Tableaux
Utilité des tableaux

Créons un algorithme qui permet de **stocker les notes de 12 étudiants** et de **calculer la moyenne de ces notes**.

Algorithme Moyenne_Notes

Var : N1, N2, N3, N4, N5, N6, N7, N8, N9, N10, N11, N12, Moy : réel

Début
  Lire ("Entrez la note de l'étudiant 1" ; N1)
  Lire ("Entrez la note de l'étudiant 2" ; N2)
  .
  .
  Lire ("Entrez la note de l'étudiant 12" ; N12)
  Moy $\leftarrow$ (N1+N2+N3+N4+N5+N6+N7+N8+N9+N10+N11+N12)/12
  Ecrire ("La moyenne des notes est " ; Moy)
Fin

Qu'en est-il lorsqu'on a à gérer quelques centaines ou quelques milliers de notes d'étudiants ?

Tableaux
Utilité des tableaux

- L'algorithmique nous permet de **rassembler** toutes **ces variables en une seule**, au sein de laquelle chaque **valeur** sera désignée par un **numéro**.
- "**la note numéro 1**", "**la note numéro 2**", "**la note numéro 8**", etc.
- On peut aussi les exprimer : "**Note(1)**", "**Note(2)**", "**Note(8)**", etc.

Définitions

- Un **ensemble de valeurs** portant le **même nom de variable** et **repérées par un nombre**, s'appelle un **tableau**, ou encore une **variable indicée**.
- Le **nombre** qui, au sein d'un tableau, sert à **repérer chaque valeur** s'appelle l'**indice**.
- Chaque fois que l'on doit **désigner un élément du tableau**, on fait figurer le **nom du tableau**, suivi de l'**indice de l'élément**, entre **parenthèses**.

Tableaux
Notation et utilisation algorithmique

- **Dans notre exemple**, on peut donc faire appel à **un tableau** appelé **Note**.
- Chaque **note individuelle** (chaque élément du tableau Note) sera donc désignée **Note(0)**, **Note(1)**, etc.
- En général, les **indices des tableaux** commencent à 0 et non pas à 1.
- La $i^{ème}$ **élément** du tableau **Note** dans le tableau correspond à **Note(i - 1)**.

★ Le quatrième étudiant à eu une note égale à 17,5.

- L'opération d'affectation : **Note(3)** $\leftarrow$ **17,5**

0	1	2	3	4	5	6	7	8	9	10	11
			17,5								

Tableaux
Notation et utilisation algorithmique

Déclaration d'un tableau

Un tableau est déclaré en précisant le **nombre** et le **type** de **valeurs** qu'il contiendra.

Conventions

- les "cases" d'un tableau sont numérotées à partir de zéro, autrement dit, **le plus petit indice est zéro**.
- lors de la déclaration d'un tableau, on précise la **plus grande valeur** de l'indice.

Var : Note(11) : réel

Tableaux
Notation et utilisation algorithmique

- L'algorithme de calcul de la moyenne des 12 notes devient :

Algorithme Moyenne_Notes

Var : Note(11), Moy : réel ; i : entier

Début
  Pour i de 0 à 11 **Faire**
    Ecrire ("Entrez la note de l'étudiant " ; i + 1)
    Lire (Note(i))
  Fin Pour
  Moy $\leftarrow$ 0
  Pour i de 0 à 11 **Faire**
    Moy $\leftarrow$ Moy + Note(i)
  Fin Pour
  Moy $\leftarrow$ Moy /12
  Ecrire ("La moyenne des notes est " ; Moy)
Fin

Tableaux
Notation et utilisation algorithmique

Important

L'indice qui sert à désigner les éléments d'un tableau peut être exprimé directement comme :

- un **nombre en clair** ▶ **Note(4)** ;
- une **variable** ▶ **Note(i)** ;
- ou une **expression calculée** ▶ **Note((i–1)/2)**.

Dans un tableau, la valeur d'un indice doit toujours :

- être **égale au moins à 0** ;
- être **un nombre entier**. L'élément **Note(3,5)** n'existe jamais ;
- **être inférieure ou égale** au nombre d'éléments du tableau. Si le tableau **Note** a été déclaré comme ayant 12 éléments, l'utilisation de **Note(32)** déclenchera automatiquement une erreur.

Tableaux
Notation et utilisation algorithmique

Attention

Ne pas confondre l'indice d'un élément d'un tableau avec le contenu de cet élément. La troisième maison de la rue n'a pas forcément trois habitants, et la vingtième vingt habitants. En notation algorithmique, il n'y a aucun rapport entre **i** et **Maison(i)**.

Tableaux
Exemples

Écrire un algorithme qui déclare et remplisse un tableau de 7 valeurs réelles en les mettant toutes à zéro.

Algorithme Remplissage_Tableau

Var : Truc(6) : réel ; i : entier

Début
  Pour i de 0 à 6 **Faire**
    Truc(i) $\leftarrow$ 0
  Fin Pour
Fin

Truc()

0	1	2	3	4	5	6
0	0	0	0	0	0	0

Tableaux
Exemples

Écrire un algorithme qui déclare et remplisse un tableau contenant les six voyelles de l'alphabet latin.

