UNIVERSITÉ TOULOUSE III PAUL SABATIER

Université de Toulouse

Faculté des Sciences et d'Ingénierie

SDD

CTD 2 : Tableaux et matrices

Algorithmique

Semestre 2 - 2019-2020

À toute variable est associée un **type** qui permet au traducteur du langage (compilateur ou interpréteur) de lui affecter la **place nécessaire en mémoire**.

Type

Un **type** définit :

- Un **espace de valeurs**.
- Un **ensemble d'opérations autorisées**.

Remarque : En Python, il n'y a pas de déclaration explicite, elle découle de l'affectation d'une valeur et le type de la valeur détermine le type de la variable.

Exemple : le type `int` Python

- Espace de valeurs : $\mathbb{N}$.
- Opérations autorisées : +, - , % , -= , += , etc.

Exemple : le type `int64` de NumPy

- Espace de valeurs : $-2^{63} \leq V \leq 2^{63} - 1$
  $-9223372036854775808 \leq V \leq 9223372036854775807$
- Opérations autorisées : +, - , % , -= , += , etc.

Abstraction de données

De manière plus générale on appelle **abstraction de données** la définition d'un type de données avec la description des **opérations qu'il est possible de lui appliquer**, indépendamment de la manière dont cela sera programmé.

La **description de chaque opération** précise :

- Ses entrées.
- Les préconditions qui doivent être vérifiées pour que l'opération s'exécute correctement.
- Ses sorties.
- Les post conditions vraies après exécution de l'opération.

Structure de données

Une **structure de données** est décrite en présentant :

- L'**abstraction de données** qui la définit.
- Son **implémentation**, c'est-à-dire sa mise en oeuvre matérielle.

Exemple

Nous verrons dans ce cours la structure de données "Matrice" :

- L'utilisateur de ce type de structure n'est concerné que par les opérations qu'il peut effectuer avec, ici « comment accéder à l'élément (i, j) de la matrice ». Il doit donc connaître l'**abstraction**
- Le programmeur de la structure de données doit choisir entre les différentes **implémentations possibles** de cette abstraction : tableau bidimensionnel, tableau de lignes ou liste de ses éléments non nuls (matrice creuse)...

Tableaux

- L'abstraction de données "Tableau"
- Implémentation usuelle
- Utilisation en Python natif
- Utilisation en C
- Utilisation avec la librairie NumPy
- Exercice

Abstraction "Tableau"

Un tableau permet de mémoriser une séquence d'éléments :

- Le **nombre d'éléments** du tableau est **fixe**.
- Tous les éléments du tableau sont de **même type**.
- Chaque élément est repéré par sa position dans le tableau, son indice.

Remarque : d'autres façons de représenter des séquences de données (listes chaînées, piles, files...) seront vues ultérieurement.

Tableaux - L'abstraction de données "Tableau"

Opération possible : création

La **création** d'un tableau précise :

- Le **nombre d'éléments** qu'il contient (sa dimension `DIM`).
- Le **type** des éléments du tableau.

Opération possible : accès à un élément

`tableau[i]` : élément d'indice `i` dans `tableau`

Remarque : La validité de l'indice `i` dépend du langage :

- Python, C, Java, ... : `0<=i<DIM`
- Autre convention : `1<=i<=DIM`

Représentation en mémoire

Dans une grande partie des langages de programmation, les tableaux sont représentés en mémoire sous la forme de **cellules contiguës** :

Adresse mémoire	Donnée	
0xFA70	01001000	← *Début du tableau (indice 0)*
0xFA71	11100111	*(indice 1)*
0xFA72	10000111	*(indice 2)*
0xFA73	00101111	*...*

- Adresse mémoire de l'élément `tableau[i]` : adresse de `tableau[0]` + i × taille d'une cellule.
- Le **temps d'accès** à l'élément d'indice `i` (lecture ou modification) est donc constant (indépendant de la valeur de `i` et de la dimension `DIM` du tableau.

Représentation en mémoire

Les opérations d'insertion ou de suppression d'éléments dans cette implémentation impliqueraient alors de déplacer d'autres éléments :

Adresse mémoire	Donnée	
0xFA70	01001000	← *Début du tableau*
~~0xFA71~~	~~11100111~~	← *Élément à supprimer*
0xFA72	10000111	↑ *Éléments à déplacer*
0xFA73	00101111	↑ ...

