FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Les structures de données (Les Tuples, Les séquences, Les listes, Les dictionnaires)

Master : MNSA

Pr: Youssef Ouassit

Structures de donnée

Définition :

Les structures de données sont des moyens d'organiser et de stocker des données afin qu'elles puissent être utilisées de manière efficace. En Python, plusieurs structures de données sont disponibles, chacune adaptée à différents types de tâches. Voici une présentation des principales structures de données en Python.

Structures de donnée

Définition :

Les structures de données essentielles:

Standards	Importées par des modules
list tuple str set dict	Stack Queue Linked list Tree …

Les listes

Définition d’une liste

Une liste est une structure de données flexible et mutable qui peut contenir un ensemble ordonné d'éléments de différents types.

1. Une liste une collection de données ordonnée.
2. Permet de manipuler une séquence d’objets modifiables.
3. Contient un nombre non fixé d’éléments.
4. Contient des données de différents types.

Les listes

Déclaration

```
# Liste vide
ma_liste_vide = []

# Liste avec des éléments
ma_liste = [1, 2, 3, 4, 5]

# Liste avec des types différents
ma_liste_melangee = [1, "Python", 3.14, True]
```

Les listes

Déclaration

```
# Liste vide avec list()
ma_liste_vide = list()

# Liste à partir d'une chaîne de caractères
ma_liste_chaine = list("Python") # ['P','y','t','h','o','n']

# Liste à partir d'un tuple
ma_liste_tuple = list((1, 2, 3))  # [1, 2, 3]
```

Déclaration

La "list comprehension" :

```
# Liste des carrés des nombres de 0 à 9
carres = [x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Liste des nombres pairs de 0 à 9
pairs = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]
```

Les listes

Opérateurs de base

Opérateur d'Appartenance **in**

```
fruits = ["pomme", "banane", "cerise"]
print("pomme" in fruits)  # True
print("orange" in fruits)  # False
```

Opérateur de concaténation **+**

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste3 = liste1 + liste2  # [1, 2, 3, 4, 5, 6]
```

Les listes

Opérateurs de base

Opérateur de répétition *

```
liste = [1, 2, 3]
resultat = liste * 3  # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Opérateur d’indexation []

```
fruits = ["pomme", "banane", "cerise"]
print(fruits[0])  # 'pomme'
print(fruits[-1])  # 'cerise'
```

Opérateurs de base

Opérateur de Slicing **[debut : fin : pas]**

```
chiffres = [0, 1, 2, 3, 4, 5]
sous_liste = chiffres[1:4]  # [1, 2, 3]
sous_liste2 = chiffres[:3]   # [0, 1, 2]
sous_liste3 = chiffres[::2]  # [0, 2, 4]
sous_liste3 = chiffres[:]  # [0, 1, 2, 3, 4, 5]
```

Les listes

Les méthodes

Méthode `append()`

Description : Ajoute un élément à la fin de la liste.

Syntaxe : `liste.append(élément)`

Exemple :

```python
fruits = ["pomme", "banane"]
fruits.append("cerise")  # ["pomme", "banane", "cerise"]
```

Les listes

Les méthodes

Méthode `extend()`

Description : Ajoute les éléments d'une autre liste à la fin de la liste courante.

Syntaxe : `liste.extend(autre_liste)`

Exemple :

```python
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste1.extend(liste2)  # [1, 2, 3, 4, 5, 6]
```

Les listes

Les méthodes

Méthode `insert()`

Description : Insère un élément à une position spécifique de la liste.

Syntaxe : `liste.insert(index, élément)`

Exemple :

```python
fruits = ["pomme", "cerise"]
fruits.insert(1, "banane")  # ["pomme", "banane", "cerise"]
```

Les listes

Les méthodes

Méthode `remove()`

Description : Supprime la première occurrence d'un élément dans la liste.

Syntaxe : `liste.remove(élément)`

Exemple :

```python
fruits = ["pomme", "banane", "cerise", "banane"]
fruits.remove("banane")  # ["pomme", "cerise", "banane"]
```

Les listes

Les méthodes

Méthode `pop()`

Description : Supprime et retourne l'élément à une position donnée (par défaut, le dernier).

Syntaxe : `liste.pop([index])`

Exemple :

```python
fruits = ["pomme", "banane", "cerise"]
dernier_fruit = fruits.pop()  # "cerise"
```

Les listes

Les méthodes

Méthode `clear()`

Description : Supprime tous les éléments de la liste.

Syntaxe : `liste.clear()`

Exemple :

```python
fruits = ["pomme", "banane", "cerise"]
fruits.clear()  # []
```

Les listes

Les méthodes

Méthode `index()`

Description : Retourne l'index de la première occurrence d'un élément dans la liste.

