Chap. II : Les objets structurés

Laurent Poinsot

12 février 2009

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2+3*I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2 + 3 * I$). En les combinant, on peut construire des objets de types "structurés".

Pour connaître le type d'un objet `Maple` il faut utiliser la commande `whattype` :

```
> whattype(3);
```

integer

Dans ce cours nous avons déjà rencontré des objets de types "simples", tels que `integer`, `fraction` (ex. $2/3$), `float` (ex. 2.3, π) et `complex` (ex. $2 + 3 * I$). En les combinant, on peut construire des objets de types "structurés".

Voici la liste des cinq types structurés disponibles sous `Maple` :

1. Les **séquences** (`exprseq`) ;
2. Les **listes** (`list`) ;
3. Les **ensembles** (`set`) ;
4. Les **chaînes de caractères** (`string`) ;
5. Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous `Maple` :

1. Les **séquences** (`exprseq`) ;
2. Les **listes** (`list`) ;
3. Les **ensembles** (`set`) ;
4. Les **chaînes de caractères** (`string`) ;
5. Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous `Maple` :

1. Les **séquences** (`exprseq`) ;
2. Les **listes** (`list`) ;
3. Les **ensembles** (`set`) ;
4. Les **chaînes de caractères** (`string`) ;
5. Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous `Maple` :

1. Les **séquences** (`exprseq`) ;
2. Les **listes** (`list`) ;
3. Les **ensembles** (`set`) ;
4. Les **chaînes de caractères** (`string`) ;
5. Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous `Maple` :

1. Les **séquences** (`exprseq`) ;
2. Les **listes** (`list`) ;
3. Les **ensembles** (`set`) ;
4. Les **chaînes de caractères** (`string`) ;
5. Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Voici la liste des cinq types structurés disponibles sous `Maple` :

1. Les **séquences** (`exprseq`) ;
2. Les **listes** (`list`) ;
3. Les **ensembles** (`set`) ;
4. Les **chaînes de caractères** (`string`) ;
5. Les **tableaux** (on les verra plus tard dans le chapitre sur les matrices).

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

exprseq

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

a

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à *l* (*l* est la longueur de la séquence). Par ex. dans *S*, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro *n* dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de *S* :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

Une **séquence** est une suite d'objets quelconques, séparés par des virgules :

```
> S :=1,a,2,X,3;
```

$$S := 1, a, 2, X, 3$$

```
>whattype(S);
```

```
exprseq
```

Les éléments dans une séquence sont numérotés de 1 à l (l est la longueur de la séquence). Par ex. dans S, l'élément numéro 2 est `a`. Pour sélectionner l'élément numéro n dans une séquence `seq`, on tape `seq[n]`. Par ex. pour obtenir l'élément numéro 2 de S :

```
> S[2];
```

$$a$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

On peut changer la valeur de l'élément numéro 3 de S par :

```
> S; # Affiche la séquence S
```

$$1, a, 2, X, 3$$

```
>S[3]; # Affiche l'élément 3 de S
```

$$2$$

```
>S[3] :=Pi; # On remplace 3 par π dans S
```

$$S[3] := \pi$$

```
>S; # On affiche de nouveau la séquence S
```

$$1, a, \pi, X, 3$$

La **séquence vide** se note `NULL` :

```
> T :=NULL;
```

$$T :=$$

La **séquence vide** se note `NULL` :

```
> T :=NULL;
```

$$T := $$

La **séquence vide** se note `NULL` :

```
> T :=NULL;
```

$$T :=$$

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on les écrit séparées par des virgules ",” :

```
> U :=4,6,8 :
>S,T,U;
```

$$1, a, \pi, X, 3, 4, 6, 8$$

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on les écrit séparées par des virgules "," :

```
> U :=4,6,8 :
>S,T,U;
```

$$1, a, \pi, X, 3, 4, 6, 8$$

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on les écrit séparées par des virgules "," :

```
> U :=4,6,8 :
>S,T,U;
```

$$1, a, \pi, X, 3, 4, 6, 8$$

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on les écrit séparées par des virgules "," :

```
> U :=4,6,8 :
>S,T,U;
```

$$1, a, \pi, X, 3, 4, 6, 8$$

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on les écrit séparées par des virgules "," :

```
> U :=4,6,8 :
>S,T,U;
```

$$1, a, \pi, X, 3, 4, 6, 8$$

Pour **concaténer** des séquences, c'est-à-dire les juxtaposer les unes à la suite des autres, on les écrit séparées par des virgules "," :

```
> U :=4,6,8 :
>S,T,U;
```

$$1, a, \pi, X, 3, 4, 6, 8$$

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Voici deux façons de créer automatiquement des séquences :

