

Première approche de PowerShell

par Laurent Dardenne ([Contributions](#)) >

Date de publication : 22/05/2007

Dernière mise à jour : 27/10/2007

Ce tutoriel vous propose d'aborder pas à pas les principes de base de PowerShell par la mise en #uvre d'une tâche d'administration simple.

- 1 - Public concerné
- 2 - Objectif
 - 2-1 - Aperçus sur les providers
 - 2-2 - Parcourir tous les sous-répertoires d'un répertoire donné
 - 2-3 - Exclure ou inclure certain type de fichier
 - 2-3-1 - Principe de base du Pipelining
 - 2-4 - Filtrer les fichiers
 - 2-5 - Retrouver la date du dernier accès d'un fichier
 - 2-5-1 - Une histoire de date
 - 2-6 - PowerShell et Dotnet
 - 2-7 - Filtrer les fichiers accédés depuis une semaine
 - 2-8 - Retrouver le nom complet d'un fichier
- 3 - Résultat
- 4 - Liens

1 - Public concerné

Débutant	Avancé	Confirmé
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Les pré-requis

Si vous ne connaissez pas PowerShell je vous invite à lire cette [introduction](#).

La connaissance des principes de base de la programmation procédurale facilitera la compréhension de ce tutoriel.

Concernant ceux de la POO il est préférable d'en connaître les principes, à savoir, schématiquement, le regroupement de données et de leurs traitements au sein d'une même structure. Le premier chapitre de ce [tutoriel](#) vous permettra d'en saisir les grandes lignes.

La connaissance du Framework .NET n'est pas indispensable mais vous sera nécessaire si vous souhaitez approfondir l'automatisation de tâches sous PowerShell.

Testé sous XP sp2, PowerShell v1.0 et .NET v2.0.50727.

2 - Objectif

Pour cette première approche fixons nous comme objectif la récupération de fichiers ayant une date supérieure à une date donnée, par exemple les fichiers modifiés depuis une semaine.

A cet énoncé nous devons donc effectuer les opérations suivantes :

- parcourir tous les sous-répertoires d'un répertoire donné,
- exclure ou inclure certain type de fichier,
- retrouver la date du dernier accès d'un fichier,
- filtrer les fichiers accédés depuis une semaine,
- retrouver le nom complet du fichier.

Nous avons désormais nos spécifications, simples mais précises.

Note : Sachez que sous PowerShell il est possible de proposer plusieurs solutions pour un problème donné.

2-1 - Aperçus sur les providers

Comme indiqué dans le tutoriel cité, les **providers** proposent un accès à différentes informations sur un mode commun.

Par exemple déplaçons nous dans l'espace de noms des variables d'environnement :

```
PS C:\WINDOWS\system32\windowspowershell\v1.0> Set-Location ENV:
PS Env:\> (get-location).provider
```

Name	Capabilities	Drives
----	-----	-----
Environment	ShouldProcess	{Env}

Revenons dans l'espace de noms du disque C:

```
PS Env:\> Set-Location C:
PS C:\WINDOWS\system32\windowspowershell\v1.0> (Get-Location).Provider
```


Name	Capabilities	Drives
----	-----	-----
FileSystem	Filter, ShouldProcess	{C, G, D, E...}

PowerShell propose des alias de cmdlet facilitant leur prise en main. La liste complète est accessible via l'appel de **Get-Alias** :

```
PS C:\WINDOWS\system32\windowspowershell\v1.0> Get-Alias cd,dir
```

CommandType	Name	Definition
-----	----	-----
Alias	cd	Set-Location
Alias	dir	Get-ChildItem

```
PS C:\WINDOWS\system32\windowpowershell\v1.0> Get-Alias | More
...
```

 Notez l'usage de la virgule pour séparer les valeurs d'un paramètre.

2-2 - Parcourir tous les sous-répertoires d'un répertoire donné

Sous PowerShell le cmdlet, ou commandelet, **Get-Childitem** retrouve les éléments et les éléments enfants gérés par le provider de l'emplacement courant.

