Les Structures de Données Python
Cours 2 : structures de données non linéaire
(les sets et les dictionnaires)

Halim Djerroud

ESIEE
PARIS

révision : 0.1

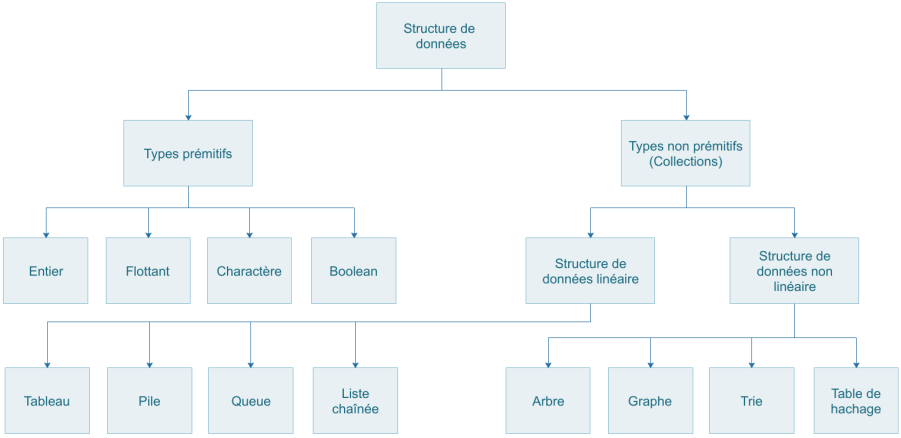
Plan

Les types en python

En fonction du nombre d'éléments stockés dans une variable, il existe deux catégories de types :

1. Les types primitifs
   - int, float, bool, str
2. Les types non primitifs (collections)
   - Structures de données non linéaire
     - **Set**
     - **Dictionnaire**
   - Séquence (Structure de données linéaire)
     - Liste
     - Tuples
     - String

Les types en python



Structure de données non linéaire

- Il organise les données dans un ordre trié et il existe une relation entre les éléments
- Impossible de parcourir les données en une seule fois.

Définition : Les sets

- Pour créer un ensemble (set) on utilise les accolades **{}**
- Un ensemble est une collection d'éléments **non ordonnée** et sans index
- Ne peut pas contenir de **doublons** (tous les éléments sont uniques)
- pour créer un ensemble vide il on utilise la fonction **set()** (*car la syntaxe {} va créer un dictionnaire vide et non pas un ensemble vide*)

Exemple :

```
# Creation et initialisation d'un set
ens_1 = {"Lea", 5, "Tennis"}

# Creation d'un set vide
ens_vide = set()
```

Utilité des Set

- Éliminer les doublons dans un ensemble de données

```
ens_1 = {"Lea", "Julie", "Thomas", "Julie", "Lea"}
print(ens_1)  # => {'Lea', 'Thomas', 'Julie'}
```

- Tester l'appartenance d'un élément à un ensemble de données : Dispose d'une méthode hautement optimisée pour vérifier si un élément spécifique est contenu dans l'ensemble

```
ens_1 = {"Lea", "Julie", "Thomas", "Julie", "Lea"}
print("Lea" in ens_1)  # => True
print("Lora" in ens_1) # => False
```

Utilité des Set

- Union : avec l'opérateur **|**

```
ens_1 = {"Lea", "Julie"}
ens_2 = {"Thomas", "Julie", "Lea"}
print(ens_1 | ens_2)   # => {'Julie', 'Lea', 'Thomas'}
```

- Intersection : avec l'opérateur **&**

```
ens_1 = {"Lea", "Julie"}
ens_2 = {"Thomas", "Julie", "Lea"}
print(ens_1 & ens_2)   # => {'Julie', 'Lea'}
```

- Différence : avec l'opérateur **-**

```
ens_1 = {"Lea", "Julie"}
ens_2 = {"Thomas", "Julie", "Lea"}
print(ens_2 - ens_1)   # => {'Thomas'}
```

Parcourir une liste (boucle)

- Il est possible de parcourir les éléments d'un set grace à la boucle **for** en utilisant l'index des éléments

Exemple :

```
villes = {"Paris", "Brest", "Lyon", "Marseille"}
for ville in villes:
    print(ville, end=" ")

# => Paris Brest Lyon Marseille
```

Remarque : *Les ensembles ne sont pas ordonnés, nous ne pouvons pas accéder aux éléments en utilisant des index comme nous le faisons dans les listes.*