Syntaxe : `liste.index(élément)`

Exemple :

```python
fruits = ["pomme", "banane", "cerise"]
index_banane = fruits.index("banane")  # 1
```

Les listes

Les méthodes

Méthode `count()`

Description : Compte le nombre de fois qu'un élément apparaît dans la liste.

Syntaxe : `liste.count(élément)`

Exemple :

```python
fruits = ["pomme", "banane", "cerise", "banane"]
nombre_banane = fruits.count("banane")  # 2
```

Les listes

Les méthodes

Méthode `sort()`

Description : Trie les éléments de la liste en place.

Syntaxe : `liste.sort([key=None, reverse=False])`

Exemple :

python Copier le code

```python
chiffres = [3, 1, 4, 1, 5, 9]
chiffres.sort()  # [1, 1, 3, 4, 5, 9]
```

Les listes

Les méthodes

Méthode `reverse()`

Description : Inverse l'ordre des éléments de la liste en place.

Syntaxe : `liste.reverse()`

Exemple :

```python
chiffres = [1, 2, 3]
chiffres.reverse()  # [3, 2, 1]
```

Les listes

Les méthodes

Méthode `copy()`

Description : Retourne une copie superficielle de la liste.

Syntaxe : `liste.copy()`

Exemple :

python Copier le code

```python
fruits = ["pomme", "banane", "cerise"]
fruits_copie = fruits.copy()  # ["pomme", "banane", "cerise"]
```

Les listes

Les méthodes

Méthode : `clear`

Description : Supprime tous les éléments de la liste.

Syntaxe : `liste.clear()`

Exemple :

```python
liste = [1, 2, 3]
liste.clear()
print(liste)  # Affiche : []
```

Les listes

Les fonctions

Fonction : `` `len` ``

Description : Retourne la longueur (le nombre d'éléments) d'une liste.

Syntaxe : `` `len(liste)` ``

Exemple :

```python
liste = [1, 2, 3]
print(len(liste))  # Affiche : 3
```

Les listes

Les fonctions

Fonction : `sorted`

Description : Retourne une nouvelle liste contenant les éléments triés d'une liste.

Syntaxe : `sorted(iterable, key=None, reverse=False)`

Exemple :

python Copier le code

```python
liste = [3, 1, 2]
print(sorted(liste))  # Affiche : [1, 2, 3]
```

Les listes

Les fonctions

Fonction : `` `sum` ``

Description : Retourne la somme des éléments d'une liste (les éléments doivent être des nombres).

Syntaxe : `` `sum(iterable, start=0)` ``

Exemple :

python Copier le code

```python
liste = [1, 2, 3]
print(sum(liste))  # Affiche : 6
```

Les listes

Les fonctions

Fonction : `min`

Description : Retourne le plus petit élément d'une liste.

Syntaxe : `min(iterable, *args, key=None)`

Exemple :

```python
liste = [3, 1, 2]
print(min(liste))  # Affiche : 1
```

Les listes

Les fonctions

Fonction : `max`

Description : Retourne le plus grand élément d'une liste.

Syntaxe : `max(iterable, *args, key=None)`

Exemple :

```python
liste = [3, 1, 2]
print(max(liste))  # Affiche : 3
```

Les listes

Les fonctions

Fonction : `any`

Description : Retourne `True` si au moins un élément de la liste est vrai.

Syntaxe : `any(iterable)`

Exemple :

```python
liste = [0, 1, 2]
print(any(liste))  # Affiche : True
```

Les listes

Les fonctions

Fonction : `all`

Description : Retourne `True` si tous les éléments de la liste sont vrais.

Syntaxe : `all(iterable)`

Exemple :

```python
liste = [1, 2, 3]
print(all(liste))  # Affiche : True
```

Les listes

Les fonctions

Fonction : `enumerate`

Description : Retourne un objet énuméré, qui est un itérable de tuples (index, élément).

Syntaxe : `enumerate(iterable, start=0)`

Exemple :

```python
liste = ['a', 'b', 'c']
for index, valeur in enumerate(liste):
    print(index, valeur)
# Affiche :
# 0 a
# 1 b
# 2 c
```

Les listes

Les fonctions

Fonction : `` `zip` ``

Description : Combine plusieurs listes (ou autres itérables) en une liste de tuples.

Syntaxe : `` `zip(*iterables)` ``

Exemple :

```python
liste1 = [1, 2, 3]
liste2 = ['a', 'b', 'c']
result = list(zip(liste1, liste2))
print(result)  # Affiche : [(1, 'a'), (2, 'b'), (3, 'c')]
```

Les listes

Les fonctions

Fonction : `reversed`

Description : Retourne un itérable qui permet de parcourir la liste en sens inverse.