Pour retrouver l'emplacement courant on utilise **Get-Location** :

```
PS C:\WINDOWS\system32\windowpowershell\v1.0> Get-Location

Path
----
C:\WINDOWS\system32\windowpowershell\v1.0
```

Pour obtenir tous les fichiers du répertoire courant et de ces sous-répertoires nous utiliserons **Get-Childitem** avec l'option *-recurse*. Nous pouvons interroger l'aide en ligne afin de retrouver les paramètres et options que ce cmdlet met à notre disposition :

```
PS C:\Temp> help Get-ChildItem -detail
```


Sous PowerShell les paramètres sont ordonnés et attendent une valeur alors qu'une option influence le traitement du cmdlet sans pour autant être obligatoire.

Pour faciliter l'écriture et l'exécution de notre recherche déclarons une variable, *\$MesFichiers*, contenant notre suite d'instruction :

```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse
```

Sous PowerShell il n'est pas nécessaire d'allouer les variables avant leur utilisation.

On déclare une variable par la saisie de son nom, *MesFichiers*, préfixé du signe \$, suivi du signe = puis de son contenu. Enfin la saisie d'un retour chariot finalise sa déclaration.

 La touche tabulation permet une auto-complétion de la saisie.

2-3 - Exclure ou inclure certain type de fichier


Pour exclure un ou plusieurs types de fichier, ajoutons cette fois-ci un paramètre *-Exclude* suivi des noms de fichiers :

```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse -exclude *.tmp,*.bak
```

Comme vous avez pu le remarquer les répertoires sont compris dans le résultat de notre recherche.

On doit donc trouver une particularité qui distingue un nom de fichier d'un nom de répertoire. PowerShell étant *basé-objet* nous pouvons interroger les objets manipulés. Pour retrouver une propriété d'un objet on utilise le cmdlet **Get-Member**, regardons ce qu'il nous propose :

```
PS C:\Temp> help Get-Member -detail
```

 Les paramètres peuvent être saisis en abrégé (-det) à partir du moment où ce qui est saisi lève toutes ambiguïtés. Par exemple si un cmdlet propose les paramètres Detail et Debug, la saisie de -de est ambiguë mais pas celle de -det.

On peut donc filtrer l'affichage des membres d'une classe en utilisant le paramètre *-MemberType* :

```
PS C:\Temp> Get-Childitem|Get-Member -MemberType *property
```

Le cmdlet nous renvoi le nom, le type et les propriétés définies dans la classe .NET sous-jacente correspondant aux objets manipulés.

Get-Member traite chaque occurrence des classes manipulées, à savoir les classes **DirectoryInfo** et **FileInfo** :

```

    TypeName: System.IO.DirectoryInfo
    ...
    TypeName: System.IO.FileInfo

Name                MemberType      Definition
----                -
PSChildName         NoteProperty    System.String PSChildName=certificate.format.pslxml
...
Attributes          Property        System.IO.FileAttributes Attributes {get;set;}
...
Mode                ScriptProperty  System.Object Mode {get=$catr = "";...
Name                Property        System.String Name {get;}
    
```

Remarquez que certaines propriétés sont en lecture seule **{get;}**.

Les propriétés de type *NoteProperty*, de la classe **FileInfo**, jouent un rôle particulier, en dehors du fait qu'elle commence toutes par PS, elles sont ajoutées par le provider pour faciliter les traitements.

La propriété **PSIsContainer**, de type booléen, détermine si l'instance, renvoyée par **Get-Childitem**, est un conteneur ou pas. Cette propriété se retrouve pour chaque élément fournis par le provider manipulant un espace de noms. Ce qui est confirmé par les instructions suivante :

```
PS C:\Temp> cd HKCU:;Get-Childitem|Get-Member -membertype *property;C:
```

 Notez que le point virgule sépare les instructions.


Pour en revenir à notre propriété, filtrons les fichiers en l'utilisant :

```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse -exclude *.tmp,*.bak|Where-Object {$_.PSIsContainer  
-eq 0}
```

Where-Object, ou l'alias **Where** ou encore le raccourcis **%**, filtre, sur une ou plusieurs propriétés, les objets reçus via le pipe.

Les caractères **\$_** représentent l'occurrence courante renvoyée par **Get-ChildItem**. Le code entre accolade est appelé un bloc de code (*scriptblock*).

Pour rechercher un objet répondant à un critère particulier utilisons **Where-Object**.