Les méthodes des sets

Il existe de nombreuses méthodes applicables sur les sets parmi elles, on trouve :

Méthode	**Decription**
add(x)	Ajouter l'élément x au set
len()	retourne nombre d'éléments dans un set

Ajout d'un élément

- L'ajout d'un élément se fait par la fonction ***add()***

```
majuscules = {'A', 'B', 'C'}
print(majuscules)        # => {'B', 'C', 'A'}
majuscules.add('D')
# La lettre D sreabien ajoutee
print(majuscules)         # => {'B', 'D', 'C', 'A'}
majuscules.add('C')
# L'ajout de la lettre C sera sans effet car existe deja
print(majuscules)         # => {'B', 'D', 'C', 'A'}
```

nombre d'éléments

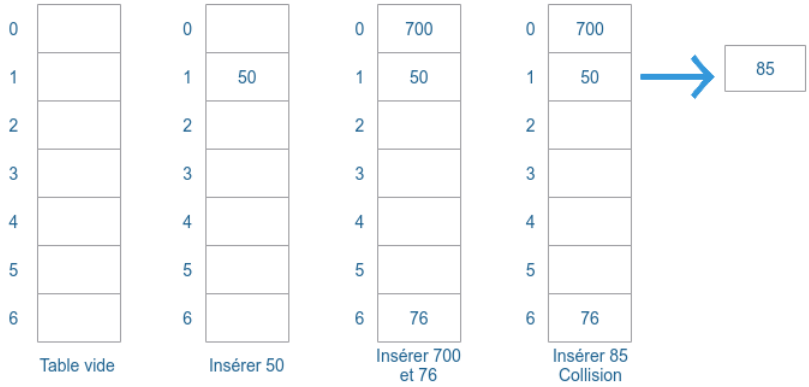
- Le nombre d'éléments dans un set s'obtient grâce à la fonction ***len()***

```
majuscules = {'A', 'B', 'C'}
print(len(majuscules))       # => {'B', 'C', 'A'}
majuscules.add('D')
# La lettre D sreabien ajoutee
print(len(majuscules))       # => {'B', 'D', 'C', 'A'}
majuscules.add('C')
# L'ajout de la lettre C sera sans effet
print(len(majuscules))       # => 4
```

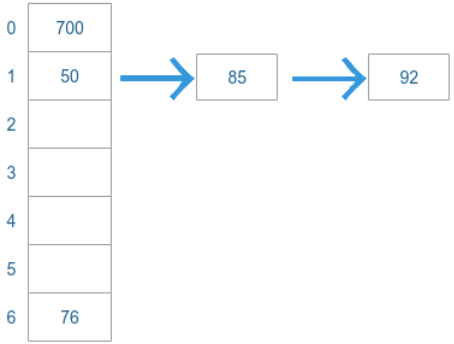
Fonctionnement interne de Set

- Basé sur une **table de hachage**
- Si plusieurs valeurs sont présentes à la même position d'index, la valeur est ajoutée à cette position d'index pour former une liste chaînée

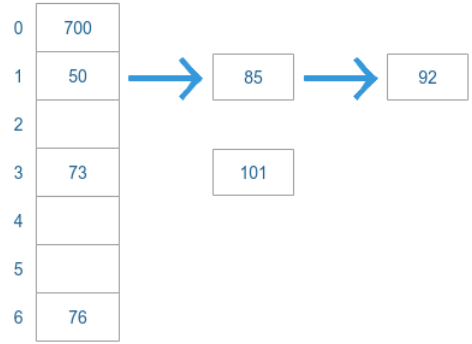
Fonctionnement table de hachage



Fonctionnement table de hachage



Insérer 92



Insérer 73 et 101

Complexité set

Opération	Cas moyen	Pire cas
x in S	O(1)	
copy	O(n)	
insert	O(1)	O(n)
Get item	O(1)	O(1)
Del item	O(n)	O(n)
s | t (union)	O(min(len(s), len(t))	O(len(s) * len(t))
s1&s2&..&sn		(n-1)*O(l) où l est max(len(s1),..,len(sn))
s - t (différence)	O(len(s))	

