Les Structures algorithmiques avec PowerShell

Table des matières

1. Les boucles

Une boucle est une structure répétitive qui permet d'exécuter plusieurs fois les instructions se trouvant à l'intérieur du bloc d'instructions. Il est important de maîtriser le concept des boucles car on le retrouve dans tout langage informatique et il est à la base de tout programme quel que soit le langage.

1.1 Boucle While

Les instructions du bloc d'instructions de cette boucle sont répétées tant que la condition de la boucle est satisfaite (est vraie). Autrement dit, dès que la condition est évaluée à la valeur $false, on sort de la boucle.

La syntaxe d'une boucle While est la suivante :

```
While (<condition>)
{
#bloc d'instructions
}
```

Son fonctionnement est le suivant :

1. La boucle évalue la condition.
2. Si la condition est fausse, le bloc d'instruction n'est pas exécuté et la boucle se termine.
3. Si la condition est vraie, alors le bloc d'instructions est exécuté.
4. Retour à l'étape 1.

Voici un exemple basique d'une boucle **While** qui liste les valeurs contenues dans un tableau. Dans cette boucle, tant que la valeur $nombre est strictement inférieure à la taille du tableau, le bloc d'instructions retourne la valeur du tableau à l'indice $nombre.

```
$nombre = 0
$tab = 0..99
While($nombre -lt $tab.Length)
{
$tab[$nombre]
$nombre++
}
```

1.2 Boucle Do-While

La boucle Do-While s'apparente à la boucle While, à la différence près que la condition est évaluée à la fin. La boucle Do-While se structure de la façon suivante :

```
Do
{
#bloc d'instructions
}
While (<condition>)
```

Le test de condition étant à la fin, le bloc d'instructions est exécuté au moins une fois, même si le test est faux. Par exemple, avec la boucle suivante, l'utilisateur est amené à saisir un nombre entre 0 et 10 une première fois. Si le nombre saisi ne s'avère pas compris entre ces valeurs, alors le bloc d'instructions sera exécuté de nouveau.

```
Do
{
[int]$var = Read-Host 'Entrez une valeur entre 0 et 10'
}

While( ($var -lt 0 ) -or ($var -gt 10) )
```

1.3 Boucle Do-Until

Cette boucle est la jumelle de la boucle Do-While . En français, celle-ci signifie littéralement « boucle jusqu'à ce que… ». En effet, avec cette structure, on boucle jusqu'à ce que la condition de sortie soit évaluée à vrai ($true). Autrement dit, tant que celle-ci est fausse, on continue d'exécuter le bloc de script. La condition de sortie est donc inversée par rapport à la boucle Do-While.

```
Do
{
[int]$var = Read-Host 'Entrez une valeur entre 0 et 10'
}
Until( ($var -ge 0 ) -and ($var -le 10) )
```

1.4 Boucle For

La boucle For permet d'exécuter un certain nombre de fois un bloc d'instructions.

Lorsque l'on utilise une boucle For, on indique une valeur de départ, une condition de répétition de la boucle ainsi que le pas d'incrémentation, c'est-à-dire la valeur dont elle est augmentée à chaque itération.

La syntaxe de la boucle For est la suivante :

```
For (<initial> ;<condition> ;<incrément>)
{
#bloc d'instructions
}
```

Son fonctionnement est le suivant :

1. L'expression initiale est évaluée, il s'agit en général d'une affectation qui initialise une variable.
2. La condition de répétition est évaluée.
3. Si la condition est fausse, l'instruction For se termine.

4. Si la condition est vraie, le bloc d'instructions est exécuté.

5. L'expression est incrémentée avec le pas choisi et l'exécution reprend à l'étape 2. Reprenons l'exemple du parcours d'un tableau, mais cette fois-ci avec une boucle For.

```
$tab = 0..99
For($i=0 ;$i -le 99 ;$i++)
{
$tab[$i]
}
```

1.5 Boucle Foreach

La boucle Foreach , bien que d'apparence simple, est probablement celle qui donne le plus de fil à retordre aux débutants. Et ce, non pas parce qu'elle est difficile à utiliser, mais parce qu'il est possible de l'utiliser de différentes façons. Chaque façon ayant évidemment quelques petites spécificités à connaître…

Ce type de traitement, lorsque maîtrisé, est de loin le plus pratique pour parcourir une collection de données. Contrairement à la boucle For, il n'est pas nécessaire de déterminer à l'avance le nombre d'éléments contenus dans la collection ; ce qui contribue à éviter des erreurs et fait gagner du temps.

1.5.1 Première technique

Bien qu'utilisable en ligne de commandes, c'est-à-dire directement dans la console, on trouve plus fréquemment cette forme dans les scripts. En effet, elle s'apparente beaucoup au traitement Foreach du C#. Dans cette forme, nous n'avons pas affaire à une cmdlt mais à un mot-clé.

La syntaxe est la suivante :

```
Foreach (<élément> in <collection>)
{
#bloc d'instructions
}
```

Le principe est d'utiliser une variable arbitraire qui n'aura d'existence qu'à l'intérieur du bloc Foreach. Cette variable recevra à chaque itération un élément de la collection, et ce jusqu'à ce que toute la collection soit parcourue. Ainsi, dans le bloc de script nous utiliserons cette variable exactement comme une autre.

