Chapitre III : Programmation sous Maple

Les instructions itératives : Boucles do

Introduction & règles de syntaxe.

Maple possède une instruction itérative unique, qui posséde à la fois les fonctionnalités du "faire pour" et du "tant que" des languages de programmation classique, il s'agit de l'instruction for dont la forme génerale est donnée ci dessous:

for cpt from I0 by p to In while cond do
instruction 1
instruction 2
.....
od;

Cette instruction effectue une boucle controlée à la fois par la condition logique **cond** et par le compteur **cpt** qui varie de **I0** jusqua **In** par pas de **p**.la boucle est fermée par l'instruction **od** ou **end do**

```
> for i from 1 to 5 do
  i! od;
```

1

2

6

24

120

```
> i;
```

6

Remarque 1 : La variable de contrôle de la boucle (i dans cet exemple) n'est pas muette, elle est évaluée et prend la dernière valeur calculée.
Remarque 2 : Il faut faire attention au fait que les bornes des boucles I0 et In doivent être du type **numeric,** le type constant n'est pas accepté, par exemple on ne peut pas finir une boucle par Pi ou exp(1) comme on peut le constater dans les exemples suivants.

```
> for i from 1 to Pi do cos(i); od;
Error, final value in for loop must be numeric or character
```

```
> for i from 1 to exp(1) do ln(i); od;
Error, final value in for loop must be numeric or character
```

```
> S:=0:
  for i from 1 to sqrt(5) do
  S:=S+i;od;
Error, final value in for loop must be numeric or character
```

Par contre on peut utiliser la valeur approchée d'une constante pour contrôler une boucle.

```
> for i from 1 to evalf(Pi) do cos(i); od;
```

$$\cos(1)$$

$$\cos(2)$$

$$\cos(3)$$

```
> for i from 1 to exp(1.) do ln(i); od;
```

$$0$$

$$\ln(2)$$

```
> S:=0:
  for i from sqrt(2.) to sqrt(5.) do
  S:=S+i;od;
```

$$S := 1.414213562$$

Si les mots **from** et **by** ne sont pas spécifiés ils prennent la valeur 1

```
> S:=0:
  for i to sqrt(5.) do
  S:=S+i;
  od;
```

$$S := 1$$

$$S := 3$$

Pour la ponctuation c'est celle de l'instruction **od** qui décide de l'affichage des résultats si elle se termine par ; les résultats des instructions internes de la boucle sont affichés même si elles se terminent par : et si l'instruction **od** se termine par : les résultats des instructions internes de la boucle ne sont pas affichés même si elles terminent par ;

```
> S:=0:
  for i to sqrt(5.) do
  S:=S+i:
  od;
```

$$S := 1$$

$$S := 3$$

```
> S:=0:
  for i to sqrt(5.) do
  S:=S+i;
  od:
> S;
```

$$3$$

On peut ommettre l'indice de la boucle s'il n'est pas nécéssaire comme dans une méthode itérative par exemple, l'objectif étant simplement la répétition du même bloc d'instructions

```
> x:=0.;
  for i from 1 to 5 do
  x:=cos(x):
  od;
```

$$x := 0.$$

$$x := 1.$$

$$x := 0.5403023059$$

$$x := 0.8575532158$$

$$x := 0.6542897905$$

$$x := 0.7934803587$$

Choix de l'indice de la boucle dans une liste

On peut choisir de prendre les indices d'une boucle dans une liste donnée, pour celà on utilise l'opérateur **in.**

```
> Liste:=[-1,3,-2,1,0,4,5];
```

$$Liste := [-1, 3, -2, 1, 0, 4, 5]$$

```
> S:=0:
  for i in Liste do
  i;
  S:=S+i:
  od;
```

$$-1$$

$$S := -1$$

$$3$$

$$S := 2$$

$$-2$$

$$S := 0$$

$$1$$

$$S := 1$$

$$0$$

$$S := 1$$

$$4$$

$$S := 5$$

$$5$$

$$S := 10$$

Mot clé while

Utilisation de tests booléen / mot clé **while**
On dispose des mots clés suivants pour effectuer les tests logiques
a<b , a<=b, a=b, a<>b (a différent de b)
On peut combiner plusieurs expressions à valeurs booléene avec les opérateurs and (et), or (ou) et not (négation).

```
> a,b:=3,5:
> not(a<=b);
```