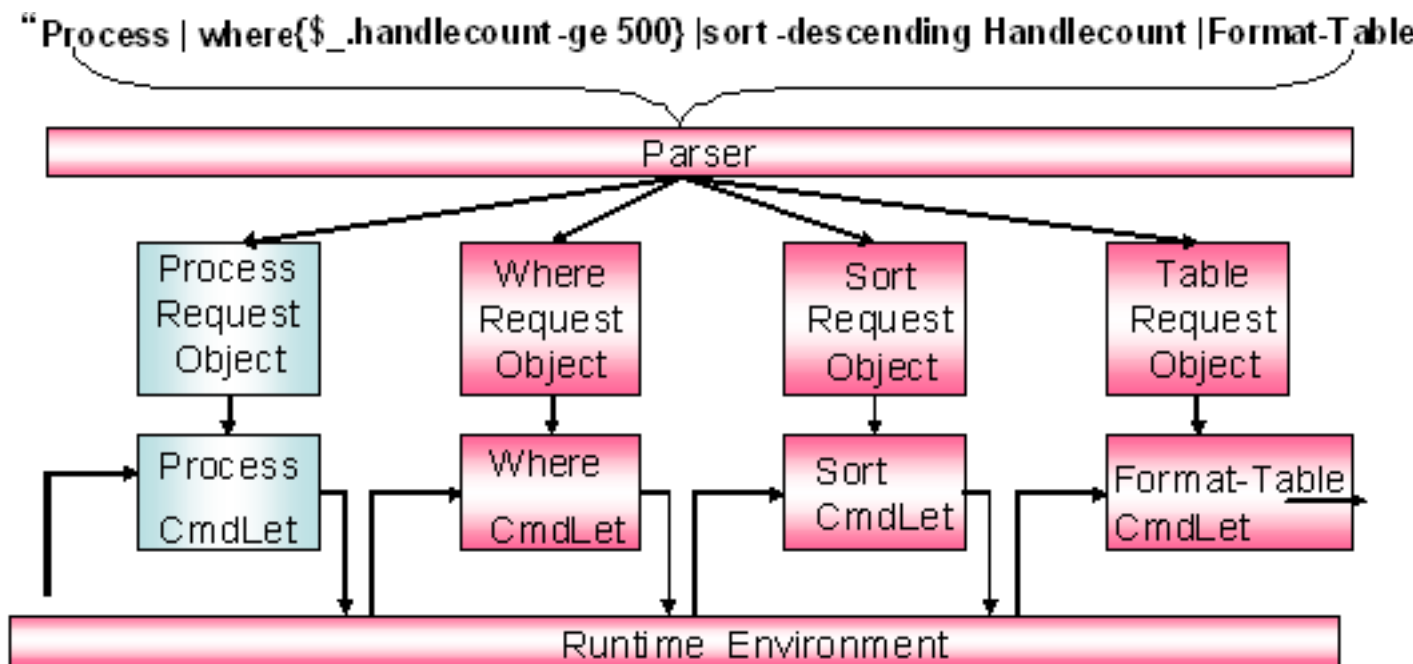
 *Pour plus d'informations consultez les fichiers texte `about_xxxx`, présent dans le répertoire d'installation par défaut de PowerShell :
C:WINDOWS\system32\windowspowershell\v1.0\fr*

*Voir également dans la rubrique liens l'outil **PowerShellDocumentationPack**.*

2-3-1 - Principe de base du Pipelining

Comme nous le dit la documentation fournie (C:WINDOWS\system32\windowspowershell\v1.0\fr\userguide.rtf) :

```
...  
Un pipeline agit comme une série de segments de canal connectés. Les éléments qui parcourent le  
pipeline passent par chaque segment.  
Pour créer un pipeline dans Windows PowerShell, vous connectez des commandes au moyen de  
l'opérateur de pipeline  
(la barre verticale " | ") et la sortie de chaque commande est utilisée comme entrée de la  
suivante.  
...
```



The Monad Automation Model (cliquez sur l'image pour accéder au document)

PowerShell est construit de telle manière que les données, c'est à dire des objets, transitent d'une commande à une autre sans être transformées au format texte. C'est seulement quand le texte est nécessaire qu'une conversion de l'objet est faite.

2-4 - Filtrer les fichiers

Reprenons l'objectif en cours, retrouver le nom complet du fichier. Le résultat de la commande précédente n'est pas très exploitable ainsi. Réutilisons la commande **Get-Member** en sélectionnant uniquement les propriétés (property), sans connaître le Framework dotnet on peut sans trop se tromper se dire que la propriété *FullName* répond à notre besoin. Par prudence la consultation de la documentation nous confirmera son contenu.