Observons l'exemple suivant :

```
Foreach ($element in Get-Process) {
'{0} démarré le : {1}' -f $element.Name, $element.StartTime
}
```

Résultat :

```
...
Notepad    démarré le : 2/12/2021 09:57:00
Powershell démarré le : 2/12/2021 09:09:24
WINWORD    démarré le : 2/12/2021 08:57:40
...
```

Vous l'avez sans doute remarqué, il n'y a nul besoin de stocker le résultat de la commande Get- Process dans une variable. Lors de l'exécution, PowerShell commence par évaluer Get-Process, puis stocke en mémoire toute la collection d'objets. Si pour vous la consommation mémoire est un problème car vous avez un très gros volume de données à traiter, alors mieux vaut vous tourner vers la seconde forme d'utilisation de Foreach.

Pour revenir à notre exemple, lors de la première itération, la variable $element contient le premier objet renvoyé par Get-Process . Chaque élément de la collection étant un objet, nous pouvons agir sur ses propriétés et ses méthodes. À l'itération suivante $element contient le second objet de la collection, et ainsi de suite jusqu'à passer en revue tous les objets.

1.5.2 Seconde technique

Dans cette forme, la collection d'objets à traiter est passée via le pipeline. Ici ce n'est donc plus le mot-clé Foreach qui est utilisé mais bien la commande Foreach-Object . Cependant, il est malgré tout possible d'utiliser l'alias Foreach afin de s'économiser quelques frappes de touches. On rencontre généralement cette forme davantage dans la console que dans les scripts. Cela tient probablement au fait que l'usage du pipe est très naturel en mode « interactif » et l'est peut-être un peu moins lorsque l'on écrit un script dans un éditeur.

Observons l'exemple suivant, dont le résultat est le même que celui de l'exemple précédent :

```
Get-Process | Foreach-Object {
'{0} démarré le : {1}' -f $_.Name, $_.StartTime
}
```

Cette fois il n'est plus question de définir une variable arbitraire pour contenir chaque objet de la collection, mais au contraire il est question d'utiliser la variable automatique $_. Cette variable contient l'objet courant transitant dans le pipeline à un instant t.

Un autre avantage de Foreach-object est que cette commande autorise l'exécution d'un bloc de script avant et après le traitement principal grâce aux paramètres -begin et -end. En effet, sans le savoir nous faisons appel au paramètre -process lorsqu'on utilise cette commande dans sa forme la plus simple.

Exemple

```
Get-Process | Foreach-Object -Begin {'* Début du traitement *'} `
-Process {'{0} démarré le : {1}' -f $_.Name, $_.StartTime}`
-End {'* Fin du traitement *'}
```

Ainsi, dans cet exemple, nous effectuons un affichage de chaîne au début et à la fin du traitement qui précise les actions réalisées, mais nous aurions pu tout à fait faire autre chose.

Résultat

```
* Début du traitement *
...
Notepad    démarré le : 2/12/2021 09:57:00
Powershell démarré le : 2/12/2021 09:09:24
WINWORD    démarré le : 2/12/2021 08:57:40
...
* Fin du traitement *
```

2. Structure conditionnelle If, Else, ElseIf

Une structure conditionnelle permet, via une évaluation de condition, d'orienter l'exécution vers un bloc d'instructions ou vers un autre. La syntaxe d'une structure conditionnelle est la suivante :

```
If (condition)
{
#bloc d'instructions
}
```

Pour mieux comprendre l'utilisation d'une structure conditionnelle, en voici quelques exemples :

Imaginons que nous souhaitions déterminer si une valeur entrée par l'utilisateur est la lettre A. Pour cela, nous allons utiliser une structure conditionnelle avec une condition sur la valeur de la variable testée. En utilisant un opérateur de comparaison, la structure est la suivante :

```
$var = Read-Host "Entrez un caractère"
If ($var -eq 'A')
{
"Le caractère saisi par l'utilisateur est un 'A'"
}
```

Si la variable entrée par l'utilisateur est un A, alors une chaîne de caractères sera émise, sinon l'exécution poursuivra son cours sans exécuter le bloc d'instructions suivant le If. À l'instruction If peut être associée la clause Else. Cette clause permet en cas de retour d'une valeur False d'orienter le traitement vers un second bloc d'instructions. Prenons l'exemple suivant :

```
 If (($var1 -eq 15) -and ($var2 -eq 18))
{
# Bloc d'instructions 1
}
Else
{
# Bloc d'instructions 2
}
```

Dans un premier temps, PowerShell évalue la première condition, à savoir « la variable $var1 est égale à 15 ». Si le test est vrai alors la première condition prend la valeur true. Puis, il évalue la seconde condition « la variable $var2 est égale à 18 ». Si le test est vrai alors la seconde condition prend également la valeur true. Puis, si les deux valeurs sont vraies, l'opérateur logique -and de la condition retourne la valeur true (vrai ET vrai = vrai), et ainsi le bloc d'instruction 1 sera exécuté, sinon, si la condition est fausse, ce sera le bloc d'instructions 2 qui sera exécuté.