Algorithme Voyelles_Alphabet

Var : Voyelle(5) : chaîne

Début

Voyelle(0) ← "a"
Voyelle(1) ← "e"
Voyelle(2) ← "i"
Voyelle(3) ← "o"
Voyelle(4) ← "u"
Voyelle(5) ← "y"

Fin

Voyelle()

0	1	2	3	4	5
"a"	"e"	"i"	"o"	"u"	"y"

Collections

Définition : une **collection** est une structures de données permettant de stocker des ensembles.

Trois types en Python :

- Les listes
- Les tuples
- Les dictionnaires

Les opérations que l'on peut effectuer sur chacune sont spécifiques au type de collection.

Particularité : les éléments contenus dans les collections ne sont pas forcément tous du même type.

Les listes

Ensemble d'éléments

- non triés
- modifiables

Notation :

- La liste est donnée entre crochets
- Les éléments sont séparés par des virgules

```
1    >>> l = []
2    >>> type( l )
3    <type 'list'>
4    >>> m = [ 5, 7.6, "bonjour"]
```

Une liste est *ordonnée* : l'ordre dans lequel ses éléments sont disposés a de l'importance.

- Pour accéder à un élément de la liste : utilisation de l'opérateur crochets
- La numérotation des indices commence à 0.

```
1    >>> print m[0]
2    5
```

Opérations sur les listes

Nombre d'éléments dans une liste : fonction `len()`

```
1  >>> len( m )
2  3
```

Ajouter un élément dans une liste :

- À la fin de la liste : fonction `append()`

```
1  >>> jour = ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
2  >>> jour.append( 'dimanche' )
3  >>> print jour
4  ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'dimanche']
```

- À un endroit précis de la liste : fonction `insert()`

```
1  >>> jour.insert( 5, 'samedi')
2  >>> print jour
3  ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi',
4  'dimanche']
```

Compter le nombre de fois où un élément apparait dans le liste : fonction `count()`

```
1  >>> lst5 = [ 3, 5, 2, 4, 4, 3]
2  >>> lst5.count( 4 )
3  2
```

Opérations sur les listes

Retirer un élément d'une liste : fonction `remove()`

```
1 >>> jour.remove( 'samedi' )
2 >>> print jour
3 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'dimanche']
```

Si l'élément se trouve plusieurs fois dans la liste, seul le premier est retiré.

Retirer un élément d'une liste en le retournant : fonction `pop()`

- Si un indice est passé en paramètre : on retire l'élément correspondant à cet indice
- Si pas de paramètre passé : on retire le dernier élément de la liste

```
1 >>> jour.pop()
2 'dimanche'
3 >>> print jour
4 ['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
5 >>> jour.pop(2)
6 'mercredi'
7 >>> print jour
8 ['lundi', 'mardi', 'jeudi', 'vendredi']
```

Opérations sur les listes

Concaténation de listes : opérateur `+`

```
1 >>> lst1 = [1, 2, 3]
2 >>> lst2 = [9, 8, 7]
3 >>> lst3 = lst1 + lst2
4 >>> print lst3
5 [1, 2, 3, 9, 8, 7]
```

Tri des éléments d'une liste : fonction `sort()`

```
1 >>> lst2.sort()
2 >>> print lst2
3 [7, 8, 9]
```

Assemblage des éléments d'une liste sous forme d'une chaîne de caractères : fonction `join()` (avec une autre chaîne de caractères)

```
1 >>> lst4 = [ 'i', 'u', 'tv' ]
2 >>> "".join(lst4)
3 'iutv'
```

Les tuples

Tuple : structure de donnée proche d'une liste, mais **non modifiable**

- La liste est donnée entre parenthèses
- Les éléments sont séparés par des virgules

Opérations sur les tuples : les mêmes que pour les listes, en-dehors des opérations de modifications.

```
1 >>> jour = ('lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi')
2 >>> jour.append( 'dimanche' )
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 AttributeError: 'tuple' object has no attribute 'append'
```

Pourquoi utiliser des tuples :

- Implémentés de façon plus simple que les listes : ils utilisent moins de ressources (en particulier en mémoire).
- Pour s'assurer qu'un ensemble ne sera pas modifié par une autre partie du programme

Pourquoi ne pas utiliser des tuples :

- Non modifiables : plus contraignants

Les dictionnaires

Dictionnaire : structure de donnée qui effectue une association entre une **clé** et une **valeur**.