Vérifions le en ajoutant un second filtre :


```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse -exclude *.tmp,*.bak|where-object {$_.PSIsContainer -eq 0}|foreach {$_.Get_FullName() }
```

i Les objets sont injectés dans le pipeline l'un après l'autre une fois le traitement effectué, et non pas une fois le traitement terminé sur l'ensemble des données, comme le pipe sous MSDOS ou le mode console sous NT.

2-5 - Retrouver la date du dernier accès d'un fichier

L'étape suivante consiste à retrouver la date du dernier accès d'un fichier, il s'agit de la propriété de la classe **FileInfo** nommée *LastWriteTime*, ajoutons la en paramètre au dernier filtre :

```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse -exclude *.tmp,*.bak|where-object {$_.PSIsContainer
-eq 0}|`
foreach {$_.get_Fullname(), $_.LastWriteTime}
```

 L'apostrophe inverse (Alt Gr+7) indique que l'instruction continue sur la ligne suivante.

2-5-1 - Une histoire de date

Il nous reste enfin à filtrer les fichiers que l'on a modifiés depuis une semaine. Pour obtenir la date du jour saisissons :

```
PS C:\Temp> Date
```

Essai concluant. Comme **Date** ne semble à priori pas être un appel de cmdlet, recherchons à quoi il peut correspondre. Vérifions dans l'espace de nom des alias :

```
PS C:\Temp> Dir Alias:|Sort
```

pas de trace d'un quelconque alias *Date*, essayons dans l'espace de noms des fonctions :

```
PS C:\Temp> Dir Functions:|Sort
```

toujours pas, essayons dans celui des variables :

```
PS C:\Temp> Dir Variable:|Sort
```

encore moins. Sortons la pelle et la pioche pour creuser ce détail :

```
PS C:\Temp> Set-psDebug -trace 2 -step
```

comme nous le dit la documentation en ligne, **Set-psDebug**

```
Active et désactive les fonctions de débogage, définit le niveau de suivi et active/désactive le
mode strict.
```

Si on valide désormais la saisie de **Date**, nous obtenons le message suivant :

```
PS C:\Temp> Voulez-vous continuer cette opération ?
1+ date
[O] Oui [T] Oui pour tout [N] Non [U] Non pour tout [S] Suspendre [?] Aide (la valeur par
défaut est « O ») :
```

Répondez 'T', le résultat renvoyé est :

```
DÉBOGUER : 1+ date
```

```
DÉBOGUER : 1+ if ($this.DisplayHint -ieq "Date")
DÉBOGUER : 11+ "{0} {1}" -f $this.ToLongDateString(),
$this.ToLongTimeString()
DÉBOGUER : ! CALL method 'System.String ToLongDateString()'
DÉBOGUER : ! CALL method 'System.String ToLongTimeString()'
```

Il s'agit ici d'un appel aux extensions du système de type de PowerShell, bon comme nous avons la réponse à notre question nous nous arrêterons là.

Rangeons la pelle et la pioche, jusqu'à la prochaine fois ;-)

```
PS C:\Temp> Set-psDebug -trace 0
```

Après ce petit intermède revenons au dernier point, filtrer les fichiers que l'on a modifiés depuis une semaine.

2-6 - PowerShell et Dotnet

Comme vous prenez goût aux tours et détours sous PowerShell, allons faire un tour cette fois-ci du côté de chez dotNET.

Maîtriser Powershell ne peut se faire qu'au travers de la connaissance des fonctionnalités de ce framework.

Pour retrouver notre informations nous utiliserons la classe **DateTime**. Inspectons cette classe comme nous l'avons déjà fait :

```
PS C:\Temp> DateTime|gm
```

On obtient l'erreur :

```
Le terme « datetime » n'est pas reconnu en tant qu'applet de commande, fonction, programme
exécutable ou fichier de script.
Vérifiez le terme et réessayez.
```

Bon ce n'est pas ça, essayons avec **Get-Date** :

```
PS C:\Temp> Get-Date|gm
```

C'est déjà mieux. Ah oui j'ai oublié de vous dire que le cmdlet **Get-Date** existait ;-)

```
PS C:\Temp> Get-Command *Date*
```

Récupérons maintenant la date du jour mais en utilisant la méthode statique **Now**, **Get-Date** renvoi une instance de la classe **DateTime**. L'affichage de **Get-Member** ne signalant pas par défaut les méthodes statiques, ajoutons l'option **-static** :

```
PS C:\Temp> [DateTime]|gm -static
```

Comme vous le voyez pour signaler l'accès aux informations d'une classe il faut placer son nom entre crochet.