- La suppression et l'insertion ne sont donc pas autorisées pour le type abstrait "Tableau".

Utilisation en Python natif

Utilisation du type `list` natif de Python, en s'imposant les règles suivantes :

- La **taille** et le **type** des éléments sont fixés à la première affectation.
- Toutes les opérations modifiant la taille de l'objet de type `list` (append, insert, `del`...) sont interdites.
- On ne mélange pas les types à l'intérieur de l'objet de type `list`.
- Les seules opérations autorisées sont la lecture et l'écriture à un indice donné.

Tableaux - Utilisation en Python natif

Exemple :

```
1 # Creation d'un tableau d'entiers de taille 20 :
2 tableau = [0]*20
3 # Ecriture dans le tableau :
4 tableau[0] = int(input())
5 # Lecture et ecriture
6 tableau[1] = 2*tableau[0]
```

Utilisation en C

Un programme respectant l'abstraction tableau se traduira naturellement en langage C. Ainsi l'exemple précédent donnera :

```c
int main () {
  // Tableau d'entiers de taille 20 :
  int tableau[20] = {0}; // Valable uniquement
   pour 0
  // Ecriture dans le tableau :
  scanf("%d",&tableau[0]);
  // Lecture et ecriture
  tableau[1] = 2*tableau[0];
  return(0);
}
```

La librairie NumPy

- Important package Python pour le calcul scientifique
- Utilisation :

```
1 import numpy as np
```

Avantages :

- Gestion efficace de tableaux multidimensionnels.
- Contrôle des indices et des types.

Création d'un tableau

La création d'un tableau se fait en précisant :

- `shape` : Sa **longueur**.
- `dtype` (optionnel) : Le **type** de ses éléments (`float64` généralement si le paramètre n'est pas précisé)

Syntaxe

```
1 # Valeurs toutes initialisees a 0
2 np.zeros(shape,dtype)
3 # Valeurs toutes initialisees a 1
4 np.ones(shape,dtype)
5 # Valeurs non initialisees
6 np.empty(shape,dtype)
```

Tableaux - Utilisation avec la librairie NumPy

Exemple

```
1 import numpy as np
2
3 # Creation d'un tableau d'entiers de taille 20 :
4 tableau = np.zeros(20,int)
5 # Ecriture dans le tableau :
6 tableau[0] = int(input())
7 # Lecture et ecriture
8 tableau[1] = 2*tableau[0]
```

Propriétés du tableau

On peut accéder à différentes propriétés du tableau :

```
1 import numpy as np
2
3 tableau = np.zeros(17,int)
4 print("Nombre de dimensions :",tableau.ndim)
5 print("Valeurs des dimensions :",tableau.shape)
6 print("Nombre d elements :",tableau.size)
7 print("Type des elements :",tableau.dtype)
```

```
1 Nombre de dimensions : 1
2 Valeurs des dimensions : (17,)
3 Nombre d elements : 17
4 Type des elements : int64
```

Exercice 1

On considère la fonction suivante :

```
1 def decaler(tableau,i_source,i_destination):
2     assert 0<=i_source<len(tableau) and
       0<=i_destination<len(tableau), "Indices␣
       incorrects"
3     valeur = tableau[i_source]
4     del tableau[i_source]
5     tableau.insert(i_destination,valeur)
```

- Décrire l'opération réalisée par cette fonction.
- Pour quelles raisons cette fonction ne respecte pas l'abstraction tableau ?
- Corriger ce code afin qu'elle respecte l'abstraction.

Indication : on s'autorise le pythonesque `a,b=b,a` *afin de permuter deux éléments du tableau.*

Corrigé

Interprétation dans le cas : `i_source=2` et `i_destination=4`

i	0	1	2	3	4	5
liste[i]	7	5	9	0	3	7

```
1 del tableau[i_source]
```

i	0	1	2	3	4
liste[i]	7	5	0	3	7

```
1 tableau.insert(i_destination,valeur)
```

i	0	1	2	3	4	5
liste[i]	7	5	0	3	9	7

- Changement de taille → non-respect de l'abstraction tableau.