Syntaxe : `reversed(iterable)`

Exemple :

```python
liste = [1, 2, 3]
print(list(reversed(liste)))  # Affiche : [3, 2, 1]
```

Les tuples

Définition d’un tuple

Un tuple est une séquence ordonnée d'éléments, qui peuvent être de types différents (par exemple, entiers, chaînes de caractères, listes, etc.).

Contrairement aux listes, les tuples sont immuables, ce qui signifie que leurs éléments ne peuvent pas être modifiés après leur création. Une fois qu'un tuple est défini, il ne peut pas être modifié, étendu ou réduit.

Les tuples

Déclaration

```
# Tuple vide
tuple_vide = ()

# Tuple avec des éléments
mon_tuple = (1, 2, 3, 4, 5)

# Tuple avec des éléments sans parenthèses
mon_tuple = 1, 2, 3, 4, 5

# Tuple avec des types différents
mon_tuple_melangee = (1, "Python", 3.14, True)
```

Les tuples

Déclaration

```
# Tuple vide avec tuple()
mon_tuple_vide = tuple()

# Tuple à partir d'une chaîne de caractères
mon_tuple_chaine = tuple("Python") # ('P','y','t','h','o','n’)

# Tuple à partir d’une liste
mon_tuple_tuple = tuple([1, 2, 3])  # (1, 2, 3)

# Tuple avec un seul élément
mon_tuple_vide = (1, )
```

Les tuples

Déclaration

```
# Tuple des carrés des nombres de 0 à 9
carres = (x**2 for x in range(10))
# (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

# Tuple des nombres pairs de 0 à 9
pairs = (x for x in range(10) if x % 2 == 0)
# (0, 2, 4, 6, 8)
```

Les tuples

Opérateurs de base

Opérateur d'Appartenance **in**

```
fruits = ("pomme", "banane", "cerise" )
print("pomme" in fruits)  # True
print("orange" in fruits)  # False
```

Opérateur de concaténation **+**

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
tuple3 = tuple1 + tuple2  # (1, 2, 3, 4, 5, 6)
```

Les tuples

Opérateurs de base

Opérateur de répétition *

```
tuple = [1, 2, 3]
resultat = tuple * 3  # (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Opérateur d’indexation []

```
fruits = ("pomme", "banane", "cerise« )
print(fruits[0])  # 'pomme'
print(fruits[-1])  # 'cerise'
```

Opérateurs de base

Opérateur de Slicing **[debut : fin : pas]**

```
chiffres = (0, 1, 2, 3, 4, 5)
sous_tuple = chiffres[1:4]  # (1, 2, 3)
sous_tuple2 = chiffres[:3]   # (0, 1, 2)
sous_tuple3 = chiffres[::2]  # (0, 2, 4)
sous_tuple4 = chiffres[:]  # (0, 1, 2, 3, 4, 5)
```

Les tuples

Les méthodes

Méthode : `count`

Description : Retourne le nombre d'occurrences d'un élément spécifique dans le tuple.

Syntaxe : `tuple.count(element)`

Exemple :

```python
mon_tuple = (1, 2, 2, 3)
print(mon_tuple.count(2))  # Affiche : 2
```

Les tuples

Les méthodes

Méthode : `index`

Description : Retourne l'index de la première occurrence d'un élément spécifique dans le tuple. Lève une exception `ValueError` si l'élément n'est pas trouvé.

Syntaxe : `tuple.index(element, start=0, end=len(tuple))`

Exemple :

```python
mon_tuple = (1, 2, 3)
print(mon_tuple.index(2))  # Affiche : 1
```

Les tuples

Les fonctions

Les fonctions sur les listes sont aussi applicable sur les tuples :

- Len
- min
- max
- sum
- sorted
- reversed
- zip
- any
- all

Les str

Définition d’une chaîne de caractères

Le type **str** en Python est une séquence d'unités de caractères Unicode.
Les chaînes de caractères sont utilisées pour stocker et manipuler du texte.

Les chaînes de caractères sont définies en utilisant des guillemets simples ('), des guillemets doubles ("), ou des triples guillemets (''' ou """) pour les chaînes multilignes.

