OFPPT

مكتب التكوين المهني وإنعاش الشغل

Collections En Poo

Réalisée par

Safae **E**louatyq

Manal **B**enzakour

Ilham **S**ahmoudi

Fatima ezzahraa **H**adrouch

Encadrée par

Mr HAQAY

Sommaire

I. Introduction à la Programmation Orientée Objet (POO)

1. Définition de la POO

La Programmation Orientée Objet (POO) est un paradigme de programmation qui repose sur le concept d'"objets", des entités qui peuvent contenir à la fois des données (appelées attributs) et des méthodes (fonctions associées à ces données). En POO, le programme est conçu autour d'objets interagissant les uns avec les autres pour accomplir des tâches.

2. Principes Fondamentaux de la POO

La POO repose sur quatre principes fondamentaux :

- **Encapsulation :** L'encapsulation consiste à regrouper les données et les méthodes qui les manipulent dans une seule entité, l'objet. Cela permet de cacher les détails d'implémentation internes et de protéger les données de manipulation non autorisée.

- **Héritage :** L'héritage permet à un objet objet

objet de prendre les caractéristiques (attributs et méthodes) d'un autre objet de manière hiérarchique. Cela favorise la réutilisation du code et la création de relations entre les différents objets.

- **Polymorphisme :** Le polymorphisme permet à des objets de différentes classes d'être traités de manière uniforme. Cela signifie que le même nom de méthode peut être utilisé pour des objets de différentes classes, mais avec des implémentations différentes.

- **Abstraction** : L'abstraction consiste à représenter les caractéristiques essentielles d'un objet sans inclure les détails d'implémentation. Cela permet de se concentrer sur ce qui est important pour l'utilisateur sans se perdre dans les détails techniques.

II. Les Collections en Python

1. Qu'est-ce qu'une collection ?

Une collection en Python est un conteneur qui peut contenir un ensemble

d'éléments de données. Ces éléments peuvent être de différents types tels que des entiers, des chaînes de caractères, des tuples, des listes, des ensembles, des dictionnaires, etc. Les collections offrent des moyens efficaces de stocker, organiser et manipuler des données dans un programme.

3. Pourquoi les collections sont-elles importantes en programmation ?

Les collections sont importantes en programmation pour plusieurs raisons :

- **Stockage de Données :** Les collections offrent des structures de données flexibles pour stocker des ensembles d'éléments de manière organisée et efficace.

- **Optimisation des Performances :** Les collections sont optimisées pour offrir des performances élevées dans des

opérations courantes, ce qui les rend adaptées à une large gamme d'applications.

- Abstraction de Complexité : Les collections permettent aux développeurs de se concentrer sur la logique métier plutôt que sur les détails de gestion des données, ce qui simplifie le processus de développement et améliore la maintenabilité du code. En combinant les principes de la POO avec les fonctionnalités des collections en Python, les développeurs peuvent créer des programmes plus modulaires, flexibles et faciles à comprendre.

- Manipulation Facile des Données : Les opérations disponibles sur les collections permettent de manipuler facilement les données, comme l'ajout, la suppression, la recherche, le tri, etc.

III. La Classe OrderedDict :

1. Définition :

OrderedDict est une sous-classe de dictionnaire qui conserve l'ordre d'insertion des éléments. Contrairement aux dictionnaires standard, où l'ordre des éléments n'est pas garanti, **OrderedDict** conserve l'ordre dans lequel les éléments ont été ajoutés. Cela en fait un choix approprié lorsque l'ordre des éléments est important.