Corrigé

```
1  def decaler(tableau,i_source,i_destination):
2      assert 0<=i_source<len(tableau) and
        0<=i_destination<len(tableau), "Indices incorrects"
3      valeur = tableau[i_source]
4      while i_source < i_destination:
5          tableau[i_source] = tableau[i_source + 1]
6          i_source += 1
7      while i_source > i_destination:
8          tableau[i_source] = tableau[i_source - 1]
9          i_source -= 1
10     tableau[i_destination] = valeur
```

Abstraction "Matrice"

Une matrice permet de mémoriser des éléments selon deux dimensions :

- Le **nombre de lignes** `NB_LIGNES` et de **colonnes** `NB_COLONNES` est `fixe`.
- Tous les éléments de la matrice sont de **même type**.
- Chaque élément est repéré par sa position dans la matrice, constituée par le couple (numéro de ligne, numéro de colonne).

Opération possible : création

La **création** d'une matrice précise :

- Son nombre de lignes `NB_LIGNES` et son nombre de colonnes `NB_COLONNES`.
- Le **type** des éléments de la matrice.

Matrice - L'abstraction de données "Matrice"

Opération possible : accès à un élément

matrice[i][j] : élément de la matrice situé à la ligne `i` et à la colonne `j`

Remarque 1 : La validité des indices `i` et `j` dépend du langage :

- Généralement : `0<=i<NB_LIGNES` et `0<=j<NB_COLONNES`
- Autre convention : `1<=i<=NB_LIGNES` et `1<=j<=NB_COLONNES`

Remarque 2 :

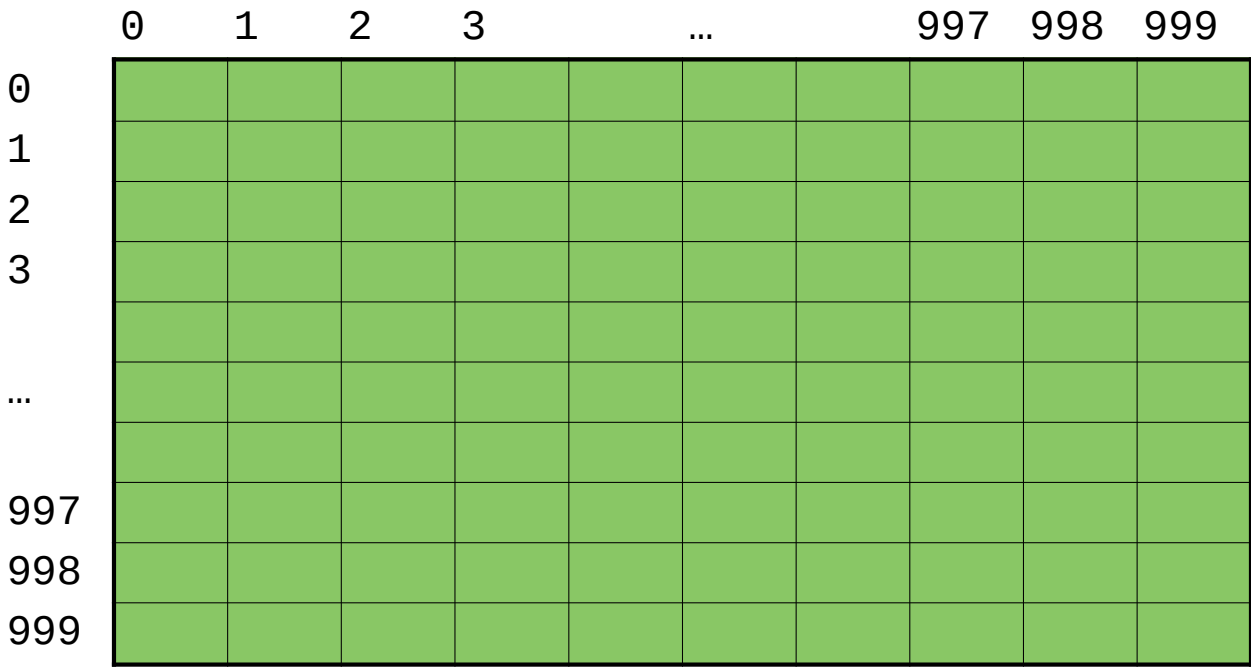
- `matrice[i]` est un tableau de dimension `NB_COLONNES`.
- Certains langages autorisent aussi l'écriture `matrice[i,j]`.