$$false$$

```
> x:=5.:
  while x>0 do
  x:=sqrt(x-1):
  od;
```

$$x := 2.000000000$$

$$x := 1.000000000$$

$$x := 0.$$

Il faut faire attention ici à une erreur qui peut se produire si on utilise uniquement le mot clé **while** qui concerne le risque de boucle infinie, en effet l'analyseur syntaxique de Maple qui peut détecter les affectations conduisant à des définitions récursives infinies ne peut pas détecter le risque d'une boucle infinie sur une condition logique.

```
> restart:
  x:=x+1;
Error, recursive assignment
```

```
> for i from 1 to 5 do
  x:=x+1;od;
```

$$x := x + 1$$

```
Error, too many levels of recursion
```

```
> x:=1.:
  while x<0 do
  x:=sqrt(x):
  od;
```

Sorties prématurées d'une boucle (mots clés next & break)

Saut d'indice dans une boucle mot clé **next** .
Le mot clé next permet de ne pas effectuer une séquence d'instructions non désirée, il est couplé à un test de choix si ce test de choix est vérifié, l'indice passe à la valeur suivante.

```
> L:=[-2,-1,0,3];
```

$$L := [-2, -1, 0, 3]$$

```
> P:=1:
  for i from 1 to nops(L) do
  if L[i]=0 then next fi;
  P:=P*L[i]:od;
```

$$P := -2$$

$$P := 2$$

$$P := 6$$

Sortie prématurée d'une boucle.
On peut sortir d'une boucle avant de l'avoir totalement parcourue, le mot clé **break** permet d'arréter l'exécution des instructions dans la boucle quand une condition logique est vérifiée et d'en sortir.

```
> L:=[-2,-1,0,3]:
```

```
P:=1:
for i from 1 to nops(L) do
if L[i]=0 then break fi;
P:=P*L[i]:od;
```

$$P := -2$$

$$P := 2$$

Les instructions conditionnelles:

Maple dispose également d'une instruction conditionnelle unique, qui lui permet de simuler facilement les deux instructions conditionnelles classiques que sont le "si alors sinon" et le "cas" Dans sa version de base la syntaxe de cette instruction est la suivante :

if cond then
instructions1;
else
instructions2;
fi;

l'exécution de cette instruction est évidente,si la condition logique cond est vérifié, on exécutera le bloc instructions1, si ce n'est pas le cas c'est le bloc instructions2, qui sera au contraire exécuté, pour les test conditionelles à plusieurs choix ,on dispose en plus de la primitive elif (contraction de else et if), elle sert à offrir plusieurs possibilitées, son utilisation est illustré par le schéma suivant :

if cond1 then
instructions1;
elif cond2 then
instructions2;
else
instructionsN;
fi;

```
> x:=1.5:
  if x<2 then x^2
  else -x^2
  fi;
```

$$2.25$$

Si le test de choix comporte uniquement deux possibilités on peut utiliser le mot clé **if** en tant qu'opérateur **`if`**(cond, instruction_vraie, instruction_faux), si la condition cond est vérifiée alors c'est la première instruction qui est exécutée sinon c'est la seconde.

```
> x:=1.5:
  `if`(x<2,x^2,-x^2);
```

$$2.25$$

Il faut garder en mémoire que les test de comparaisons ne fonctionnent que si les valeurs sont de types (**numeric**) sinon le test échoue.

```
> x:=2:
  if x<Pi then x^2
  else -x^2
  fi;
```

Error, cannot determine if this expression is true or false: 2

```
< Pi
```

```
> x:=2:
  if x<evalf(Pi) then x^2
  else -x^2
  fi;
```

$$4$$

```
> x:=2:
  if x<sqrt(3) then x^2
  else -x^2
  fi;
```

```
Error, cannot determine if this expression is true or false: 2
< 3^(1/2)
```

```
> x:=2:
  if x<sqrt(3.) then x^2
  else -x^2
  fi;
```

$$-4$$

Les procédures.

Une procédure est un ensemble d'instructions qui sont capables de traiter des données pour founir un résultat, elle peut étre vue comme une fonction dépandant des paramétres (p1,...pr).

