ROYAUME DU MAROC

مكتب التكوين المهني وإنعاش الشغل

Office de la Formation Professionnelle et de la Promotion du Travail

OFPPT

DE TECHNOLOGIE APPLIQUEE NTIC2 SIDI MAAROUF CASABLANCA

RAPPORT MANIPULATION DES DONNEES (COLLECTIONS)

GROUPE 2

REALISE PAR :

AMINA NASSIR

OUMAIMA MAGHFOUR

OTHMANE LAKRICHI

KENZA HRAGUE

SOMMAIRE

COLLECTIONS

Le module collections contient des conteneurs de données spécialisés

La liste des conteneurs est la suivante

Conteneur	Utilité
Namedtuple	Une fonction permettant de créer une sous-classe de tuple avec des champs nommés
Deque	Un conteneur ressemblant à une liste mais avec ajout et suppression rapide à chacun des bouts
ChainMap	Permet de linker entre eux plusieurs mappings ensemble pour les gérer comme tout
Counter	Permet de compter les occurrences d'objet hachable
OrderedDict	Une sous classe de dictionnaire permettant de savoir l'ordre des entrées
defaultdict	Une sous classe de dictionnaire permettant de spécifier une valeur par défaut dans le constructeur

COLLECTIONS – NAMEDTUPLE

Définition

En Python, `namedtuple` est une fonctionnalité du module `collections` qui permet de créer des tuples nommés, c'est-à-dire des tuples dont les éléments sont accessibles via des noms de champs plutôt que par leur index numérique. Cela rend le code plus lisible et plus facile à comprendre.

Exemple

```python
from collections import namedtuple

# Définition d'une namedtuple nommée "Point" avec deux champs : "x" et "y"
Point = namedtuple('Point', ['x', 'y'])

# Création d'une instance de Point
p1 = Point(1, 2)

# Accès aux éléments de la namedtuple par leur nom de champ
print("Coordonnées de p1 :", p1.x, p1.y)

# Utilisation de la méthode _asdict() pour obtenir un dictionnaire des champs et de leurs valeurs
print("p1 sous forme de dictionnaire :", p1._asdict())

# Modification des valeurs des champs (les namedtuple sont immuables, donc cela crée une nouvelle instance)
p2 = p1._replace(x=5,y=3)
print("Nouvelles coordonnées de p2 :", p2.x, p2.y)
print(Point._fields)
#retourne les noms de champs affiche: ('x','y')
Color =namedtuple ('Color', 'red green, blue')
Pixel= namedtuple ('Pixel', Point._fields + Color._fields)
#on crée un nouveau tuple avec les champs de point et de color
print(Pixel(11,22,128,266,0))
 #affiche: Pixel(x=11, y=22, red=128, green=266, blue=0)
```

Dans cet exemple, nous avons créé une namedtuple appelée "**Point**" avec deux champs "x" et "y". Ensuite, nous avons créé une instance de cette namedtuple appelée "p1" avec les valeurs (1, 2). Nous avons accédé aux valeurs des champs en utilisant les noms de champs ("x" et "y"). Ensuite, nous avons utilisé la méthode `_**asdict**

() ` pour obtenir un dictionnaire des champs et de leurs valeurs. Ensuite, nous avons utilisé la méthode `**_replace ()** ` pour créer une nouvelle instance "p2" avec une valeur différente pour le champ "x". Enfin, nous avons utilisé la méthode ` **_mytuple**. _ **Fields** ` permet de récupérer les noms des champs de notre tuple. Elle est utile si on veut créer un nouveau tuple avec les champs d'un tuple existant.

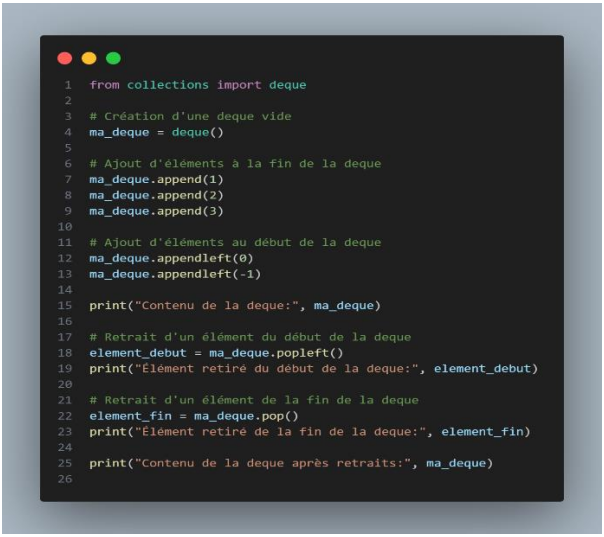
COLLECTIONS - DEQUE

Définition

La classe deque du module collections est un objet de type liste qui permet d'insérer des éléments au début ou à la fin d'une séquence avec une performance à temps constant (O (1)).

• O (1) performance signifie que le temps nécessaire pour ajouter un élément au début de de liste n'augmentera pas, même si cette liste a des milliers ou des millions d'éléments.