2. Syntaxe :

```
from collections import OrderedDict

# Création d'un OrderedDict vide
ordered_dict = OrderedDict()

# Création d'un OrderedDict avec des éléments initiaux
ordered_dict = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

3. Exemples :

```python
# Exemple d'utilisation d'OrderedDict
ordered_dict = OrderedDict([('a', 1), ('b', 2), ('c', 3)])

# Accès aux éléments dans l'ordre d'insertion
for key, value in ordered_dict.items():
    print(key, value)
# Résultat : a 1, b 2, c 3

# Modification de la valeur d'une clé
ordered_dict['b'] = 20

# Affichage après modification
print(ordered_dict)
# Résultat : OrderedDict([('a', 1), ('b', 20), ('c', 3)])
```

IV. La Fonction defaultdict

1. Définition :

defaultdict est une sous-classe de dictionnaire qui permet de spécifier une valeur par défaut pour les clés qui n'existent pas encore. Cela évite les erreurs de clé manquante et simplifie le code lors de l'utilisation de dictionnaires pour le comptage, l'indexation, etc.

2. Syntaxe :

```python
from collections import defaultdict

# Création d'un defaultdict avec une fonction par défaut
default_dict = defaultdict(int)
```

3. Exemples :

```
# Exemple d'utilisation de defaultdict pour le comptage des lettres dans
une chaîne
from collections import defaultdict

s = "programming"
letter_counts = defaultdict(int)

for letter in s:
    letter_counts[letter] += 1

print(letter_counts)
# Résultat : defaultdict(<class 'int'>, {'p': 1, 'r': 2, 'o': 1, 'g': 2,
'a': 1, 'm': 2, 'i': 1, 'n': 1})
```

V. La Fonction Counter :

1. Définition :

Counter est une sous-classe de dictionnaire conçue pour compter des objets hashables. Elle est utile pour effectuer rapidement des comptages d'éléments dans une collection, comme des listes ou des chaînes de caractères.

2. Syntaxe :

```
from collections import Counter

# Création d'un Counter à partir d'une séquence
counter = Counter(['a', 'b', 'a', 'c', 'b', 'a'])
```

3.Exemples :

```
# Exemple d'utilisation de Counter pour compter les éléments d'une liste
from collections import Counter

data = ['a', 'b', 'a', 'c', 'b', 'a']
counter = Counter(data)

print(counter)
# Résultat : Counter({'a': 3, 'b': 2, 'c': 1})

# Accès aux éléments les plus courants
print(counter.most_common(1))
# Résultat : [('a', 3)]
```

VI. La Fonction ChainMap :

1.Définition :

ChainMap est une classe qui permet de gérer plusieurs dictionnaires comme une seule unité. Elle maintient une liste ordonnée de dictionnaires et effectue des recherches dans chacun d'eux séquentiellement jusqu'à ce qu'une clé soit trouvée.

2.Syntaxe :

```
from collections import ChainMap

# Création d'une ChainMap
chain_map = ChainMap(dict1, dict2, dict3)
```

3.Exemples :

```
# Exemple d'utilisation de ChainMap pour combiner plusieurs dictionnaires
from collections import ChainMap

dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict3 = {'d': 5}

combined_dicts = ChainMap(dict1, dict2, dict3)

print(combined_dicts['b'])
# Résultat : 2 (obtenu à partir de dict1, car dict1 est le premier
dictionnaire dans la chaîne)
```

VII. La Fonction deque :

1.Définition :

deque est une double-ended queue (file à double extrémité) qui permet des opérations d'ajout et de retrait rapides aux deux extrémités de la file. Elle est utile pour les cas où vous devez ajouter ou retirer des éléments fréquemment à la fois en tête et en queue.

2.Syntaxe :

```
from collections import deque

# Création d'une deque vide
d = deque()

# Création d'une deque à partir d'une séquence
d = deque([1, 2, 3, 4, 5])
```

3.Exemples :

```
# Exemple d'utilisation de deque pour implémenter une file circulaire
from collections import deque

q = deque(maxlen=3)

q.append(1)
q.append(2)
q.append(3)
print(q)
# Résultat : deque([1, 2, 3], maxlen=3)

q.append(4)
print(q)
# Résultat : deque([2, 3, 4], maxlen=3)
```

VIII. La Fonction namedtuple :

1.Définition :

namedtuple est une fonction qui crée des sous-classes de tuple avec des noms de champs. Elle permet d'accéder aux éléments de manière nommée plutôt que par index, ce qui rend le code plus lisible et maintenable.

2.Syntaxe :