Les chaînes de caractères

Déclaration

```
# Définir une chaîne

chaine1 = 'Bonjour'

chaine2 = "le monde"

chaine3 = '''Ceci est une chaîne qui s'étend sur
plusieurs lignes.'''
```

Les chaînes de caractères

Opérateurs de base

Opérateur d'Appartenance **in**

```
ch = "pomme"
print("po" in ch)  # True
print("ora" in ch)  # False
```

Opérateur de concaténation **+**

```
ch1 = "Bonjour "
ch2 = "Ahmed"
ch3 = ch1 + ch2  # "Bonjour Ahmed"
```

Les chaînes de caractères

Opérateurs de base

Opérateur de répétition *

```
ch = "bon"
resultat = ch * 2  # "bonbon"
```

Opérateur d’indexation []

```
ch = "pomme"
print(ch[0])  # 'p'
print(ch[-1])  # 'e'
```

Opérateurs de base

Opérateur de Slicing **[debut : fin : pas]**

```
ch = "Bonjour"
sous_chaine = ch[:3]  # Bon
sous_chaine2 = ch[1:3]   # on
sous_chaine3 = ch[::2]  # Bnor
sous_chaine4 = ch[:]  # Bonjour
```

Les chaînes de caractères

Les méthodes

Méthode : `upper`

Description : Retourne une nouvelle chaîne où tous les caractères sont convertis en majuscules.

Syntaxe : `str.upper()`

Exemple :

```python
texte = "bonjour"
print(texte.upper())  # Affiche : BONJOUR
```

Les chaînes de caractères

Les méthodes

Méthode : `lower`

Description : Retourne une nouvelle chaîne où tous les caractères sont convertis en minuscules.

Syntaxe : `str.lower()`

Exemple :

```python
texte = "BONJOUR"
print(texte.lower())  # Affiche : bonjour
```

Les chaînes de caractères

Les méthodes

Méthode : `strip`

Description : Supprime les espaces (ou autres caractères spécifiés) au début et à la fin de la chaîne.

Syntaxe : `str.strip([caractères])`

Exemple :

```python
texte = "  bonjour  "
print(texte.strip())  # Affiche : bonjour
```

Les chaînes de caractères

Les méthodes

Méthode : `replace`

Description : Remplace toutes les occurrences d'une sous-chaîne par une autre dans la chaîne d'origine.

Syntaxe : `str.replace(old, new[, count])`

Exemple :

```python
texte = "bonjour monde"
print(texte.replace("monde", "tout le monde"))  # Affiche : bonjour tout le monde
```

Les chaînes de caractères

Les méthodes

Méthode : `split`

Description : Divise la chaîne en une liste de sous-chaînes en utilisant le séparateur spécifié. Si aucun séparateur n'est fourni, les espaces sont utilisés par défaut.

Syntaxe : `str.split([separator[, maxsplit]])`

Exemple :

```python
texte = "bonjour tout le monde"
print(texte.split())  # Affiche : ['bonjour', 'tout', 'le', 'monde']
```

Les chaînes de caractères

Les méthodes

Méthode : `` `join` ``

Description : Concatène les éléments d'un itérable (comme une liste) en une seule chaîne, en utilisant la chaîne sur laquelle la méthode est appelée comme séparateur.

Syntaxe : `` `str.join(iterable)` ``

Exemple :

python — Copier le code

```python
liste = ["bonjour", "tout", "le", "monde"]
print(" ".join(liste))  # Affiche : bonjour tout le monde
```

Les chaînes de caractères

Les méthodes

Méthode : `` `find` ``

Description : Retourne l'index de la première occurrence de la sous-chaîne spécifiée dans la chaîne. Retourne `` `-1` `` si la sous-chaîne n'est pas trouvée.

Syntaxe : `` `str.find(sub[, start[, end]])` ``

Exemple :

python — Copier le code

```python
texte = "bonjour"
print(texte.find("jour"))  # Affiche : 3
```

Les chaînes de caractères

Les méthodes

Méthode : `count`

Description : Retourne le nombre de fois qu'une sous-chaîne apparaît dans la chaîne.

Syntaxe : `str.count(sub[, start[, end]])`

Exemple :

```python
texte = "bonjour tout le monde"
print(texte.count("o"))  # Affiche : 3
```

Les chaînes de caractères

Les fonctions

On retrouve les même fonctions déjà vu :

- len
- max
- min
- …

Des fonctions spécifiques aux chaînes de caractères:

- **ord** : Retourne le code Unicode d'un caractère.
- **chr** : Retourne le caractère correspondant à un code Unicode
- **eval :** Évalue une chaîne comme une expression Python.
- **input** : Demande une entrée utilisateur.

Les dictionnaires

Définition

Un dictionnaire est une collection non ordonnée de paires clé-valeur. Chaque clé dans un dictionnaire doit être unique et immuable (par exemple, une chaîne de caractères, un nombre, un tuple), tandis que les valeurs peuvent être de n'importe quel type (chaîne, nombre, liste, autre dictionnaire, etc.).