1. En utilisant le symbole ”$” :

```
> x$5;
```

$$x, x, x, x, x$$

2. En utilisant la commande `seq` :

```
seq(1/n,n=1..6);
```

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

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

list

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

list

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

$$list$$

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

$$list$$

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

list

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

list

Une **liste** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des crochets "[" et "]" :

```
> L :=[1,5,a,8];
```

$$L := [1, 5, a, 8]$$

```
>whattype(L);
```

list

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

Pour transformer une liste en une séquence, il faut enlever les crochets par la commande `op` :

```
> S :=op(L);
```

$$S := 1, 5, a, 8$$

Pour tansformer une liste en une séquence, il suffit de rajouter les crochets :

```
> S :=x,y,z,alpha;
```

$$S := x, y, z, \alpha$$

```
>L :=[S];
```

$$L := [x, y, z, \alpha]$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

La **liste vide** se note par des crochets vides ”[]” :

```
> M :=[];
```

$$M := []$$

Le nombre d'éléments de la liste est obtenu par la commande `nops` :

```
> L; # Pour afficher la valeur de L
```

$$[x, y, z, \alpha]$$

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

$$4$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
> L[2];
```

$$y$$

```
> op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
> L[2];
```

$$y$$

```
> op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
> L[2];
```

$$y$$

```
> op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour accèder à un élément dans une liste il y a deux façons de procéder. Considérons la liste L précédente. Pour sélectionner par exemple son deuxième élément :

```
1 > L[2];
```

$$y$$

```
2 > op(2,L);
```

$$y$$

On peut alors changer la valeur d'un élément dans une liste. Par exemple,

```
> L[3] :=a :
> L;
```

$$[x, y, a, \alpha]$$

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”“”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”“”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”“”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”“”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”“”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”‘”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”‘”.s

Pour transformer une liste en *somme* ou *produit*, on utilise la commande `convert` :

```
> M :=[2,8,4,x,z^2];
```

$$M := [2, 8, 4, x, z^2]$$

```
> convert(M,`+`);
```

$$14 + x + z^2$$

```
> convert(M,`*`);
```

$$64xz^2$$

Notez bien que le caractère ”+” ou ”*” doit être entouré par des accents graves ”‘”.s

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

On peut **sélectionner** (`select`) ou **supprimer** certains éléments d'une liste :

```
>L :=[seq(i,i=5..10)];
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour sélectionner les nombres premiers de cette liste, on utilise `select` avec la commande `isprime` (définie par `isprime (n)` est `true` si `n` est premier, et `false` dans le cas contraire) :

```
> select(isprime,L);
```

$$[5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Attention, la liste L n'a pas été modifiée !

```
> L;
```

$$L := [5, 6, 7, 8, 9, 10]$$

Pour la modifier on doit écrire :

```
> L :=select(isprime,L);
```

$$L := [5, 7]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Maintenant pour supprimer les éléments d'une liste, on utilise la commande `remove` :

