MARO 037 - Competitive Programming

Fabian Lecron - Arnaud Vandaele

Faculté Polytechnique - Services MIT et MARO

UMONS

Année académique 2024-2025

Séance 02 :
Les structures de données

Séance 02 – Les structures de données

Tables des matières

Les structures de données en Python

1. Les structures de données natives
   - Type None : `None`
   - Types numériques : `int`, `float`, `complex`, `bool`
   - Types séquentiels : `str`, `list`, `tuple`, `range`
   - Types d'ensembles : `set`, `frozenset`
   - Types de correspondances : `dict`
2. Le module `collections` offrant des structures de données plus spécialisées que les structures natives
3. Le module `array` permettant de manipuler des tableaux
4. L'extension `NumPy` proposant des structures de données adaptées au calcul scientifique

Plan

Une liste est une séquence d'éléments **contigus** en mémoire

- `list` : type de données natif dans Python
- Une liste est un objet muable (que l'on peut modifier après sa création)
- Les listes sont implémentées via des tableaux
- L'accès à un élément d'une liste se fait via l'opérateur `[]` avec une complexité $\mathcal{O}(1)$
- Les éléments sont des objets qui peuvent être de types différents dans une même liste

```
liste=[0,1,'deux',[3,4],5]
```

Parcourir une liste avec une boucle `for`

- Accéder aux éléments

```
>>> chiffres = ['un','deux','trois']
>>> for chiffre in chiffres:
...     print(chiffre)
'un'
'deux'
'trois'
```

- Modifier les éléments

```
>>> chiffres = ['un','deux','trois']
>>> for i in range(len(chiffres)):
...     chiffres[i] = i+1
>>> print(chiffres)
[1,2,3]
```

Complexité des opérations élémentaires sur les listes

Opération	Complexité
`x = list(y)`	$\mathcal{O}(n)$
`a=x[i]`	$\mathcal{O}(1)$
`x[i]=a`	$\mathcal{O}(1)$
`z=x+y`	$\mathcal{O}(n)$
`x.append(a)`	$\mathcal{O}(1)^*$
`x.insert(i,e)`	$\mathcal{O}(n)$

Opération	Complexité
`del x[i]`	$\mathcal{O}(n)$
`x == y`	$\mathcal{O}(n)$
`for a in x:`	$\mathcal{O}(n)$
`len(x)`	$\mathcal{O}(1)$
`a in x`	$\mathcal{O}(n)$
`x.sort()`	$\mathcal{O}(n \log n)$

*Complexité *amortie*

La compréhension de liste : une manière concise de construire des listes

- Les éléments de la liste sont le résultat d'une opération appliquée aux éléments d'une autre séquence

```
>>> liste = [i**2 for i in range(1,5)]
>>> print(liste)
[1, 4, 9, 16]
```

- L'opération peut être la vérification d'une condition

```
>>> liste = [i for i in [-2,-1,0,1,2] if i>0]
>>> print(liste)
[1,2]
```

- Des structures plus complexes peuvent être créées à partir d'autres objets

```
>>> mots = 'voici une phrase'.split()
>>> print([[mot,len(mot)] for mot in mots])
[['voici', 5], ['une', 3], ['phrase', 6]]
```

Un tuple est une séquence immuable d'éléments

- `tuple` : type caractérisant les tuples, des séquences similaires aux listes et partageant la plupart des méthodes
- Les tuples sont des séquences d'éléments séparés par des virgules. Il est conseillé de les délimiter par des parenthèses

```
tple = (0,1,'deux',(3,4),5)
```

- La fonction `tuple()` transforme une séquence en tuple d'éléments de cette séquence

```
>>> tuple('competitive')
('c', 'o', 'm', 'p', 'e', 't', 'i', 't', 'i', 'v', 'e')
```

- La création d'un tuple contenant un seul élément nécessite une virgule

```
tple = (1,)
```

- Une utilisation importante des tuples est l'assignement de plusieurs variables en une fois

```
liste=[1,2]
x,y=liste
```

- Les tuples sont hachables et peuvent servir en tant que clefs pour les dictionnaires

Les ensembles sont des collections d'éléments uniques et hachables

- `set` : type caractérisant des ensembles **muables** interdisant les doublons et sans ordre

```
>>> set1={0,1,'deux',(3,4),5}
>>> print(set1)
{(3, 4), 0, 1, 5, 'deux'}
```