Matrice implémentée par un tableau

On affecte une **zone mémoire contiguë** suffisamment grande pour contenir `NB_LIGNES` $\times$ `NB_COLONNES` éléments.

Exemple

Matrice de 1000 $\times$ 1000 éléments :



Matrice implémentée par un tableau

Avantages :

- Structuration en ligne préservée.
- Opérations efficaces sur les lignes *(Exemple : affecter la valeur d'une ligne à une autre ligne ou transmettre une ligne en paramètre)*.
- Assure un **temps d'accès faible et constant** à un élément de la matrice.

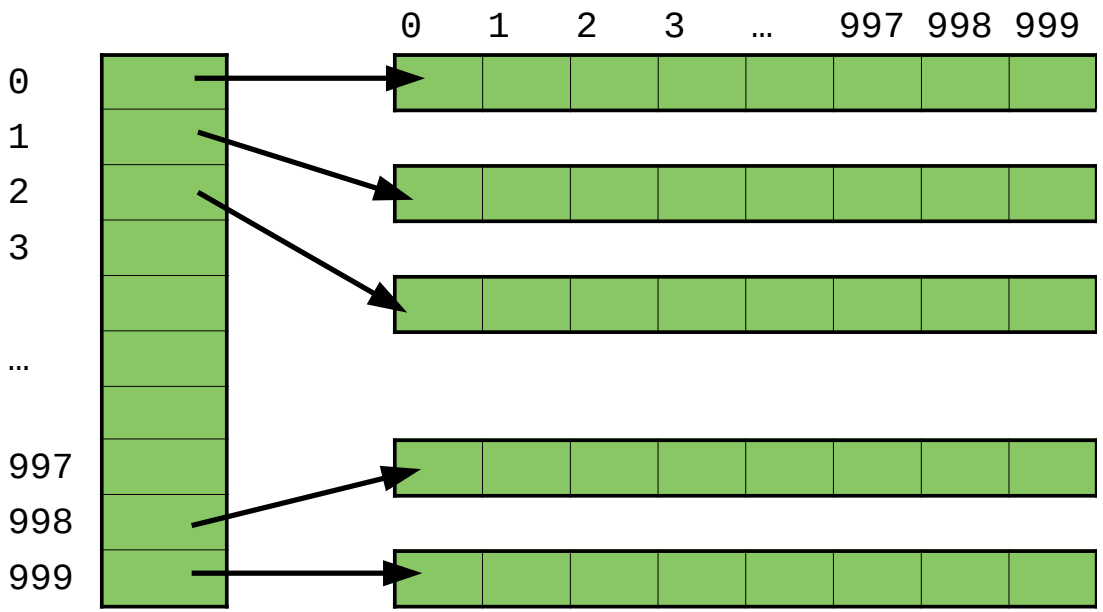
Inconvénients :

- Peut occuper une **place importante en mémoire**, alors même que peu d'éléments ont une valeur significative *(Exemple : matrice contenant essentiellement des 0 $\rightarrow$ Cf. matrice creuse)*

Matrice implémentée par un vecteur de vecteurs

On représente chaque ligne par un vecteur et la matrice est un **vecteur de vecteurs**.

- Similaire à celle que l'on retrouve en Python natif.
- Permet de manipuler globalement les lignes d'une matrice et optimiser certains traitements *(Exemple : échanger les valeurs de deux lignes, sans recopie des contenus de ces lignes)*.