Exemple



```python
from collections import deque

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

# Ajout d'éléments à la fin de la deque
ma_deque.append(1)
ma_deque.append(2)
ma_deque.append(3)

# Ajout d'éléments au début de la deque
ma_deque.appendleft(0)
ma_deque.appendleft(-1)

print("Contenu de la deque:", ma_deque)

# Retrait d'un élément du début de la deque
element_debut = ma_deque.popleft()
print("Élément retiré du début de la deque:", element_debut)

# Retrait d'un élément de la fin de la deque
element_fin = ma_deque.pop()
print("Élément retiré de la fin de la deque:", element_fin)

print("Contenu de la deque après retraits:", ma_deque)
```

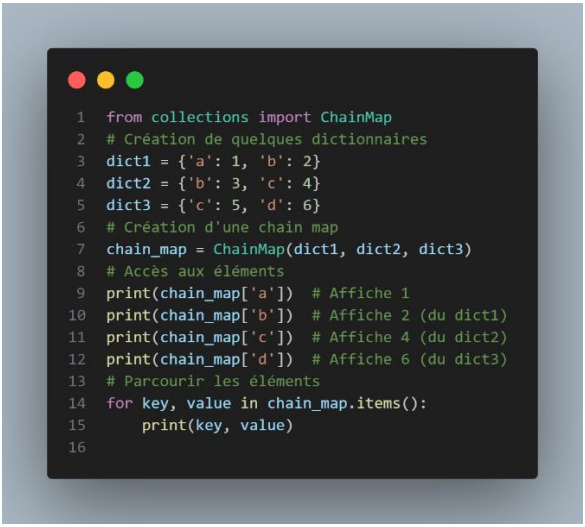
Dans cet exemple, une deque est créée à l'aide de la fonction `**deque ()** ` du module `**collections**`. Des éléments sont ajoutés à la fin de la deque à l'aide de la méthode `**append ()** `, et des éléments sont ajoutés au début avec `**appendleft ()** `. Ensuite, des éléments sont retirés du début avec `**popleft ()** ` et de la fin avec `**pop ()** `.

COLLECTIONS – CHAINMAP

Définition

Une `**chain map**` en Python est une structure de données fournie par le module `collections` qui permet de lier plusieurs dictionnaires pour les parcourir comme s'ils étaient un seul dictionnaire. Cela est utile lorsque vous avez plusieurs dictionnaires et que vous souhaitez les traiter ensemble sans les fusionner en un seul.

Exemple



```python
from collections import ChainMap
# Création de quelques dictionnaires
dict1 = {'a': 1, 'b': 2}
dict2 = {'b': 3, 'c': 4}
dict3 = {'c': 5, 'd': 6}
# Création d'une chain map
chain_map = ChainMap(dict1, dict2, dict3)
# Accès aux éléments
print(chain_map['a'])  # Affiche 1
print(chain_map['b'])  # Affiche 2 (du dict1)
print(chain_map['c'])  # Affiche 4 (du dict2)
print(chain_map['d'])  # Affiche 6 (du dict3)
# Parcourir les éléments
for key, value in chain_map.items():
    print(key, value)

```

Dans cet exemple, la `**chain map**` agit comme un seul dictionnaire qui contient les clés et les valeurs des trois dictionnaires. Lorsqu'une clé est recherchée, elle est d'abord cherchée dans le premier dictionnaire de la chaîne, puis dans le suivant, et ainsi de suite. Si une clé est présente dans plusieurs dictionnaires, seule la valeur de la première occurrence est retournée lors de l'accès.

COLLECTIONS – COUNTER

Définition

La classe collections. Counter est une sous-classe de dict. qui permet de compter des objets hachable.

• Enfait c'est un dictionnaire avec comme clé les éléments et comme valeurs leur nombre.

• class collections. Counter([iterable-or-mapping]) ceci retourne un Counter. L'argument permet de spécifier ce que l'on veut mettre dedans et qui doit être compté

Exemple

```python
from collections import Counter

# Exemple d'utilisation de Counter.elements()
# Une liste d'éléments
elements = ['a', 'b', 'c', 'a', 'b', 'a', 'd', 'c', 'a', 'b', 'b']

# Création d'un objet Counter à partir de la liste
element_counter = Counter(elements)

# Récupération de tous les éléments avec leurs occurrences
all_elements = list(element_counter.elements())
print("Tous les éléments avec leurs occurrences :", all_elements)

# Exemple d'utilisation de Counter.most_common()
# Affiche les elements les plus courants et leurs occurrences
print("Éléments les plus courants :", element_counter.most_common(2))

# Exemple d'utilisation de Counter.subtract()
# Liste d'éléments à soustraire
elements_to_subtract = ['a', 'b', 'b']

# Soustraction des éléments_to_subtract de l'élément_counter
element_counter.subtract(elements_to_subtract)

# Affichage du comptage mis à jour après la soustraction
print("Comptage après soustraction :", element_counter)
```

Dans cet exemple

- **Counter.elements()** renvoie un itérateur sur tous les éléments présents dans le compteur, en tenant compte de leurs occurrences. Dans notre exemple, cela renvoie la liste complète de tous les éléments avec leurs occurrences.