- `frozenset` : type caractérisant des ensembles **immuables** interdisant les doublons et sans ordre. Ils peuvent être utilisés en tant que clefs pour les dictionnaires

```
>>> frozenset1=frozenset({0,1,'deux',(3,4),5})
>>> print(frozenset1)
frozenset({0, 1, 5, 'deux', (3, 4)})
```

- Les éléments d'un `set` et d'un `frozenset` doivent être immuables (ils sont donc hachables)
- `frozenset` partage les méthodes de `set` qui ne modifient pas l'objet
- On accède facilement aux éléments avec une boucle `for`

```
for elem in set1: print(elem)
```

Les ensembles pour des opérations bien spécifiques

```
set1={0,1,'deux',(3,4),5}
set2={'zero',1,2,(3,4),'cinq'}
```

Quelques opérations à partir des deux objets définis ci-dessus

- Le test d'appartenance

```
>>> 1 in set1
True
```

- L'union

```
>>> set1.union(set2) # ou set1|set2
{(3, 4), 0, 1, 2, 5, 'cinq', 'deux', 'zero'}
```

- L'intersection

```
>>> set1.intersection(set2) # ou set1&set2
{(3, 4), 1}
```

- La différence

```
>>> set1.difference(set2) # ou set1-set2
{0, 5, 'deux'}
```

- La différence symétrique

```
>>> set1.symmetric_difference(set2) # ou set1^set2
{0, 2, 5, 'cinq', 'deux', 'zero'}
```

Complexité de quelques opérations sur les ensembles

- Dans la suite, `s` est un objet de type `set` et le paramètre `t` peut être tout objet Python supportant l'itération
- n peut prendre les valeurs suivantes : `len(s)`, `len(t)`, `len(s)+len(t)`

Opération	Complexité
`s=set(t)`	$\mathcal{O}(n)$
`e in s`	$\mathcal{O}(1)$
`s.add(e)`	$\mathcal{O}(1)$
`s.remove(e)`	$\mathcal{O}(1)$
`s.union(t)`	$\mathcal{O}(n)$
`s.intersection(t)`	$\mathcal{O}(n)$
`s.difference(t)`	$\mathcal{O}(n)$
`s.symmetric_difference(t)`	$\mathcal{O}(n)$

Des complexités possibles grâce au principe de hachage

- La fonction `hash()` dans Python transforme un objet en un entier

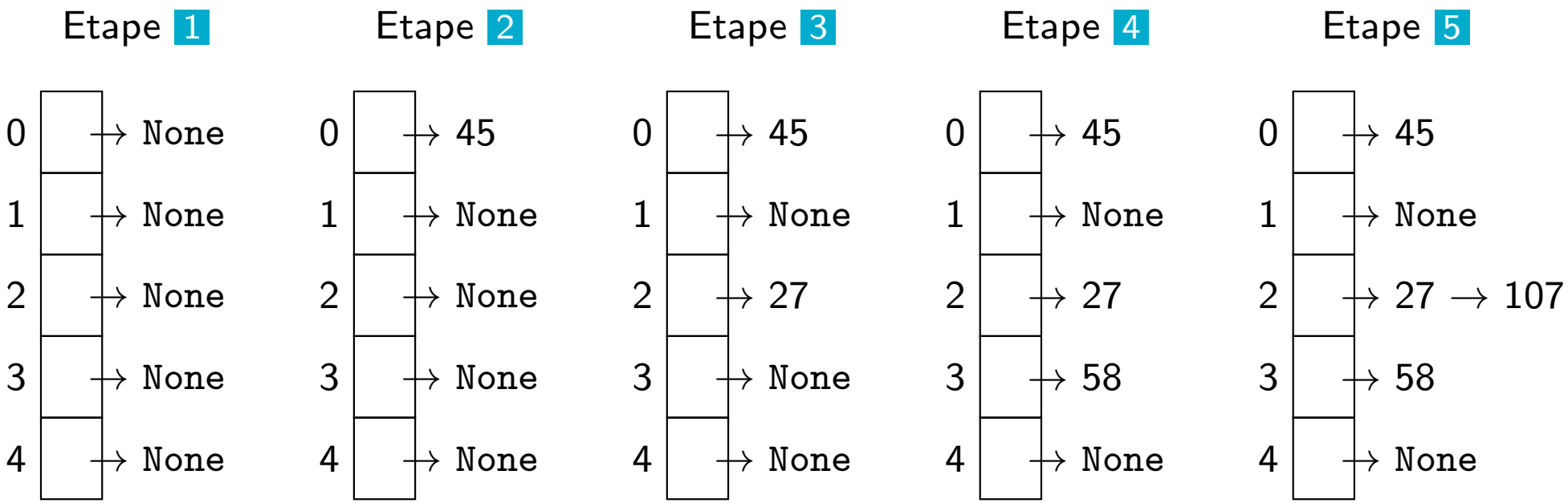
```
>>> hash(45)
45

>>> hash('a')
4922047246574185236
```