Les dictionnaires

Déclaration

```
# Définir un dictionnaire vide avec accolades
D = { }

# Définir un dictionnaire vide avec dict
D = dict()

# Exemple d'un dictionnaire simple
etudiant = {
      "nom": "Alice",
      "âge": 25,
      "ville": "Paris"
}
```

Les dictionnaires

Déclaration

```
# À partir d'une liste de tuples
liste_tuples = [("nom", "Alice"), ("âge", 25), ("ville", "Paris")]
etudiant = dict(liste_tuples)

# À partir de deux listes
clés = ["nom", "âge", "ville"]
valeurs = ["Alice", 25, "Paris"]
etudiant = dict(zip(clés, valeurs))
```

Les dictionnaires

Opérateurs de base

Opérateur d'Appartenance in

```
D = {"nom": "Alice", "âge": 25}
print("nom" in D)  # True
print("adresse" in D)  # False
```

Opérateur de concaténation +

```
ch1 = "Bonjour "
ch2 = "Ahmed"
ch3 = ch1 + ch2  # "Bonjour Ahmed"
```

Les dictionnaires

Opérateurs de base

Opérateur de l'union |

```
dict1 = {"nom": "Alice", "âge": 25}
dict2 = {"ville": "Paris", "âge": 26}
result = dict1 | dict2
# Résultat : {"nom": "Alice", "âge": 26, "ville": "Paris"}
```

Opérateur d’indexation []

```
D = {"nom": "Alice", "âge": 25}
print(D['nom'])  # 'Alice'
print(D['âge'])  # 25
```

Les dictionnaires

Opérateurs de base

Union et Mise à Jour en Place |=

```
dict1 = {"nom": "Alice", "âge": 25}
dict2 = {"ville": "Paris", "âge": 26}
dict1 |= dict2
# dict1 devient {"nom": "Alice", "âge": 26, "ville": "Paris"}
```

Les dictionnaires

Les méthodes

Méthode `clear()`

Description : Supprime toutes les paires clé-valeur du dictionnaire.

Syntaxe : `dictionnaire.clear()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
mon_dict.clear()
print(mon_dict)  # Affiche : {}
```

Les dictionnaires

Les méthodes

Méthode `copy()`

Description : Retourne une copie superficielle du dictionnaire.

Syntaxe : `dictionnaire.copy()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
copie_dict = mon_dict.copy()
print(copie_dict)  # Affiche : {'nom': 'Alice', 'âge': 25}
```

Les dictionnaires

Les méthodes

Méthode `fromkeys()`

Description : Crée un nouveau dictionnaire avec les clés fournies et la même valeur pour toutes les clés.

Syntaxe : `dict.fromkeys(clés, valeur)`

Exemple :

```python
clés = ["nom", "âge", "ville"]
mon_dict = dict.fromkeys(clés, "Inconnu")
print(mon_dict)  # Affiche : {'nom': 'Inconnu', 'âge': 'Inconnu', 'ville': 'Inconnu'}
```

Les dictionnaires

Les méthodes

Méthode `get()`

Description : Retourne la valeur associée à une clé. Si la clé n'existe pas, retourne une valeur par défaut.

Syntaxe : `dictionnaire.get(clé, valeur_par_défaut)`

Exemple :

python Copier le code

```python
mon_dict = {"nom": "Alice", "âge": 25}
age = mon_dict.get("âge", "Non spécifié")
print(age)  # Affiche : 25
```

Les dictionnaires

Les méthodes

Méthode `items()`

Description : Retourne une vue dynamique contenant les paires clé-valeur du dictionnaire sous forme de tuples.

Syntaxe : `dictionnaire.items()`

Exemple :

python Copier le code

```python
mon_dict = {"nom": "Alice", "âge": 25}
print(mon_dict.items())  # Affiche : dict_items([('nom', 'Alice'), ('âge', 25)])
```

Les dictionnaires

Les méthodes

Méthode `keys()`

Description : Retourne une vue dynamique contenant les clés du dictionnaire.

Syntaxe : `dictionnaire.keys()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
print(mon_dict.keys())  # Affiche : dict_keys(['nom', 'âge'])
```

Les dictionnaires

Les méthodes

Méthode `pop()`

Description : Supprime et retourne la valeur associée à une clé. Si la clé n'existe pas, lève une exception `KeyError`, à moins qu'une valeur par défaut ne soit spécifiée.

Syntaxe : `dictionnaire.pop(clé, valeur_par_défaut)`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
age = mon_dict.pop("âge")
print(age)  # Affiche : 25
print(mon_dict)  # Affiche : {'nom': 'Alice'}
```

Les dictionnaires

Les méthodes

Méthode `popitem()`

Description : Supprime et retourne la dernière paire clé-valeur insérée dans le dictionnaire sous forme de tuple. Si le dictionnaire est vide, lève une exception `KeyError`.