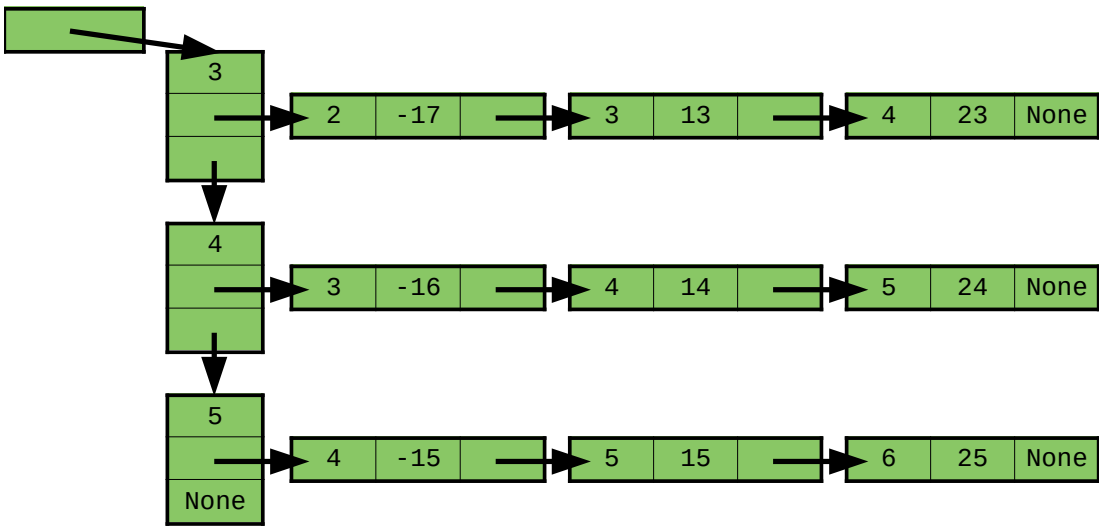
Matrice creuse

Matrice dont la grande majorité des éléments valent 0 $\rightarrow$ **seuls les éléments non nuls sont mémorisés**.

- Optimise la place occupée et/ou réduit le temps d'exécution de certaines opérations.

Inconvénients :

- Augmente le temps pour accéder à un élément (i,j) donné.
- Complexifie l'interface utilisateur



Utilisation en Python natif

Utilisation du type `list` natif de Python, sous forme d'une **liste de listes**. Comme pour l'abstraction tableau, les règles suivantes doivent être respectées :

- Le **nombre de lignes, de colonnes** et le **type** des éléments sont fixés à la première affectation.
- Toutes les opérations de modification de taille (`append`, `insert`, `del`...) sont interdites.
- Aucune modification de type n'est autorisée.
- Les seules opérations autorisées sont la lecture et l'écriture à une position donnée.

Matrice - Utilisation en Python natif

Exemple :

```
1 # Creation d'une matrice d'entiers de 19 lignes
    et 13 colonnes :
2 matrice = [None]*19
3 for i in range(len(matrice)):
4   matrice[i] = [0]*13
5 # Ecriture dans la matrice :
6 matrice[3][4] = int(input())
7 # Lecture et ecriture
8 matrice[2][7] = 2*matrice[3][4]
```

Utilisation en C

Un programme respectant l'abstraction matrice se traduira naturellement en langage C. Ainsi l'exemple précédent donnera :

```c
int main(void) {
  // Matrice d'entiers de 19 lignes et 13 colonnes :
  int matrice[19][13] = {0}; // valable uniquement pour 0
  // Ecriture dans le tableau :
  scanf("%d",&matrice[3][4]);
  // Lecture et ecriture
  matrice[2][7] = 2*matrice[3][4];
  return(0);
}
```

Création d'une matrice

La création d'une matrice se fait en précisant :

- `shape` : Un tuple `(NB_LIGNES, NB_COLONNES)`
- `dtype` (optionnel) : Le **type** de ses éléments (`float64` généralement si le paramètre n'est pas précisé)

Syntaxe

```
1 # Valeurs toutes initialisees a 0
2 np.zeros(shape,dtype)
3 # Valeurs toutes initialisees a 1
4 np.ones(shape,dtype)
5 # Valeurs non initialisees
6 np.empty(shape,dtype)
```

Matrice - Utilisation avec la librairie NumPy

Exemple

```
1 import numpy as np
2
3 # Creation d'une matrice d'entiers de 19 lignes
    et 13 colonnes :
4 matrice = np.zeros((19,13),int)
5 # Ecriture dans la matrice :
6 matrice[3][4] = int(input())
7 # Lecture et ecriture
8 matrice[2][7] = 2*matrice[3][4]
```