```
> L :=[seq(i,i=5..10)] :
> remove(isprime,L);
```

$$[6, 8, 9, 10]$$

```
> L;
```

$$[5, 6, 7, 8, 9, 10]$$

```
> L :=remove(isprime,L) :
> L;
```

$$[6, 8, 9, 10]$$

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2) ;
```

false

```
>evalb(0=0) ;
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2) ;
```

false

```
>evalb(0=0) ;
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2) ;
```

false

```
>evalb(0=0) ;
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2) ;
```

false

```
>evalb(0=0) ;
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Plus généralement, si $f : x \mapsto f(x)$ est une **fonction booléenne**, c'est-à-dire $f(x) \in \{\texttt{true}, \texttt{false}\}$, on peut employer les commandes `select(L,f)` et `remove(L,f)` :

1. `select(f,L)` permet de sélectionner les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$;
2. `remove(f,L)` permet de supprimer les éléments $L[i]$ de la liste L tels que $f(L[i]) = \texttt{true}$.

Il arrive souvent d'utiliser la commande `evalb` pour construire une fonction booléenne. Soit `P` une certaine condition, alors `evalb(P)` renvoie `true` lorsque P est vraie, et renvoie `false` si P est fausse.

```
> evalb(3<2);
```

false

```
>evalb(0=0);
```

true

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

false

```
> positif(4);
```

true

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

false

```
> positif(4);
```

true

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

false

```
> positif(4);
```

true

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

$$false$$

```
> positif(4);
```

$$true$$

Exemple

Soit la liste :

```
> L :=[seq(i=-5..10)];
```

$$L := [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour supprimer les nombres positifs, il faut tout d'abord créer une *fonction booléenne* qui prend un entier relatif en argument et qui renvoie `true` si l'entier est positif et `false` si l'entier est négatif :

```
> positif := x -> evalb(x>=0);
```

$$positif := x \rightarrow evalb(0 \leq x)$$

```
> positif(-3);
```

false

```
> positif(4);
```

true

On peut maintenant procéder à la suppression des nombres négatifs de L :

```
> L;
```

$$[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> remove(positif,L);
```

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

On peut maintenant procéder à la suppression des nombres négatifs de L :

```
> L;
```

$$[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> remove(positif,L);
```

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

On peut maintenant procéder à la suppression des nombres négatifs de L :

```
> L;
```

$$[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> remove(positif,L);
```

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

On peut maintenant procéder à la suppression des nombres négatifs de L :

```
> L;
```

$$[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> remove(positif,L);
```

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

On peut maintenant procéder à la suppression des nombres négatifs de L :

```
> L;
```

$$[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

```
> remove(positif,L);
```

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

Fonction `map`

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ :=x->x+1; # Fonction successeur
```

$$succ := x \to x + 1$$

```
> L :=[3,-1,0,5] :
> map(succ,L);
```

$$[4, 0, 1, 6]$$

Fonction `map`

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ :=x->x+1; # Fonction successeur
```

$$succ := x \to x + 1$$

```
> L :=[3,-1,0,5] :
> map(succ,L);
```

$$[4, 0, 1, 6]$$

Fonction `map`

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ :=x->x+1; # Fonction successeur
```

$$succ := x \to x + 1$$

```
> L :=[3,-1,0,5] :
> map(succ,L);
```

$$[4, 0, 1, 6]$$

Fonction `map`

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ :=x->x+1; # Fonction successeur
```

$$succ := x \to x + 1$$

```
> L :=[3,-1,0,5] :
> map(succ,L);
```

$$[4, 0, 1, 6]$$

Fonction `map`

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ :=x->x+1; # Fonction successeur
```

$$succ := x \to x + 1$$

```
> L :=[3,-1,0,5] :
> map(succ,L);
```

$$[4, 0, 1, 6]$$

Fonction `map`

On peut appliquer une même fonction à tous les éléments d'une liste à l'aide de la commande `map` :

```
> succ :=x->x+1; # Fonction successeur
```

$$succ := x \to x + 1$$

```
> L :=[3,-1,0,5] :
> map(succ,L);
```

$$[4, 0, 1, 6]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Pour **ajouter un élément à la fin d'une liste** : On convertit la liste en une séquence par la commande `op`. Puis on concatène la séquence obtenue avec le nouvel élément grâce à la virgule ",". Enfin on transforme cette séquence en une liste en l'entourant des crochets :

```
> L :=[seq(i,i=1..9)];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9]$$

```
> L :=[op(L),10];
```

$$L := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

$$set$$

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

$$set$$

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}".

Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

$$set$$

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

$$set$$

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

$$set$$

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

set

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

set

Un **ensemble** est une suite d'objets quelconques, séparés par des virgules, et encadrée par des accolades "{" et "}". Dans un ensemble on ne tient compte ni de l'ordre ni des répétitions !