Syntaxe : `dictionnaire.popitem()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
item = mon_dict.popitem()
print(item)  # Affiche : ('âge', 25)
print(mon_dict)  # Affiche : {'nom': 'Alice'}
```

Les dictionnaires

Les méthodes

Méthode `values()`

Description : Retourne une vue dynamique contenant les valeurs du dictionnaire.

Syntaxe : `dictionnaire.values()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
print(mon_dict.values())  # Affiche : dict_values(['Alice', 25])
```

Les dictionnaires

Les méthodes

Méthode `update()`

Description : Met à jour le dictionnaire avec les paires clé-valeur d'un autre dictionnaire ou d'une séquence de paires clé-valeur.

Syntaxe : `dictionnaire.update(autre_dictionnaire)`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
mise_a_jour = {"ville": "Paris", "âge": 26}
mon_dict.update(mise_a_jour)
print(mon_dict)  # Affiche : {'nom': 'Alice', 'âge': 26, 'ville': 'Paris'}
```

Les ensembles

Définition

En Python, un set (ensemble) est une collection non ordonnée d'éléments uniques. Les ensembles sont utiles pour stocker des valeurs uniques et effectuer des opérations ensemblistes comme l'union, l'intersection et la différence.

Les ensembles

Déclaration

```
# Création d'un set vide
mon_set = set()

# Création d'un set avec des éléments
mon_set = {1, 2, 3, 4}

# Les éléments en double sont automatiquement éliminés
mon_set = {1, 2, 2, 3, 4}  # Résultats : {1, 2, 3, 4}
```

Les ensembles

Les méthodes

```
# Création de sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Ajouter un élément
set1.add(6)

# Supprimer un élément
set1.discard(2)

# Vérifier l'appartenance
print(4 in set2)  # True
```

Les ensembles

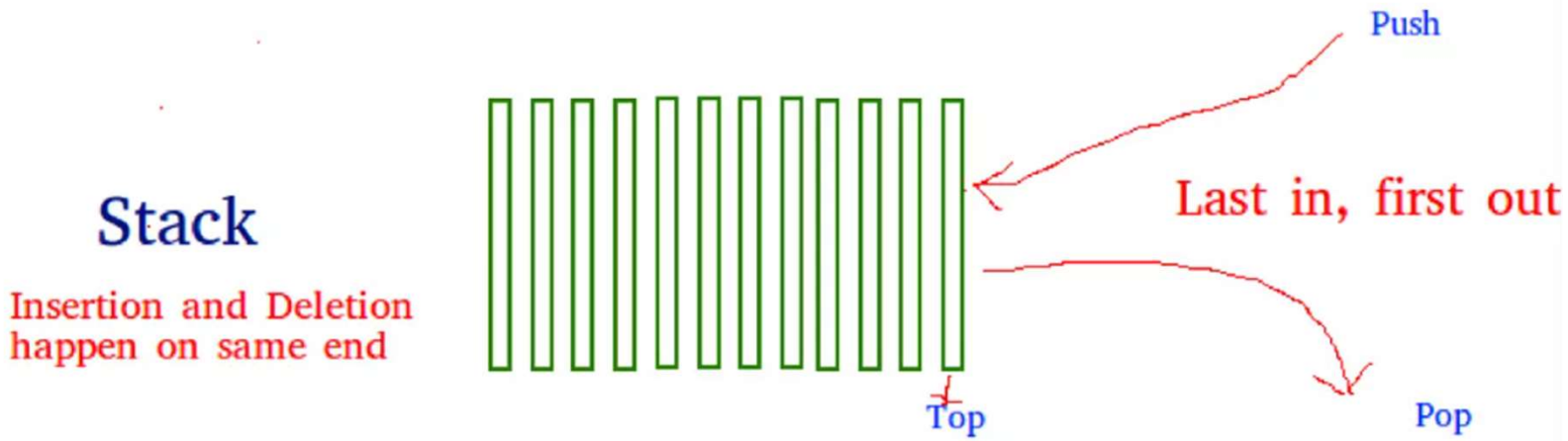
Les méthodes