Appelons la méthode statique **Now** :

```
PS C:\Temp> [DateTime].Now()
```

mais cet appel nous renvoi :

```
L'appel de la méthode a échoué parce que [System.RuntimeType] ne contient pas de méthode nommée  
« Now ».
```

Les concepteurs du langage ont préférés une syntaxe différente pour les appels des méthodes statiques, à la place du point on double le caractère 2 points :

```
PS C:\Temp> [DateTime]::Now  
samedi 12 mai 2007 14:09:22
```

A y regarder de plus près le résultat ne convient pas car l'heure actuelle faussera les résultats. Essayons la méthode **Today** :

```
PS C:\Temp> [DateTime]::Today
```

C'est parfait car l'heure est bien à 00.00.00, l'heure d'exécution de notre script n'entrera pas en ligne de compte dans le résultat final seulement la date. Dernier point, retrouver une date dans le passé. A premier vue, en consultant la documentation .NET, nous avons la méthode **AddDays** :

```
PS C:\Temp> [DateTime]::Today.AddDays(7)
```


nous venons de découvrir le script qui manipule les fichiers de la semaine prochaine, d'un intérêt pratique quasi nul dans notre contexte, je vous l'accorde ;-)

Au cas où, vérifions la documentation de la classe **DateTime** :

```
value  
A number of whole and fractional days. The value parameter can be negative or positive.
```

Bonne pioche !

```
PS C:\Temp> [DateTime]::Today.AddDays(-7)
```

 Si vous voulez modifier une propriété d'un fichier utilisez le cmdlet **Set-ItemProperty** :

```
PS C:\Temp> Set-ItemProperty -path FileSystem::C:\Temp\Test.txt -name LastWriteTime -value  
([DateTime]::Now)
```

2-7 - Filtrer les fichiers accédés depuis une semaine

En regardant le résultat de **Get-Member** sur la classe **DateTime**, l'affichage des informations de la méthode **Today** indique que le type du résultat qu'elle renvoi est du même type que la classe qu'on manipule, c'est à dire **DateTime** :

```
Todays Property System.DateTime Now {get;}
```

Dans ce cas on peut donc appeler plusieurs méthodes sans avoir à utiliser de variable intermédiaire :

```
PS C:\Temp> [DateTime]::Today.AddDays(-7)
```

Il nous reste à réorganiser notre expression :

```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse -exclude *.tmp,*.bak|where-object {$_.PSIsContainer  
-eq 0}|  
where {$_.LastWriteTime -ge [DateTime]::today.adddays(-7)}
```

Pensez à vous placer dans un répertoire contenant au moins un fichier modifié il y a une semaine.

La validation de cette instruction, par entrée, effectue le traitement mais n'affiche pas le résultat car il est mémorisé dans la variable **\$MesFichiers**.

```
PS C:\Temp> $MesFichiers  
  
Répertoire : Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\Debug\UserMode  
  
Mode                LastWriteTime         Length Name  
----                -  
-a---             07/05/2007   19:00      183776 userenv.log  
...
```

Cette variable contient une collection d'objet, ce que confirme l'appel de la méthode **GetType** (fournie par la classe **Object**) :

```
PS C:\Temp> $MesFichiers.GetType().FullName  
System.Object[]
```


Chaque élément étant du type **FileInfo** :

```
PS C:\Temp> $MesFichiers[0].GetType().FullName  
System.IO.FileInfo
```

On peut aussi retrouver le nombre d'élément de cette collection :

```
PS C:\Temp> $MesFichiers.Length
```

Rappelez vous que la variable `$MesFichiers` contient un résultat correct au moment de son exécution mais le système de fichier d'un serveur évolue rapidement.

 Sous `dotNET` la convention d'écriture signale un type tableau par la présence de 2 crochets `[]` :

```
System.Object[] # Tableau d'objets
```

2-8 - Retrouver le nom complet d'un fichier

Si on souhaite récupérer uniquement le nom complet du fichier, par exemple pour passer cette liste à un outil tiers, on utilisera la méthode **`Get_FullName`** de la classe **`FileInfo`** :

```
PS C:\Temp> Dir|ForEach {$_ .Get_Fullname}
```

Ici il est nécessaire de préciser l'appel d'une méthode par l'ajout, après le nom de la méthode, de 2 parenthèses :

```
PS C:\Temp> Dir|ForEach {$_ .Get_Fullname() }
```

car dans le premier cas l'analyse lexicale de PowerShell considère l'instruction comme une recherche d'informations sur la méthode de la classe alors que dans le second elle la considère comme un appel à la méthode.

