CHAPITRE 4: SÉQUENCES, LISTES, ENSEMBLES, TABLES ET TABLEAUX

LES SÉQUENCES:

Une ***séquence*** , de type *exprseq* , est une suite d'éléments séparés par une virgule . On accède à chaque élément par son rang , mais l'affectation est interdite :

```
> restart:
  s:=a,b,-1,2*p-q,r;
```

$$s := a, b, -1, 2\,p - q, r$$

```
> s[4];s[2..5];s[3]:=0;
```

$$2\,p - q$$

$$b, -1, 2\,p - q, r$$

```
Error, cannot assign to an expression sequence
```

NULL désigne la séquence vide:

```
> v:=NULL;
```

$$v :=$$

Pour répéter un élément , utiliser l'opérateur dollar **$** :

```
> t:=a$2,y$5;
```

$$t := a, a, y, y, y, y, y$$

Pour concaténer 2 séquences ,utiliser l'opérateur virgule **,**

```
> u:=s,t;
```

$$u := a, b, -1, 2\,p - q, r, a, a, y, y, y, y, y$$

op(expr) retourne la séquence formée des opérandes de l'expression *expr* :

```
> op(a-y*(a+y+z)+3*z);
```

$$a, -y\,(a + y + z), 3\,z$$

La fonction ***seq*** permet de créer une séquence
pour un indice *i* variant de *a* à *b* avec un pas de 1 : syntaxe ***seq(expr,i=a..b)*** , ou
pour un indice décrivant les opérandes d'une expression *e* : syntaxe ***seq(expr,i=e)*** :

```
> seq(i^3,i=-1/2..2);
```

$$\frac{-1}{8}, \frac{1}{8}, \frac{27}{8}$$

```
> e:=a-y*(a+y+z)+3*z:op(e);
  seq(mu+i,i=e);
```

$$a, -y\,(a + y + z), 3\,z$$

$$\mu + a, \mu - y\,(a + y + z), \mu + 3\,z$$

```
> seq(seq(a||i||j,j=1..3),i=1..3);
```

$$a11, a12, a13, a21, a22, a23, a31, a32, a33$$

LES LISTES:

Une ***liste*** , de type *list* , est obtenue en plaçant une séquence entre les crochets [et] .
La liste vide se note [] .

```
> s:=a$2,b$3,alpha,beta;L:=[s];
```

$$s := a, a, b, b, b, \alpha, \beta$$

$$L := [a, a, b, b, b, \alpha, \beta]$$

On accède à chaque élément d'une liste par son rang , l'affectation est autorisée:

```
> L[6];L[2..5];L[3]:=delta;
```

$$\alpha$$

$$[a, b, b, b]$$

$$L_3 := \delta$$

op transforme une liste en séquence , ***nops*** donne le nombre d'éléments d'une liste .
Les fonctions ***op*** et ***nops*** ne peuvent agir sur une séquence *s*: faire ***ops([s])*** et ***nops([s])***

```
> op(L);
```

$$a, a, \delta, b, b, \alpha, \beta$$

```
> nops(L);
```

$$7$$

Pour modifier une liste, on peut aussi utiliser ***subs*** ou ***subsop*** :

```
> L:=subs(b=b1,L);
```

$$L := [a, a, \delta, b1, b1, \alpha, \beta]$$

```
> L:=subsop(3=gamma,L);
```

$$L := [a, a, \gamma, b1, b1, \alpha, \beta]$$

La fonction booléenne ***member*** permet de savoir si une expression est membre d'une liste:

```
> member(gamma,L), member(epsilon,L);
```

$$true, false$$

LES ENSEMBLES :