- La création d'un objet `set` vide se fait en créant une liste contenant un certain nombre x de valeurs de type `None`
- Pour ajouter un objet dans le `set`, il faut d'abord calculer son `hash` et ensuite convertir la valeur obtenue en un indice i compris entre 0 et $x - 1$, en utilisant le reste de la division du `hash` par x
- La recherche d'un élément dans le `set` se réalise en $\mathcal{O}(1)$ puisque son `hash` nous donne sa position
- En pratique, deux `hash` différents peuvent donner le même indice. Deux objets devant être stockés au même indice sont stockés à l'aide d'une liste chaînée
- Les objets de type `set` sont gérés de sorte que cela arrive le moins souvent possible. La complexité $\mathcal{O}(1)$ pour la recherche d'un élément est donc une complexité amortie

Illustration du principe de hachage pour l'indexation des ensembles

1. Création d'une liste contenant 5 éléments de type `None`
2. Ajout de l'entier 45 à la position 0 (`hash(45)%5` $= 0$)
3. Ajout de l'entier 27 à la position 2 (`hash(27)%5` $= 2$)
4. Ajout de l'entier 58 à la position 3 (`hash(58)%5` $= 3$)
5. Ajout de l'entier 107 à la position 2 via une liste chaînée (`hash(107)%5` $= 2$)



Les dictionnaires sont des collections faisant correspondre des clefs uniques à des valeurs

- `dict` : type caractérisant les dictionnaires, des collections d'objets (`value`) indexés par d'autres objets immuables (`key`) tels que des nombres, des chaînes de caractères, des tuples, etc.
- Les dictionnaires sont des objets muables
- Les dictionnaires sont créés à l'aide de la syntaxe {`key:value`} et l'opérateur `[key]` permet d'accéder aux valeurs

```
>>> dictio={'un': 1, 'deux': 2, 'trois': 3}
>>> dictio['quatre']=4
>>> dictio.update({'cinq': 5})
>>> print(dictio)
{'un': 1, 'deux': 2, 'trois': 3, 'quatre': 4, 'cinq': 5}
```

- La recherche d'une valeur via sa clef s'opère en $\mathcal{O}(1)$ grâce au principe de hachage

```
>>> 'deux' in dictio
True
```

- La compréhension de dictionnaire est possible

```
>>> {clef:valeur for (clef,valeur) in dictio.items() if valeur>3}
{'quatre': 4, 'cinq': 5}
```

Trier un dictionnaire selon la clef ou la valeur

```
dictio={'un': 1, 'deux': 2, 'trois': 3,'quatre': 4, 'cinq': 5}
```

Quelques opérations à partir du dictionnaire défini ci-dessus

- Trier les clefs du dictionnaire

```
>>> sorted(dictio)
['cinq', 'deux', 'quatre', 'trois', 'un']
```

- Trier les valeurs du dictionnaire

```
>>> sorted(dictio.values())
[1, 2, 3, 4, 5]
```

- Ordonner les clefs du dictionnaire en fonction du tri des valeurs

```
>>> sorted(dictio,key=dictio.__getitem__)
['un', 'deux', 'trois', 'quatre', 'cinq']
```

- Ordonner de façon décroissante les clefs du dictionnaire en fonction du tri des valeurs

```
>>> sorted(dictio,key=dictio.__getitem__,reverse=True)
sorted(dictio,key=dictio.__getitem__,reverse=True)
```

Complexité de quelques opérations sur les dictionnaires

- Dans la suite, `d` est un objet de type `dict`, `k` est une clef, et `v` est une valeur