Toujours dans le même registre voici un autre exemple :

```
[int]$var1 = Read-Host 'Saisissez un nombre' [int]$var2 = Read-Host
'Saisissez un nombre'

If ($var1 -ge $var2)
{
"$var1 est plus grand ou égal que $var2"
}
Else
{
"$var1 est plus petit que $var2"
}
```

Dans ce second exemple, l'utilisateur saisit deux nombres stockés respectivement dans les variables $var1 et $var2. PowerShell teste ensuite si la première variable est plus grande ou égale à la seconde. Si la condition est vraie, alors nous retournons une chaîne indiquant que la première valeur est plus grande ou égale à la seconde. Si la condition est fausse, c'est la chaîne se situant dans le bloc d'instructions de la clause Else qui est retournée.

Enfin, pour finir sur les structures conditionnelles voici comment les améliorer grâce à l'instruction ElseIf. L'instruction ElseIf permet, si la condition précédente est fausse, de tester une autre condition. Ainsi, en utilisant une structure conditionnelle avec des ElseIf, nous ne nous limitons plus à une orientation binaire, mais nous augmentons les possibles orientations du flot d'exécution.

Exemple

```
[int]$val = Read-Host 'Entrez une valeur : 1,2 ou 3'
If ($val -eq 1)
{ 'La valeur saisie est égale à 1'} ElseIf($val -eq 2)
{ 'La valeur saisie est égale à 2'} ElseIf($val -eq 3)
{ 'La valeur saisie est égale à 3'} Else
{ "La valeur saisie n'est pas égale à 1 ni à 2 ni à 3 !"}
```

De cette manière, on aurait pu créer autant de ElseIf que voulu. Mais l'utilisation intensive des ElseIf, bien que viable, n'est pas une solution très élégante. Le fait qu'il y ait autant de conditions que de blocs d'instructions ne rend pas le code très souple ni très lisible, et on préférera s'orienter vers l'instruction Switch.

2.1 Switch

L'instruction Switch permet de remplacer avantageusement toute une série de If, ElseIf et Else. À la différence de l'instruction If qui, pour une expression donnée, oriente la suite de l'exécution vers l'un des deux blocs d'instructions, l'instruction Switch oriente l'exécution vers plusieurs blocs d'instructions distincts. Et ce, avec une seule expression. Ce qui lui confère une utilisation nettement plus souple.

On peut construire un Switch de plusieurs manières, en fonction de ce que l'on veut tester.

2.1.1 Structure simple

La syntaxe de Switch est la suivante :

```
Switch (<Expression>)
{
<Valeur_1> { bloc d'instructions 1 ; Break }
<Valeur_2> { bloc d'instructions 2 ; Break }
<Valeur_3> { bloc d'instructions 3 ; Break }

Default   { bloc d'instructions 4 }
}
```

La valeur Default est facultative, son bloc d'instructions n'est exécuté uniquement dans le cas où l'expression ne correspond à aucune valeur.

Il est d'usage d'utiliser l'instruction Break à l'intérieur des blocs d'instructions. Cela permet de quitter le traitement Switch en cours. En effet, si l'on exécute un bloc d'instructions, c'est que la valeur évaluée correspond à l'expression. Il n'y a donc pas lieu de tester les autres valeurs.

Prenons pour exemple d'application, le cas basique où l'utilisateur doit saisir un nombre entre 1 et 5, et PowerShell détermine quel nombre a été saisi. Le code est le suivant :

```
[Int]$Nombre = Read-Host 'Entrez un nombre compris entre 1 et 5 '


Switch($Nombre)
{
1 { 'Vous avez saisi le nombre 1 ' ; Break }
2 { 'Vous avez saisi le nombre 2 ' ; Break }
3 { 'Vous avez saisi le nombre 3 ' ; Break }
4 { 'Vous avez saisi le nombre 4 ' ; Break }
5 { 'Vous avez saisi le nombre 5 ' ; Break }
Default { "Nombre saisi incorrect !"}
}
```

2.1.2 Structure à base de sous-expressions

On peut aussi utiliser l'instruction Switch à l'aide de sous-expressions afin de se donner plus de souplesse d'utilisation. Cette forme est très peu utilisée car elle n'est pas documentée par Microsoft, ce qui est bien dommage car elle s'avère extrêmement pratique. Elle remplacera avantageusement les tests spaghettis intrinsèques aux structures if/elseif/else.

Exemple :

```
[Int]$Nombre = Read-Host 'Saisissez un nombre entier positif'

Switch($Nombre)
{
{$_ -lt 10} { 'Nombre saisi strictement inférieur à 10 '; Break}
{$_ -ge 10 -and $_ -le 20} { 'Nombre saisi est entre 10 et 20' ; Break}
Default { 'Nombre saisi supérieur à 20'}
}
```

Lorsque l'on utilise Switch avec des sous-expressions, la valeur testée est affectée à la variable automatique $_.