Un ***ensemble***, de type *set* , est obtenu en plaçant une séquence entre les accolades { et }.
Les éléments sont rangés par adresse , donc l'ordre initial n'est pas nécessairement conservé et les éléments en
double sont supprimés. L'ensemble vide se note {} .

```
> s:=a$2,b$3,alpha,beta;e:={s};
```

$$s := a, a, b, b, b, \alpha, \beta$$

$$e := \{a, b, \alpha, \beta\}$$

On accède à chaque élément d'un ensemble par son rang , mais l'affectation est interdite :

```
> e[2];e[2..3];e[3]:=delta;
```

$$b$$

$$\{b, \alpha\}$$

```
Error, cannot assign to a set
```

op transforme un ensemble en séquence , ***nops*** donne le nombre d'éléments d'un ensemble .

```
> op(e);nops(e);
```

$$a, b, \alpha, \beta$$

$$4$$

Pour modifier un ensemble , l'affectation étant interdite , utiliser ***sub***s ou ***subsop*** :

```
> e:=subs(b=b1,e);
```

$$e := \{a, \alpha, \beta, b1\}$$

```
> e:=subsop(3=delta,e);
```

$$e := \{a, \alpha, \delta, b1\}$$

La fonction booléenne ***member*** permet de savoir si une expression est membre d'un ensemble:

```
> member(delta,e), member(epsilon,e);
```

$$true, false$$

Convertir un ensemble en somme, en produit, en liste, ou convertir une liste en ensemble avec ***convert*** :

```
> convert(e,`+`);convert(e,`*`);
  e:=convert(e,list);e:=convert(e,set);
```

$$a + \alpha + \delta + b1$$

$$a\ \alpha\ \delta\ b1$$

$$e := [a, \alpha, \delta, b1]$$

$$e := \{a, \alpha, \delta, b1\}$$

Opérations sur les ensembles:
union (union) , ***intersect*** (intersection) , ***minus*** (différence) .
la fonction booléenne *a* ***subset*** *b* indique si *a* est un sous-ensemble de *b*.

```
> e:={alpha,beta,gamma,delta}:f:={beta,delta,epsilon,phi,lambda}:
  u:=e union f;i:=e intersect f;m:=f minus e;
```

$$u := \{\alpha, \beta, \delta, \varepsilon, \lambda, \gamma, \phi\}$$

$$i := \{\beta, \delta\}$$

$$m := \{\varepsilon, \lambda, \phi\}$$

```
> i subset e, i subset m;
```

$$true, false$$

QUELQUES OUTILS DU PACKAGE COMBINAT:

```
> with(combinat,choose):
```
(appel de l'outil ***choose***, qui génère les sous-listes ordonnées d'une liste donnée)
```
> choose([a,b,c]);
```
$$[[\ ], [a], [b], [a, b], [c], [a, c], [b, c], [a, b, c]]$$
Génération de toutes les sous-listes de longueur 2 ordonnées d'une liste de longueur 3:
```
> choose([a,b,c],2);
```
$$[[a, b], [a, c], [b, c]]$$
choose(*n*,*p*) avec *n*,*p* entiers , génère toutes les sous-listes de longueur *p* ordonnées de [1,2,..,*n*]
```
> choose(4,3);
```
$$[[1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]]$$

```
> with(combinat,permute):
```
(appel de l'outil ***permute***, qui génère les listes de permutation d'une liste donnée)
```
> permute([a,b,c]);
```
$$[[a, b, c], [a, c, b], [b, a, c], [b, c, a], [c, a, b], [c, b, a]]$$
Génération de toutes les sous-listes de permutation de longueur 2 de [a,b,c] :
```
> permute([a,b,c],2);
```
$$[[a, b], [a, c], [b, a], [b, c], [c, a], [c, b]]$$
permute(*n*) , avec *n* entier , donne les listes de permutation de [1,2,...,*n*]
```
> permute(3);
```
$$[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]$$
permute(*n*,*p*) , avec *n*,*p* entiers , donne les sous-listes de permutation de de longueur *p* de [1,2,...,*n*]
```
> permute(3,2);
```
$$[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]$$
```
> restart;
```