```
> E :={a,b,c,d};
```

$$E := \{a, b, c, d\}$$

```
>whattype(E);
```

set

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note {} :

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note {} :

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note {} :

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note {} :

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note {} :

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note {} :

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note $\{\}$:

```
> {};
```

On transforme un ensemble en une séquence par la commande `op` :

```
> op(E);
```

$$a, b, c, d$$

Pour transformer un ensemble en une liste, on utilise `convert` :

```
> convert(E,list);
```

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

L'ensemble **vide** se note $\{\}$:

```
> {};
```

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On peut effectuer différentes opérations sur les ensembles : **réunion** (`union`), **intersection** (`intersect`), **différence ensembliste** (`minus`) et le **test d'appartenance** d'un élément à un ensemble (`member`) :

```
> E :={a,b,c};
```

$$E := \{a, b, c\}$$

```
> F :={d,c,f,b,e};
```

$$F := \{d, c, f, b, e\}$$

```
> E union F;
```

$$\{a, b, c, d, f, e\}$$

```
> E intersect F;
```

$$\{c, b\}$$

```
> E minus F;
```

$$\{a\}$$

On a aussi :

```
> member(a,E);
```

true

```
> member (a,F);
```

false

On a aussi :

```
> member(a,E);
```

true

```
> member (a,F);
```

false

On a aussi :

```
> member(a,E);
```

true

```
> member (a,F);
```

false

On a aussi :

```
> member(a,E);
```

true

```
> member (a,F);
```

false

On a aussi :

```
> member(a,E);
```

true

```
> member (a,F);
```

false

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

$$mot := \text{”}bonjour\text{”}$$

```
> whattype(mot);
```

$$string$$

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

$$\text{”}onjo\text{”}$$

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

mot := ”*bonjour*”

```
> whattype(mot);
```

string

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

”*onjo*”

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

$$mot := "bonjour"$$

```
> whattype(mot);
```

$$string$$

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

$$"onjo"$$

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

$$mot := \text{"}bonjour\text{"}$$

```
> whattype(mot);
```

string

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

"*onjo*"

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

$$mot := \text{"}bonjour\text{"}$$

```
> whattype(mot);
```

$$string$$

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

$$\text{"}onjo\text{"}$$

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

$$mot := "bonjour"$$

```
> whattype(mot);
```

$$string$$

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

$$"onjo"$$

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (”).

```
> mot :="bonjour";
```

$$mot := "bonjour"$$

```
> whattype(mot);
```

$$string$$

Pour extraire une **sous-chaîne** on utilise les crochets ”[” et ”]” :

```
> mot[2..5];
```

$$"onjo"$$

Une **chaîne de caractères** est une suite de caractères encadrée par des guillemets (").

```
> mot :="bonjour";
```

$$mot := "bonjour"$$

```
> whattype(mot);
```

$$string$$

Pour extraire une **sous-chaîne** on utilise les crochets "[" et "]" :

```
> mot[2..5];
```

$$"onjo"$$

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat(mot," chez vous");
```

$$mot2 := "bonjour\ \ chez\ \ vous"$$

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length(mot2);
```

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat(mot," chez vous");
```

mot2 := "*bonjour chez vous*"

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length(mot2);
```

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat(mot," chez vous");
```

$$mot2 := \text{"}bonjour\ \ chez\ \ vous\text{"}$$

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length(mot2);
```

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat(mot," chez vous");
```

$$mot2 := "bonjour\ \ chez\ \ vous"$$

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length(mot2);
```

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat(mot," chez vous");
```

$$mot2 := "bonjour\ \ chez\ \ vous"$$

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length(mot2);
```

On peut **concaténer** deux chaînes de caractères par la commande `cat` :

```
> mot2 :=cat(mot," chez vous");
```

$$mot2 := "bonjour\ \ chez\ \ vous"$$

La **longueur** d'une chaîne est donnée par la commande `length` :

```
> length(mot2);
```