- **Counter**.**most_common**(**n**) renvoie une liste des n éléments les plus courants et de leurs occurrences, triés par ordre décroissant d'occurrence. Dans notre exemple, cela renvoie les deux éléments les plus courants avec leurs occurrences.

- **Counter**.**subtract**(**iterable**) soustrait les occurrences des éléments présents dans l'itérable fourni (dans notre exemple, la liste elements_to_subtract) du compteur. Cela signifie que les occurrences de ces éléments seront réduites dans le compteur.

COLLECTIONS – ORDREDDICT

Définition

Les collections. OrderedDict sont comme les dict. mais ils se rappellent l'ordre d'entrée des valeurs. Si on itère dessus les données seront retournées dans l'ordre d'ajout dans notre dict

Exemple

```
from collections import OrderedDict
d=OrderedDict()
d['a']='1' #remplir d
d['b']='2'
d['c']='3'
d['d']='4'
d.move_to_end('b')
print(d) #affiche OrderedDict([('a', '1'), ('c', '3'), ('d', '4'), ('b', '2')])
d.move_to_end('b',last=False)
print(d) #affiche OrderedDict([('b', '2'), ('a', '1'), ('c', '3'), ('d', '4')])
print(d.popitem(True)) #affiche ('d', '4')
print(d) #affiche OrderedDict([('b', '2'), ('a', '1'), ('c', '3')])
```

Dans cet exemple, La fonction **popitem(last=True)** : fait sortir une paire clé valeur de notre dictionnaire et si l'argument last est a 'True' alors les pairs seront retournés en LIFO sinon ce sera en FIFO.

• La fonction **move_to_end (key, last=True)** : permet de déplacer une clé à la fin de notre dictionnaire si last est à True sinon au début de notre dict

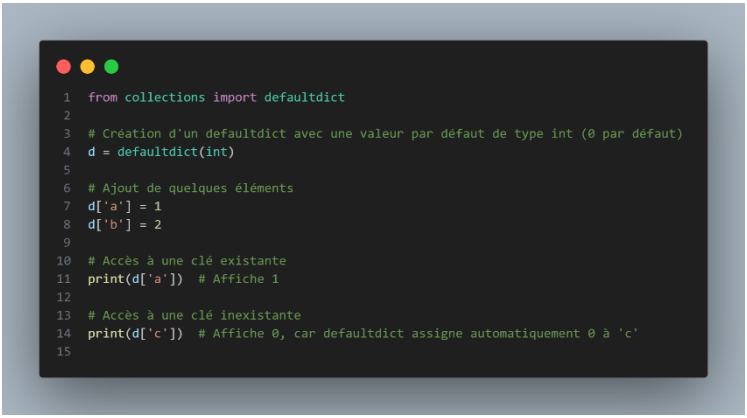
COLLECTIONS – DEFAULTDICT

Définition

Le module `**collections**` de Python inclut une classe très pratique appelée `**defaultdict**`, qui est une sous-classe de `dict`. La

particularité de `**defaultdict**` est qu'il attribue automatiquement une valeur par défaut à une clé si cette clé n'existe pas encore dans le dictionnaire. Cela évite les erreurs de clé manquante lorsqu'on essaie d'accéder à une clé inexistante.

Exemple



```python
from collections import defaultdict

# Création d'un defaultdict avec une valeur par défaut de type int (0 par défaut)
d = defaultdict(int)

# Ajout de quelques éléments
d['a'] = 1
d['b'] = 2

# Accès à une clé existante
print(d['a'])  # Affiche 1

# Accès à une clé inexistante
print(d['c'])  # Affiche 0, car defaultdict assigne automatiquement 0 à 'c'
```

Dans cet exemple, lorsque nous accédons à la clé `**'c'**`, qui n'existe pas encore dans le dictionnaire `d`, au lieu de lever une exception KeyError comme le ferait un dictionnaire normal, `**defaultdict**` ajoute automatiquement la clé `'c'` au dictionnaire avec une valeur par défaut de 0, puis renvoie cette valeur. Cela rend `**defaultdict**` très utile pour de nombreuses situations où vous travaillez avec des dictionnaires et où vous voulez éviter les erreurs de clé manquante.

CONCLUSION

La manipulation des données, dans le domaine des collections en informatique, englobe un ensemble d'opérations algorithmiques visant à gérer, organiser et transformer des ensembles de données structurées, telles que des listes, des tableaux, des ensembles ou des dictionnaires. Ces opérations peuvent inclure l'insertion, la suppression, la mise à jour et la recherche d'éléments au sein de la collection, ainsi que des tâches plus

avancées comme le tri, le filtrage, la réduction, le mapping et la transformation des données selon des critères spécifiques. La manipulation des données est fondamentale dans le développement logiciel et la science des données, permettant de traiter efficacement et efficacement des volumes importants d'informations pour répondre à divers besoins analytiques, computationnels et opérationnels.

➢ Formation par :

Mr. ÔTHMAN HAQQAY