LES TABLES:

Une ***table*** est une structure indexée dont les *indices* (ou *index*) peuvent être de n'importe quel type.
La fonction ***table*** permet de créer une table :

Création d'une table vide:
```
> v:=table();
```
$$v := \text{table}([])$$
Création d'une table T avec une liste de valeurs:
```
> T:=table([valeur1,valeur2]);
```
$$T := \text{table}([1 = valeur1, 2 = valeur2])$$
Création d'une table U avec liste d'index (ici ° . *) et de valeurs:
```
> U:=table([`°`=valeur1,`.`=valeur2,`*`=valeur3]);
```
$$U := \text{table}([. = valeur2, ° = valeur1, * = valeur3])$$
Pour afficher le contenu d'une table , utiliser ***print*** ou ***eval***
```
> print(T),eval(U);
```
$$\text{table}([1 = valeur1, 2 = valeur2])$$
$$\text{table}([. = valeur2, ° = valeur1, * = valeur3])$$

On peut modifier , créer un nouvel élément , ou supprimer un élément par affectation :

```
> T[a]:=valeur3:U[~]:=valeur4:print(T),eval(U);
```

table([1 = *valeur1*, 2 = *valeur2*, *a* = *valeur3*])

table([. = *valeur2*, ° = *valeur1*, ~ = *valeur4*, * = *valeur3*])

Une table peut être directement créée par assignation :

```
> V[a]:=1:V[b]:=2:V[c]:=3:eval(V);
```

table([*c* = 3, *b* = 2, *a* = 1])

Fonctions agissant sur des tables:

```
> op(op(V));
```

[*c* = 3, *b* = 2, *a* = 1]

```
> indices(V);
```

[*c*], [*b*], [*a*]

```
> entries(V);
```

[3], [2], [1]

La fonction ***map*** permet d'appliquer une fonction sur les valeurs d'une table:

```
> V:=map(sqrt,V);
```

$V := \text{table}([c = \sqrt{3}, b = \sqrt{2}, a = 1])$

La fonction ***copy*** permet d'effectuer une copie d'une table:

```
> W:=copy(V);
```

$W := \text{table}([c = \sqrt{3}, b = \sqrt{2}, a = 1])$

LES TABLEAUX:

Un ***tableau*** est une structure indexée, de type *array*, de plusieurs dimensions, dont les indices sont des entiers appartenant à un intervalle *a..b* (de type *range*) .

La fonction ***array*** permet de créer un tableau :
Des fonctions analogues à celles utilisées pour les tables existent :

Création d'un tableau à une dimension:

```
> T:=array(1..3);
```

T := array(1 .. 3, [ ])

Création d'un tableau à une dimension avec initialisation partielle:

```
> U:=array(1..5,[a,b,c]);
```

$U := [a, b, c, U_4, U_5]$

Création d'un tableau à 2 dimensions avec initialisation complète:

```
> V:=array([[a,b,c],[d,e,f]]);
```

$$V := \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

Création d'un tableau rectangulaire d'entiers par affectation à l'aide de boucles imbriquées:

```
> T:=array(1..2,1..3):
```

```
  for i to 2 do
    for j to 3 do
       T[i,j]:=(i-1)*3+j;
    end do;
  end do:
  print(T);
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
> op(op(T));
```

$$1 .. 2, 1 .. 3, [(2, 2) = 5, (1, 2) = 2, (1, 1) = 1, (2, 1) = 4, (1, 3) = 3, (2, 3) = 6]$$

```
> indices(T);
```

$$[2, 2], [1, 2], [1, 1], [2, 1], [1, 3], [2, 3]$$

```
> entries(T);
```

$$[5], [2], [1], [4], [3], [6]$$

```
> map(x->x^2,T);
```

$$\begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \end{bmatrix}$$

Conversion d'un tableau en liste (si la dimension est 1) ou en liste de listes (si la dimension est >1):

```
> convert(U,list);convert(T,listlist);
```

$$[a, b, c, U_4, U_5]$$

$$[[1, 2, 3], [4, 5, 6]]$$

Conversion d'un tableau de dimension quelconque en ensemble:

```
> convert(T,set);
```

$$\{1, 2, 3, 4, 5, 6\}$$

```
> T:=convert([a,b,c,d],array);
```

$$T := [a, b, c, d]$$

```
> U:=copy(T);
```

$$U := [a, b, c, d]$$

Exercices corrigés 4:

Ex 4.1: Ecrire une procédure qui inverse les éléments d'une liste .

```
> restart;
  listeinverse:= proc(L::list)
    local i;
    [seq(op(nops(L)-i,L),i=0..nops(L)-1)];
   end proc:
> listeinverse([a,b,c,d,e,f]);
```

$$[f, e, d, c, b, a]$$

Ex 4.2: Créer une matrice de Vandermonde d'ordre n . Exemple , pour $n = 4$:

```
> VDM:=proc(n::posint)
    local i,j,a,V;
    a:=array(1..n):V:=array(1..n,1..n):
    for i to n do for j to n do V[i,j]:=a[i]^(j-1) end do end do;
    print(V);
  end proc:
```

```
> VDM(4);
```

$$\begin{bmatrix} 1 & a_1 & a_1^2 & a_1^3 \\ 1 & a_2 & a_2^2 & a_2^3 \\ 1 & a_3 & a_3^2 & a_3^3 \\ 1 & a_4 & a_4^2 & a_4^3 \end{bmatrix}$$

Travail dirigé 4:

TD 4.1:

1° Un tableau *T* à *n* lignes et *p* colonnes étant donné , écrire une procédure ***ligne(T , p , i)*** qui calcule la somme des éléments de *T* situés en ligne *i* .

2° Un tableau *T* à *n* lignes et *p* colonnes étant donné , écrire une procédure ***colonne(T , n , j)*** qui calcule la somme des éléments de *T* situés en colonne *j* .

3° Un tableau carré *T* à *n* lignes et *n* colonnes étant donné , écrire une procédure ***diagonale1(T , n)*** qui calcule la somme des éléments de *T* situés sur la première diagonale.

4° Un tableau carré *T* à *n* lignes et *n* colonnes étant donné , écrire une procédure ***diagonale2(T , n)*** qui calcule la somme des éléments de *T* situés sur la seconde diagonale.

5° Un tableau *T* carré à *n* lignes et *n* colonnes est dit magique si la somme des éléments de n'importe quelle ligne , de n'importe quelle colonne , et de n'importe quelle diagonale est la même .
Utiliser les procédures précédentes pour écrire une fonction booléenne ***magique(T , n)*** rendant la valeur *true* si *T* est magique, et *false* sinon .

Exemples:

$$T := \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

```
> magique(T,3);
```

true

$$U := \begin{bmatrix} 5 & 6 & 3 \\ 0 & 8 & 1 \\ 4 & 7 & 2 \end{bmatrix}$$

```
> magique(U,3);
```

false

TD 4.2:
Ecrire la procédure intitulée ***coder(st::string)*** qui convertit la chaîne de caractères *st* (écrite en lettres minuscules) en un nombre.

On code : $a \to 1$, $b \to 2$, etc ... , $z \to 26$.

Exemple: si $st = maple$ alors $m \to 13$, $a \to 1$, $p \to 16$, $l \to 12$, $e \to 5$.

```
> coder("maple");
```

1301161205

Aide: pour écrire cette procédure , on construira un alphabet puis une table T de conversion *lettres* $\to$ *nombres* et on utilisera au besoin les fonctions de Maple suivantes:

length(st) qui donne la longueur de la chaîne de caractères *st* .
substring(st,m..n) qui extrait de *st* la sous-chaîne des caractères situés entre les positions *m* et *n*.