Les dictionnaires

- En python les tables de hachage sont appelées dictionnaires
- Un dictionnaire est une autre forme de structures de données qui se crée en utilisant les accolades **{}**
- La différence avec le set est que les éléments d'un dictionnaires sont composés de paires → **clef : valeur**
- Les éléments d'un dictionnaires **ne sont pas ordonnées** → Pas de notion d'indice pour accéder aux valeurs
- Dans un même dictionnaires, **les clés sont uniques** : il n'est pas possible d'avoir deux couples avec la même clé
- Les valeurs dans un dictionnaires **ne sont pas** forcément **uniques**

Exemple : Dictionnaire

Exemple :

```
dict_1 = {}          # dictionnaire vide

# dictionnaire avec des cles entieres
dict_1 = {1: 'Fruit', 2: 'Balon'}

# dictionnaire avec des cles mixtes
dict_1 = {'nom': 'Lea', 1: [2, 4, 3]}

# en utilisant dict()
dict_1 = dict({1:'Fruit', 2:'Balon'})

# avec une cle sequence ayant chaque element par paire
dict_1 = dict([(1,'Fruit'), (2,'Balon')])
# => {1: 'Fruit', 2: 'Balon'}
```

Accéder aux éléments du dictionnaire

- Il n'est pas possible d'utiliser des indices
- L'accès à une valeur se fait par sa **clé**

Exemple :

```
# get vs [] pour recuperer des elements
dict_1 = {'nom': 'Lea', 'age': 5}
print(dict_1['nom'])       # => Lea
print(dict_1.get('age'))   # => 5

# Essayer d'acceder a des cles qui
# n'existent pas genere une erreur
print(dict_1.get('adresse'))   # => None
print(dict_1['adresse'])       # => KeyError: 'adresse'
```

Composition d'un dictionnaire : (1) les items

- **Items** (les paires **clé :valeur**) : accessibles via la méthode wma en utilisant la syntaxe :

```
nom_dict.items()
```

```
jours = {1: "lundi", 2:"mardi", 3:"mercredi", 4:"jeudi",
         5:"vendredi", 6:"samedi", 7:"dimanche"}
print(jours.items())
# =>
dict_items([(1, 'lundi'),      (2, 'mardi'),
            (3, 'mercredi'), (4, 'jeudi'),
            (5, 'vendredi'), (6, 'samedi'),
            (7, 'dimanche')])
```

Composition d'un dictionnaire : (2) les clés

- **Clés** : accessibles via la méthode `keys()` en utilisant la syntaxe :

```
nom_dict.keys()
jours = {1: "lundi", 2:"mardi", 3:"mercredi", 4:"jeudi",
         5:"vendredi", 6:"samedi", 7:"dimanche"}
print(jours.keys())
# =>
dict_keys([1, 2, 3, 4, 5, 6, 7])
```

Composition d'un dictionnaire : (3) les Valeurs

- **Valeurs** : accessibles via la méthode `values()` en utilisant la syntaxe :

```
nom_dict.values()
jours = {1: "lundi", 2:"mardi", 3:"mercredi", 4:"jeudi",
         5:"vendredi", 6:"samedi", 7:"dimanche"}
print(jours.keys())
# =>
dict_values(['lundi', 'mardi', 'mercredi', 'jeudi',
             'vendredi', 'samedi', 'dimanche'])
```

Parcourir un dictionnaire : Items

- Par **Items**

```
jours = {1: "lundi", 2:"mardi", 3:"mercredi", 4:"jeudi",
         5:"vendredi", 6:"samedi", 7:"dimanche"}
for jour in jours.items():
    print(jour) # affichage des items sous forme de tuples

# =>
# (1, 'lundi')
# (2, 'mardi')
# (3, 'mercredi')
# (4, 'jeudi')
# (5, 'vendredi')
# (6, 'samedi')
# (7, 'dimanche')
```

Parcourir un dictionnaire : clés

- Par **clés**

```
jours = {1: "lundi", 2:"mardi", 3:"mercredi", 4:"jeudi",
         5:"vendredi", 6:"samedi", 7:"dimanche"}
for jour in jours.keys():
    # affichage des valeurs correspondantes aux cles
    print(jour, end = ",")

# => 1,2,3,4,5,6,7,
```

Parcourir un dictionnaire : valeurs

- Par **valeurs**

```
jours = {1: "lundi", 2:"mardi", 3:"mercredi", 4:"jeudi",
         5:"vendredi", 6:"samedi", 7:"dimanche"}
for jour in jours.values():
    # affichage des valeurs correspondantes aux cles
    print(jour, end = ",")

# => lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche,
```