Opération	Complexité
`d={(k:v)}`	$\mathcal{O}(n)$
`k in d`	$\mathcal{O}(1)$
`d[k]=v`	$\mathcal{O}(1)$
`del d[k]`	$\mathcal{O}(1)$
`d.items()`	$\mathcal{O}(1)$
`d.keys()`	$\mathcal{O}(1)$
`d.values()`	$\mathcal{O}(1)$
`for k in d:`	$\mathcal{O}(n)$

Plan

deque pour implémenter les piles et les files

- deque : type caractérisant une liste doublement chaînée, où chaque objet de la liste *connaît* l'adresse de l'objet suivant et du précédent
- Structure de données adaptée pour des insertions et des suppressions aux extrémités mais pas pour les accès aux éléments
- Les files appliquent le principe du First In First Out : une liste chaînée avec insertion au début et suppression à la fin est une file
- Les piles appliquent le principe du Last In First Out : une liste chaînée avec insertion à la fin et suppression à la fin est une pile
- Le type `deque` est accessible en important le module `Collections`

```
>>> from collections import deque
>>> dq = deque('abc')
```

- Ajout au début et à la fin via les fonction `append()` et `appendleft()`

```
>>> dq.append('d')
>>> dq.appendleft('z')
```

- Suppression au début et à la fin via les fonction `pop()` et `popleft()`

```
>>> dq.pop()
'd'
```

```
>>> dq.popleft()
'z'
```

Complexité des opérations élémentaires sur les deques

Opération	Complexité
`x = deque(y)`	$\mathcal{O}(n)$
`a=x[i]`	$\mathcal{O}(n)$
`x[i]=a`	$\mathcal{O}(n)$
`x.append(a)`	$\mathcal{O}(1)$
`x.appendleft(a)`	$\mathcal{O}(1)$
`x.pop()`	$\mathcal{O}(1)$

Opération	Complexité
`x.popleft()`	$\mathcal{O}(1)$
`x.insert(i,e)`	$\mathcal{O}(n)$
`del x[i]`	$\mathcal{O}(n)$
`for a in x:`	$\mathcal{O}(n)$
`len(x)`	$\mathcal{O}(1)$
`a in x`	$\mathcal{O}(n)$

Plan

array : une séquence plus efficace que la liste pour la gestion de la mémoire

- array : type similaire aux tableaux en C, définissant une séquence similaire à une liste mais où tous les éléments sont du même type
- Le type des éléments contenus dans le tableau est précisé à la création

```
>>> import array as arr
>>> tableau = arr.array('i',range(5)) #i pour entier signé
>>> print(tableau)
array('i', [0, 1, 2, 3, 4])
```

- Les méthodes classiques propres aux listes existent pour les tableaux
- Les tableaux consomment moins de mémoire par rapport aux listes

```
>>> import array as arr
>>> import sys
>>> tableau = arr.array('i',range(1000000))
>>> liste = list(range(1000000))
>>> 100*sys.getsizeof(tableau)/sys.getsizeof(liste)
45.46534532014713
```

Plan

Le package NumPy pour le calcul scientifique

- Le package NumPy introduit un type de données array spécialement optimisé pour le calcul scientifique
- Importation du package

```
import numpy as np
```

- Deux façons usuelles de créer des tableaux dans NumPy

```
>>> np.arange(4)
array([0, 1, 2, 3])
>>> np.arange(2, 4, dtype=float)
array([2., 3.])
```

```
>>> np.array([0,1,2,3])
array([0, 1, 2, 3])
>>> np.array((0,1,2,3))
array([0, 1, 2, 3])
```

- La méthode np.sort(...) permet de choisir son algorithme de tri entre : quicksort, mergesort, heapsort, stable

```
>>> tableau = np.array([5,4,3,2,1])
>>> print(tableau)
[5 4 3 2 1]
>>> tableau.sort(kind='mergesort')
>>> print(tableau)
[1 2 3 4 5]
```

Plan

Exercice 1 : compter les occurrences des mots d'un texte

Input

- Un fichier de votre choix

Exemple :

```
Meunier tu dors
Ton moulin ton moulin va trop vite
Meunier tu dors
Ton moulin ton moulin va trop fort
```

Output

- Chaque ligne du fichier contiendra chaque mot et le nombre de fois que le mot apparaît dans le texte

Pour notre exemple :

```
Meunier : 2 fois
tu : 2 fois
dors : 2 fois
Ton : 2 fois
moulin : 4 fois
ton : 2 fois
va : 2 fois
trop : 2 fois
vite : 1 fois
fort : 1 fois
```