```
# Opérations ensemblistes

union = set1.union(set2)
union = set1 | set2

intersection = set1.intersect(set2)
intersection = set1 & set2

difference = set1.minus(set2)
difference = set1 - set2

difference_symetrique = set1. symmetric_difference(set2)
difference_symetrique = set1 ^ set2
```

Les Piles

Définition

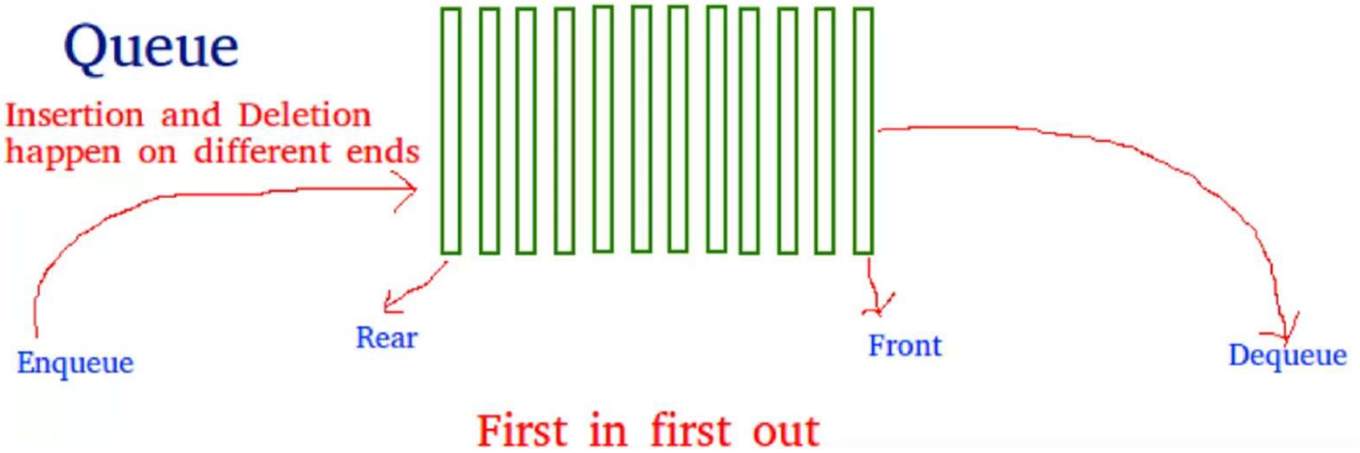
Une pile (ou stack en anglais) est une structure de données qui suit le principe LIFO (Last In, First Out), c'est-à-dire que le dernier élément ajouté est le premier à être retiré. Python ne fournit pas de structure de données dédiée pour les piles, mais on peut facilement implémenter une pile en utilisant des listes ou la classe **deque** du module **collections**.



Les Files

Définition

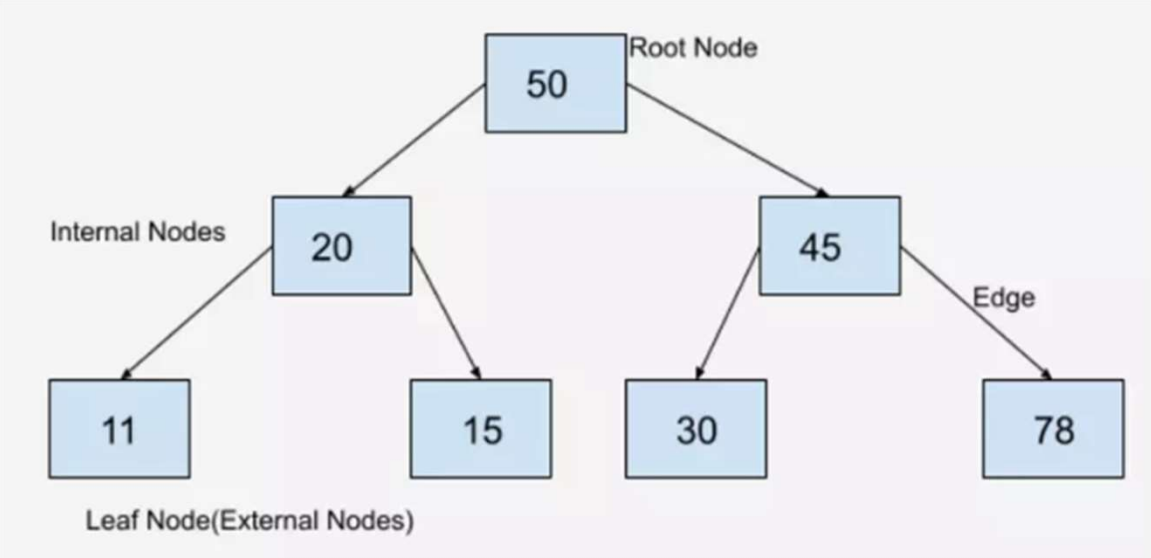
une file (ou queue en anglais) est une structure de données qui suit le principe FIFO (First In, First Out), ce qui signifie que le premier élément ajouté à la file est le premier à en être retiré. Contrairement aux piles (stacks), où le dernier élément ajouté est le premier à sortir, les files sont utilisées pour modéliser des systèmes où les éléments doivent être traités dans l'ordre de leur arrivée.



Les Arbres

Définition

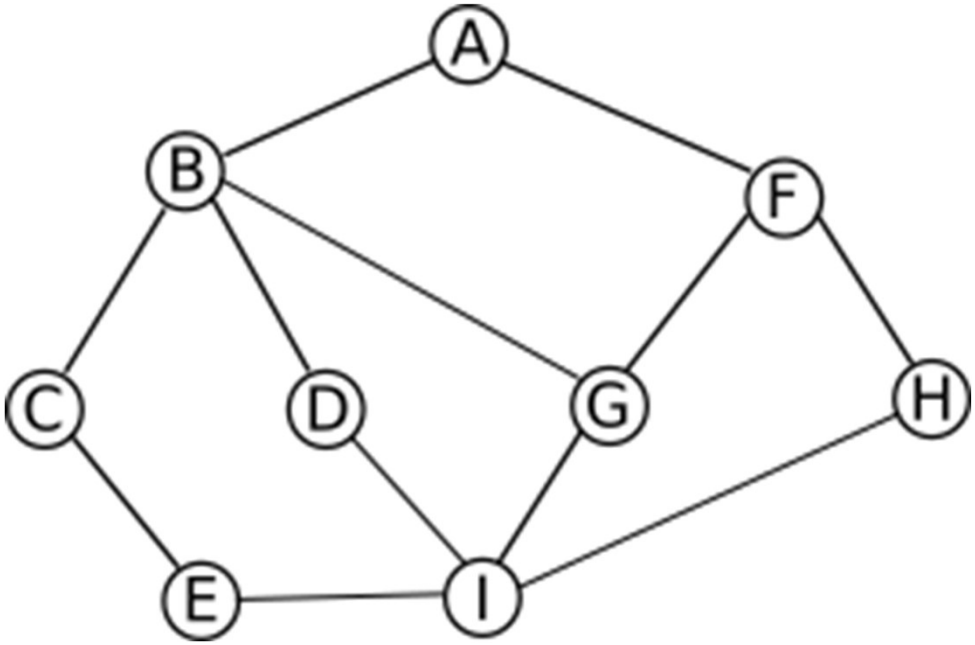
un **arbre** (ou **tree** en anglais) est une structure de données hiérarchique qui se compose de **nœuds** (ou **nodes**) reliés entre eux par des **arêtes** (ou **edges**). Les arbres sont utilisés pour représenter des relations hiérarchiques telles que des structures de dossiers, des organisations, des expressions arithmétiques, et bien plus encore.



Les Graphes

Définition

Un graphe est une structure de données composée de nœuds (ou sommets) et de liens (ou arêtes) qui relient ces nœuds. Les graphes sont utilisés pour représenter des relations entre des objets.