```
from collections import namedtuple

# Création d'une namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

3.Exemples :

```
# Exemple d'utilisation de namedtuple pour représenter des points dans un plan cartésien
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])

p = Point(1, 2)
print(p.x, p.y)
# Résultat : 1 2
```

IX. Utilisation des Collections dans la Pratique:

Les collections en Python offrent de nombreuses possibilités d'utilisation dans la pratique, en combinant les fonctionnalités de la programmation orientée objet (POO) avec des structures de données efficaces. Voici quelques exemples d'utilisation combinée et des bonnes pratiques associées.

1. Exemples d'utilisation combinée :

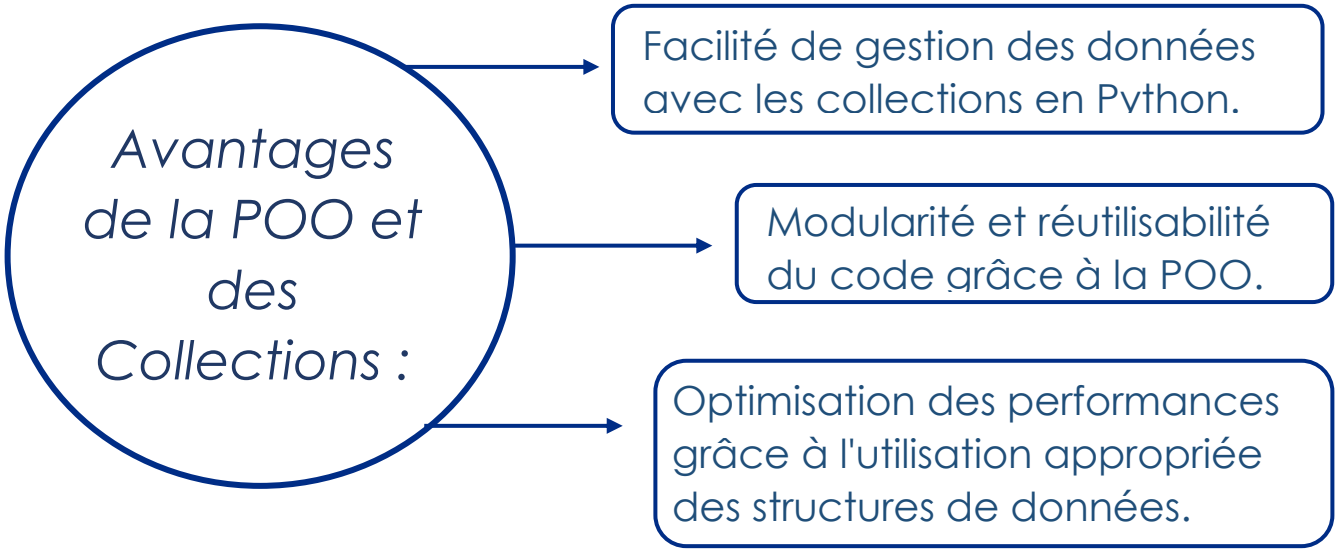
- **Gestion des Données Utilisateur :** Utiliser des dictionnaires pour stocker les informations des utilisateurs (nom, email, âge) et des listes pour gérer catégories d'utilisateurs (clients,

administrateurs, invités).

- **Analyse de Données :** Utiliser des **Counter** pour compter les occurrences de mots dans un texte, des **ChainMap** pour combiner plusieurs dictionnaires de paramètres de configuration, et des **defaultdict** pour regrouper les données par catégorie.

- **Traitement de Flux de Données :** Utiliser des **deque** pour implémenter une file circulaire pour le traitement des messages en temps réel, et des **namedtuple** pour représenter des points de données avec des noms de champs.

X. Conclusion



La maîtrise de la POO et des collections en Python est un élément essentiel de la boîte à outils de tout développeur.

En continuant à pratiquer, à expérimenter et à apprendre, vous pouvez devenir un développeur plus compétent et efficace dans la résolution de problèmes complexes.