Cette vision est d'ailleurs celle de Maple.Une procédure Maple reçoit en effet une suite de paramètres comme arguments et retourne une valeur unique qui est par définition la dernière valeur calculée.

Syntaxe d'une procédure Maple:

Une procédure Maple se déclare par le mot clef **proc** suivi de la suite des arguments de la procédure placés entre parenthéses, suivi de la suite des variables locales de la procédure, et enfin le corps de la procédure constituée d'une suite d'instructions à exécuter .Le résultat de la dernière instruction exécutée par la procédure est la valeur qu'elle retourne.

Format d'une procédure Maple :

proc(suite-de paramètres)
local suite-de-variables-locales;
instruction-1;
instruction-2;
.....
.....
résultat;
end:

```
> restart:
  f:=proc(x);
  x^2;
  end;
```

$$f := \mathbf{proc}(x)\ \ x\text{^}2\ \ \mathbf{end\ proc}$$

```
> f(5);
```

$$25$$

```
> f(a);
```

$$a^2$$

```
> f(a+b);
```

$$(a+b)^2$$

```
> Somme:=proc(n)
  local i,S;
  S:=0:
  for i from 1 to n do
  S:=S+i;
  od:
  end;
```

$$Somme := \mathbf{proc}(n)\ \mathbf{local}\ i, S;\ S := 0;\ \mathbf{for}\ i\ \mathbf{to}\ n\ \mathbf{do}\ S := S + i\ \mathbf{end\ do}\ \mathbf{end\ proc}$$

```
> Somme(10);
```

$$55$$

Remarque:
On ne peut pas utiliser le nom de la variable globale comme nom de variable locale.

```
> restart:
  Somme:=proc(n)
  local i,S,n;
  S:=0:
  for i from 1 to n do
  S:=S+i;
  od:
  end;
```

```
Error, parameter and local `n` have the same name in procedure
Somme
```

```
> Somme(10);
```

$$Somme(10)$$

Remarque:
On ne peut pas changer (affecter) la valeur de la variable globale à l'intérieur de la procédure, bien que l'analyseur syntaxique ne détecte pas l'erreur, elle apparaitra à l'éxécution de la procédure.

```
> restart:
  f:=proc(x)
  x:=x^2+1:
  end;
```

$$f := \mathbf{proc}(x)\ x := x\text{^}2 + 1\ \mathbf{end\ proc}$$

```
> f(10);
```

```
Error, (in f) illegal use of a formal parameter
```

La valeur d'une procédure est la valeur de la dernière instruction effectuée (et non du dernier calcul)

```
> restart:
```

```
J:=[-1,2,3,4,5,0,-5]:
Produit:=proc(a,L)
local i,L1;
for i from 1 to nops(L)do
L1[i]:=a*L[i]:
od:
end:
```

```
> Produit(5,J);
```

$$-25$$

```
> nfact:=proc(n)
local i,fact:
i:=n:
fact:=1:
while i>0 do
fact:=fact*i:
i:=(i-1):
od:
print(fact):
end:
```

```
> nfact(5);
```

$$120$$

```
> exp(nfact(4));
```

$$24$$

```
Error, (in exp) expecting 1 argument, got 0
```

Si on souhaite dans notre exemple avoir comme résultat le factoriel, il faut s'arranger pour que la dernière instruction évalue le factoriel, par exemple :

```
> nfact:=proc(n)
local i,fact:
i:=n:
fact:=1:
while i>0 do
fact:=fact*i:
i:=(i-1):
od:
fact:
end:
```

```
> nfact(5);
```

$$120$$

```
> exp(nfact(4));
```

$$e^{24}$$

Arguments d'une procédure

Spécification du type des arguments

on peut spécifier le type d'arguments que l'on souhaite utiliser dans une procédure, cette fonctionnalité permettra d'éviter les utilisations non prévues par le programmeur.

```
> nfact:=proc(n::integer)
  local i,fact;
  fact:=1:
  for i from 1 to n do
  fact:=fact*i;
  od;
  fact;
  end:
```

```
> nfact(5.5);
```

Error, invalid input: nfact expects its 1st argument, n, to be of type integer, but received 5.5

```
> nfact(-5);
```

1

```
> nfact:=proc(n::posint)
  local i,fact;
  fact:=1:
  for i from 1 to n do
  fact:=fact*i;
  od;
  fact;
  end:
```

```
> nfact(-5);
```