Il reste possible de mémoriser cette liste dans un fichier texte via le cmdlet **`Out-File`** qui se retrouvera toujours en dernière position tout simplement parce qu'il ne gère pas le pipelining :

```
PS C:\Temp> Out-File C:\Temp\Result.txt
```

 Le cmdlet **`Export-CSV`**, couplé à **`Select-Object`**, permet de récupérer plusieurs propriétés d'un objet au format `csv`.


3 - Résultat

Notre ligne de commande devenant donc :

```
PS C:\Temp> $MesFichiers=Get-ChildItem -recurse -exclude *.tmp,*.bak|where {$_.PSIsContainer -eq 0}|`
where {$_.LastWriteTime -ge [DateTime]::Today.AddDays(-7)}|ForEach {$_.Get_Fullname()}|Out-File C:\Temp\Result.txt
```

ou mieux car on a ainsi une collection et un fichier :

```
PS C:\Temp> $MesFichiers|Out-File C:\Temp\Result.txt
```

 *Si vous effectuez une recherche sur un nombre important de fichier, la taille mémoire de la collection de fichiers peut poser problème.*

Il reste un dernier point à régler car après la première exécution, la variable contient le résultat d'une suite d'instructions, il n'est dès lors plus possible de ré-exécuter cette suite d'instructions à partir du nom de variable .

On peut très bien rappeler la ligne de commande complète ou utiliser un script qui renverrait une collection de nom de fichiers. Je vous propose d'utiliser une autre approche au travers d'un bloc de script (*ScriptBlock*), c'est à dire de placer les instructions entre accolades {...} :

```
PS C:\Temp> $MesFichiers={Get-ChildItem -recurse -exclude *.tmp,*.bak|where {$_.PSIsContainer -eq 0}|`
where {$_.LastWriteTime -ge [DateTime]::Today.AddDays(-7)}|ForEach {$_.Get_Fullname()}}
```

Le type de la variable est logiquement modifiée, car elle contient maintenant du code :

```
PS C:\Temp> $MesFichiers.GetType()


IsPublic IsSerial Name                                     BaseType
-----
True     False     ScriptBlock                                             System.Object
```

L'exécution de **\$MesFichiers** provoquera simplement l'affichage de son contenu, pour l'exécuter on préfixera le nom de la variable par le caractère **&** :

```
PS C:\Temp> &$MesFichiers
```

Pour récupérer la collection l'usage d'une seconde variable reste possible :

```
PS C:\Temp> $CollectionFichiers=&$MesFichiers
```

 *les deux instructions **Where** portant sur des propriétés différentes d'un même objet peuvent être concaténées :*

```
where { ( $_.PSIsContainer -eq 0) -and ( $_.LastWriteTime -ge [DateTime]::Today.AddDays(-7) ) }
```

A vous de jouer maintenant !

4 - Liens

Doc Microsoft:

 [Grammaire de PowerShell.](#)


 [Site dédié.](#)

 [WMI et PowerShell.](#)

Divers :


 [Webcasts sur le langage PowerShell](#)

 [Installation de PowerShell sous Vista](#)

 [An Introduction To Microsoft PowerShell](#), E-book gratuit mais avec un enregistrement nécessaire sur le site.

 [Rappel des commandes.](#)

 [PowerShell Analyzer RC1 environnement interactif pour Windows PowerShell.](#)

 [PowerShellDocumentationPack](#), lien (Free Tools). Facilite la navigation de l'aide en ligne des cmdlets.

Cmdlets dédiés à la  [gestion réseaux et messagerie.](#)

 [Webcast TechNet Live : Windows PowerShell](#)

Ce webcast TechNet a été animé par Antoine Habert et Cédric Bravo (co-auteurs de "[Scripting Windows](#)").

Tutoriel vidéo des cmdlets payant de  [Powergadgets.](#)

Filtrage et formatage de données

D'autres ressources sur le forum  **Programmation Windows.**

Bonnes lectures :-)