Notation :

- Définition en utilisant des accolades.
- Couples clé-valeur séparés par des virgules
- Une clé est séparée de sa valeur par deurs points :

```
1 >>> dic = { 'pomme' : 'fruit' , 'poire' : 'fruit', 'poireau' :
2 'legume' }
```

Opérations sur les dictionnaires

Accéder aux valeurs : on donne la clé entre crochets

```
1 >>> dic['poireau']
2 'legume'
```

Ajouter une nouvelle paire : on affecte la valeur à sa clé passée entre crochets.

```
1 >>> dic['tomate'] = 'fruit'
2 >>> print dic
3 {'poire': 'fruit', 'tomate': 'fruit', 'poireau': 'legume', 'pomme': 'fruit'}
```

Suppression d'un couple :

- Suppression en retournant la valeur : fonction `pop()` en passant la clé de l'élément à supprimer

```
1 >>> dic.pop( 'poireau' )
2 'legume'
3 >>> print dic
4 {'poire': 'fruit', 'tomate': 'fruit', 'pomme': 'fruit'}
```

- Suppression simple : opérateur `del` sur un élément désigné par sa clé entre crochets

```
1 >>> del dic['pomme']
2 >>> print dic
3 {'poire': 'fruit', 'tomate': 'fruit'}
```

Opérations sur les dictionnaires

Liste des clés et des valeurs : fonctions `keys()` et `values()`.

```
1 >>> dic.keys()
2 ['poire', 'tomate', 'poireau', 'pomme']
3 >>> dic.values()
4 ['fruit', 'fruit', 'legume', 'fruit']
```

Remarque : ce sont des listes.

Existence d'une clé dans le dictionnaire : fonction `has_key()`.

```
1 >>> dic.has_key( 'tomate' )
2 True
```

Éléments d'une collection

On peut effectuer une boucle `for` en itérant sur les éléments contenus dans une collection : utilisation du mot-clé `in`

```
1 lst = [1, 4, 6]
2 for i in lst:
3     print i
```

La variable d'itération contient l'élément de la liste sur lequel on se trouve.

On peut tester l'`appartenance` à une collection :

```
1 lst = [1, 4, 6]
2 if 4 in lst:
3     print 4 est present
```

Éléments d'une collection

Sur un dictionnaire :

- Itération sur les clés des couples avec `in` :

```
1 >>> for d in dic:
2 ...     print d
3 ...
4 poire
5 tomate
6 poireau
7 pomme
```

  La variable d'itération contient la clé du couple sur lequel on se trouve.

- Itération sur les couples avec la fonction `iteritems()`

```
1 >>> for k, v in dic.iteritems():
2 ...     print k, v
3 ...
4 poire fruit
5 tomate fruit
6 poireau legume
```

  On récupère la clé et la valeur.

Utilisation de collections pour représenter des tableaux

On représente généralement les tableaux par des listes

- On accède aux éléments avec l'indice correspondant entre crochets

```
1 >>> tab = [ 1, 3, 6, 2, 4 ]
2 >>> print tab[2]
3 6
```

Pour aller plus loin : modules de calcul scientifique numpy et scipy

- Numpy définit un type `array` et un type `matrix`
- Fournissent des opérations sur ces matrices

Collections imbriquées

On peut **imbriquer** des collections : mettre des collections dans les champs d'une collection

- C'est notamment une façon de représenter un tableau à plusieurs dimensions

```
1  >>> lst = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]
2  >>> print lst
3  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
4  >>> print lst[2]
5  [7, 8, 9]
6  >>> print lst[2][1]
7  8
```

Précaution importante

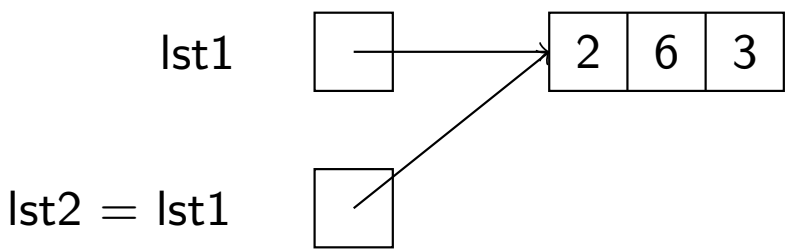
Attention : Python fonctionne en désignant les variables par leur **référence**. Une référence désigne l'adresse de début de l'espace mémoire occupé par la variable.

Exemple :

- Déclaration d'une liste `lst1`
- `lst1` contient en réalité la référence de cette liste
- On copie `lst1` dans une autre variable
- C'est en réalité la référence de la liste qui sera copiée :

```
1 >>> lst1 = [ 2, 6, 3 ]
2 >>> lst2 = lst1
3 >>> print lst2
4 [2, 6, 3]
```

- On parle de **shallow copy**



Précaution importante

Si on effectue une modification sur la liste pointée par `lst2` :

- En mémoire, `lst1` et `lst2` désignent la **même entité**
- La modification est donc effectuée sur la liste pointée par `lst1`

```
1 >>> lst2.append( 5 )
2 >>> print lst1
3 [2, 6, 3, 5]
```

Pour faire une vraie copie : copie explicite de l'élément

- On crée un nouvel élément et on recopie dedans le contenu de l'élément à copier
- On parle de **deep copy**

```
1 >>> lst3 = list( lst1 )
2 >>> print lst3
3 [2, 6, 3, 5]
4 >>> lst3.append( 1 )
5 >>> print lst3, lst1
6 [2, 6, 3, 5, 1] [2, 6, 3,
      5]
```