Error, invalid input: nfact expects its 1st argument, n, to be of type posint, but received -5

```
> h:=proc(x::integer) if x>0 then x^2;elif x<0 then -x^2;fi;
  end:
```

```
> h(1);
```

1

```
> h(1/2);
```

Error, invalid input: h expects its 1st argument, x, to be of type integer, but received 1/2

```
> h(0.5);
```

Error, invalid input: h expects its 1st argument, x, to be of type integer, but received .5

```
> l:=proc(x::fraction)
  if x>0 then x^2;
  elif x<0 then -x^2;
```

```
  fi;
  end:
> l(1/2);
```

$$\frac{1}{4}$$

```
> l(1);
```

Error, invalid input: l expects its 1st argument, x, to be of type fraction, but received 1

```
> l(0.25);
```

Error, invalid input: l expects its 1st argument, x, to be of type fraction, but received .25

```
> l:=proc(x::float)
  if x>0 then x^2;
  elif x<0 then -x^2;
  fi;
  end:
> l(1);
```

Error, invalid input: l expects its 1st argument, x, to be of type float, but received 1

```
> l(1/2);
```

Error, invalid input: l expects its 1st argument, x, to be of type float, but received 1/2

```
> l(1.);
```

$$1.$$

Arguments d'une procédure en liste ouverte

Il existe dans Maple la possibilité de manipuler des arguments d'une procédure dont on ne connait pas le nombre d'avance, lors de la création d'un polynôme par exemple, dans ce cas on omet de donner les arguments de la procédure, lors de l'appel, les arguments entrés sont placés dans une suite à laquelle on peut faire appel par la commande **args** qui renvoie sous forme de suite les arguments entrée dans la proécdure**,** l'appel **args[i]** permet d'accéder au ième argument, l'appel **nargs** permet d'obtenir le nombre total d'arguments entrés.

```
> poly:=proc()
  local P,i;
  P:=0:
  for i from 1 to nargs do
  P:=P+args[i]*x^(i-1);
  od;
  end:
> poly(1,-1,1,2,4,7);
```

$$7x^5 + 4x^4 + 2x^3 + x^2 - x + 1$$

```
> testargs:=proc()
  print(args);
```

```
  whattype(%);
  end:
> testargs(1,-1,1,2,4,7);
```

$$1, -1, 1, 2, 4, 7$$

exprseq

Variables locales / globales

Instruction local

Les variables locales sont des variables internes à une proécdure, elles ne sont pas stockés à la sortie de la procédure, elles sont déclarés par le mot clé **local** par l'utilisateur, celles qui sont omises sont automatiquement déclarés par l'analyseur syntaxique de Maple tout en affichant un message d'avertissement.

```
> nfact:=proc(n)
  local i:
  i:=n:
  fact:=1:
  while i>0 do
  fact:=fact*i:
  i:=(i-1):
  od:
  fact:
  end:
```

```
Warning, `fact` is implicitly declared local to procedure `nfact`
```

```
> nfact(4);
```

$$24$$

```
> i;
```

i

```
> fact;
```

fact

On remarque bien qu'il ne reste aucune trace des variables locales i et fact après exécution de la procédure.

Exportation de variables locales.
On dit qu'une variable locale a été exportée si jamais elle apparait dans la valeur de la proécdure.

```
> restart:
  A:=proc()
  local x;
  x^2;
  end:
> A();
```

$$x^2$$

Une variable locale exportée n'est pas considérée par Maple comme le même objet que la même variable entrée par l'utilisateur, les variables entrées par l'utilisateur sont dites de premiers niveau.

```
> {A(),x^2};
```

$$\{x^2, x^2\}$$

```
> evalb(A()=x);
```

$$false$$

```
> int(A(),x);
```

$$x^2\, x$$

Instruction global

Les variables globales sont contrairement aux variables locales stockés après exécution de la proécdure, on les déclarent avec le mot clé **global.**

```
> nfact:=proc(n)
  local i,fact:
  i:=n:
  fact:=1:
  while i>0 do
  fact:=fact*i:
  i:=(i-1):
  od:
  fact:
  end:
> nfact(4);
```

$$24$$

```
> fact;
```

$$fact$$