Comparaison

Le choix de la structure de données pour résoudre un problème dépend de plusieurs facteurs, notamment la nature des opérations que vous devez effectuer, les contraintes de performance, et les caractéristiques spécifiques de vos données. Voici quelques critères clés à considérer pour choisir la structure de données la plus adaptée :

Les structures de données

Comparaison

Type d'Opérations :

Recherche : Si vous avez besoin d'effectuer de nombreuses recherches rapides, les hashmaps (dictionnaires en Python) ou les ensembles sont souvent appropriés.
Insertion/Suppression : Les listes chaînées sont efficaces pour les insertions et suppressions fréquentes, tandis que les tableaux dynamiques (comme les listes en Python) sont efficaces pour les accès aléatoires.
Accès aléatoire : Les tableaux dynamiques ou les listes sont souvent utilisés pour un accès rapide à des éléments à des positions spécifiques.
Ordre des Éléments : Si l'ordre des éléments est important et doit être conservé, les listes ou les structures comme les OrderedDict en Python (avant Python 3.7) sont préférables.

Comparaison

Complexité Temporelle :

Tableaux (Listes en Python) : Accès rapide $O(1)$ par index, mais les insertions et suppressions peuvent être coûteuses si elles ne se font pas à la fin $O(n)$.

Listes Chaînées : Bonnes performances pour les insertions et suppressions $O(1)$ à un endroit donné, mais les accès par index sont $O(n)$.

Hashmaps/Dictionnaires : Accès, insertion, et suppression en moyenne $O(1)$, mais peuvent souffrir de collisions et ne conservent pas l'ordre des éléments (dans les versions antérieures à Python 3.7).

Arbres Binaires de Recherche : Accès, insertion, et suppression en $O(\log n)$ en moyenne, et maintiennent un ordre partiel des éléments.