins lourd** en mémoire par rapport à un dictionnaire.
- Egalement **plus performant** qu'une liste car il n'y a pas d'indirection.

Conclusion

Les structures de données sont fondamentales pour définir des **abstractions de niveau supérieur** et simplifier la programmation.