Propriétés du tableau

On peut accéder à différentes propriétés du tableau :

```
1 import numpy as np
2
3 matrice = np.zeros((17,11),int)
4 print("Nombre de dimensions :",matrice.ndim)
5 print("Valeurs des dimensions :",matrice.shape)
6 print("Nombre d elements :",matrice.size)
7 print("Type des elements :",matrice.dtype)
```

```
1 Nombre de dimensions : 2
2 Valeurs des dimensions : (17, 11)
3 Nombre d elements : 187
4 Type des elements : int64
```

Matrice - Exercice

Énoncé

La **carte d'un jeu** de plateau comporte des cases qui rapportent des points (un nombre entier), selon une représentation en Python natif sous forme d'une matrice :

```
          0  1  2  3  4
plan =   [
      0  [1,2,0,5,3],
      1  [0,1,3,1,2],
      2  [3,1,2,0,0],
      3  [0,3,4,1,0],
      4  [0,0,1,2,3]
                    ]
```

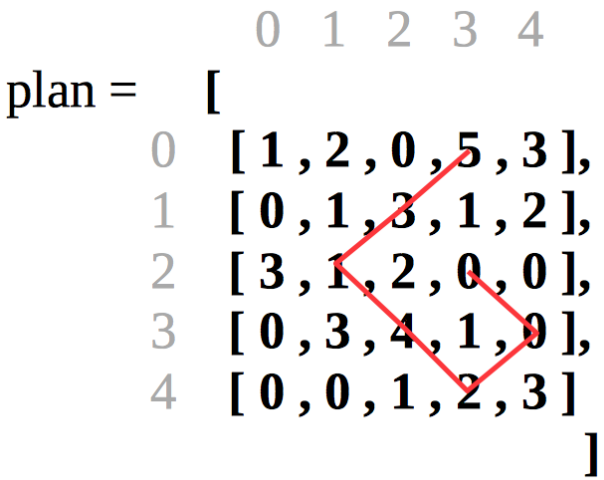
Dans la carte ci-dessus, la case `(0,3)` a pour valeur 5, et la case `(3,2)` vaut 4.

Un **point de départ** est identifié par ses coordonnées et un **chemin** par une liste de directions parmi `"NE"`, `"SE"`, `"SO"`, `"NO"`.

Matrice - Exercice

Exemple

- point de départ : (2,3)
- chemin : ["SE","SO","NO","NO","NE","NE"],



Énoncé (suite)

Écrire la fonction `maxmimum_sur_chemin` qui accepte les paramètres suivants :

- `plan` : une carte (matrice en Python natif).
- `depart` : le couple `(x,y)` des coordonnées du point de départ.
- `chemin` : liste de chaînes de caractères.

Cette fonction retourne la **plus grande des valeurs rencontrées** (les points de départ et d'arrivée sont inclus).

Indications :

- On suppose que le point de départ est bien sur la carte.
- Si jamais le chemin sort de la carte, la valeur retournée est la valeur la plus grande rencontrée jusque là.

Corrigé

```
1  def point_suivant(x,y,direction):
2    if direction=='NE':
3      x,y = x-1,y+1
4    elif direction=='NO':
5      x,y = x-1,y-1
6    elif direction=='SE':
7      x,y = x+1,y+1
8    elif direction=='SO':
9      x,y = x+1,y-1
10   return x,y
```

Matrice - Exercice

```python
def maximum_sur_chemin(plan,depart,chemin):
    # dimensions de la matrice
    nb_lignes = len(plan)
    nb_colonnes = len(plan[0])
    # initialisations
    x,y = depart
    est_interieur = True
    i_direction = 0
    maximum = plan[x][y]
    while i_direction<len(chemin) and est_interieur:
        # nouvelle position
        direction = chemin[i_direction]
        x,y = point_suivant(x,y,direction)
        est_interieur = 0<=x<nb_lignes and 0<=y<nb_colonnes
        # actualisation maximum
        if est_interieur and plan[x][y]>maximum:
            maximum = plan[x][y]
        i_direction += 1
    return maximum
```