**Introduction à PowerShell**

**Documentation sur internet:**

PowerShell Documentation https://docs.microsoft.com/en-us/powershell

PowerShell Gallery https://www.powershellgallery.com

**Introduction à PowerShell**

"PowerShell" est constitué de plusieurs applets de commande (cmdlet).

**Un cmdlet est constitué d'un verbe (VERB) suivi d'un nom (NOUN).**

Pour afficher la liste des cmdlet. -> Get-Command -CommandType cmdlet

Pour afficher le nombre de cmdlet. -> (Get-Command -CommandType cmdlet).Count

Pour afficher la liste des cmdlet qui sont dans un module particulier

* Get-Command -Module Hyper-V -CommandType cmdlet

Pour afficher le nom du module qui contient un cmdlet spécifique

* (Get-Command -Name Get-VM).ModuleName

Pour afficher la liste des cmdlets qui contiennent un nom

* Get-Command -Name \*service\* -CommandType Cmdlet
* Get-Command -Name \*process\* -CommandType Cmdlet

Pour afficher la liste des cmdlets, fonctions, workflows et alias qui contiennent le nom d'un objet

* Get-Command -Noun \*service\*
* Get-Command -Noun \*IPv4\*

Pour afficher la liste des verbes utilisés par Windows PowerShell -> Get-Verb

Pour afficher la liste des cmdlet qui effectue une action particulière

* Get-Command -Verb convert\* # Pour inclure les trois verbes: Convert, ConvertFrom, ConvertTo
* Get-Command -Verb get
* Get-Command -Verb out
* Get-Command -Verb write

**Effectuer des calculs avec PowerShell**

3 + 2 # le résultat est 5

3 – 2 # le résultat est 1

3 \* 2 # le résultat est 6

3 / 2 # le résultat est 1.5

2 / 3 # le résultat est 0,666666666666667

5%4 # le résultat est 1

# % est l'opérateur MODULO qui calcule le reste de la division

"-" \* 80 # affiche 80 tirets

1kb # affiche 1024

1mb # affiche 1048576

1gb # affiche 1073741824

1tb # affiche 1099511627776

1pb # affiche 1125899906842624

1gb / 1mb # le résultat est 1024

1tb / 1gb # le résultat est 1024

1pb / 1gb # le résultat est 1048576

2gb \* 5 # le résultat est 10737418240

0xffff # affiche 65535

16 + 0x10 # le résultat est 32

**Utilisation du cmdlet Get-Help**

Dans PowerShell, on peut afficher de l'aide sur plus d'une centaine de sujets.

La commande suivante permet d'afficher la liste complète des sujets:

**Get-Help about**

**# Commande pour afficher en ordre alphabétique la liste complète des sujets**

**$info = (Get-Help about).Name | Sort-Object -Unique**

**$info**

**"-" \* 80**

**$total = $info.Count**

**Write-Host "Nombre de sujet = $total" -ForegroundColor Yellow**

**"-" \* 80**

Voici une liste de plusieurs sujets intéressants:

about\_Arithmetic\_Operators

about\_Assignment\_Operators

about\_Comparison\_Operators

about\_Logical\_Operators

about\_Operator\_Precedence

about\_Operators

about\_Type\_Operators

about\_Do

about\_For

about\_ForEach

about\_If

about\_Switch

about\_While

about\_Arrays

about\_Functions

about\_Functions\_Advanced

about\_Functions\_Advanced\_Methods

about\_Functions\_Advanced\_Parameters

about\_Functions\_CmdletBindingAttribute

about\_Functions\_OutputTypeAttribute

about\_Hash\_Tables

about\_Ref

about\_Regular\_Expressions

about\_Scopes

about\_Script\_Blocks

about\_Scripts

about\_Try\_Catch\_Finally

about\_Variables

**Mise à jour de l'aide dans PowerShell**

Dans le cas d’un message d’erreur à cause de la langue vous pouvez tenter la commande suivante:

**Update-help -UIculture en-US -Force**

Le chargement de l’aide peut prendre de quelques secondes à une ou deux minutes.

On peut utiliser le cmdlet "**get-help**" avec le paramètre "**-online**".

exemple: get-help get-vm -online

**Utilisation des ALIAS**

Un alias remplace le nom d'un cmdlet par un nom très court.

Le cmdlet "**Get-Alias**" permet d'afficher les alias.

**Il n'est pas recommandé d'utiliser les alias dans des scripts parce qu'il peuvent porter à confusion.**

Les alias peuvent être difficiles à comprendre en particulier pour les programmeurs débutants.

Le code est plus difficile à maintenir en particulier pour un autre programmeur que l’auteur du script.

Exemple d'alias facile à comprendre

**clear** **Clear-Host**

**cp** **Copy-Item**

Plusieurs alias pour le même cmdlet

**cd Set-Location**

**chdir Set-Location**

**copy** **Copy-Item**

**cp** **Copy-Item**

**cpi** **Copy-Item**

Exemple d'alias difficile à comprendre

**%** **ForEach-Object**

**?** **Where-Object**

**Introduction à la programmation PowerShell**

**Les opérateurs arithmétiques** -> get-help about\_Arithmetic\_Operators

**Les opérateurs de comparaison** -> get-help about\_Comparison\_Operators

**Les opérateurs logiques** -> get-help about\_Logical\_Operators

Voici les commandes pour obtenir de l'aide sur les instructions:

* IF get-help about\_if
* FOR get-help about\_for
* FOREACH get-help about\_foreach
* SWITCH get-help about\_switch
* WHILE get-help about\_while

**L'opérateur IF est utilisé pour tester des conditions**

**if ($a -gt 2)**

**{**

**Write-Host "La valeur $a est plus grande que 2."**

**}**

**if ($a -gt 2)**

**{**

**Write-Host "La valeur $a est plus grande que 2."**

**}**

**else**

**{**

**Write-Host ("La valeur $a est plus petite ou égale à 2," +**

**" ou n'existe pas ou n'est pas initialisée.")**

**}**

**if ($a -gt 2)**

**{**

**Write-Host " La valeur $a est plus grande que 2."**

**}**

**elseif ($a -eq 2)**

**{**

**Write-Host " La valeur $a est égale à 2."**

**}**

**else**

**{**

**Write-Host ("La valeur $a est plus petite que 2," +**

**" ou n'existe pas ou n'est pas initialisée.")**

**}**

**L'opérateur FOR est utilisé pour effectuer une boucle**

Une boucle FOR s'exécute en utilisant une valeur de départ, un test et un incrément.

**for($i=1; $i -le 10; $i++)**

**{**

**Write-Host $i** # La boucle s'exécute 10 fois et affiche les valeurs 1,2,3,4,5,6,7,8,9,10

**}**

**for($i=10; $i -ge 1; $i--)**

**{**

**Write-Host $i** # La boucle s'exécute 10 fois et affiche les valeurs 10,9,8,7,6,5,4,3,2,1

**}**

**L'opérateur FOREACH est utilisé pour parcourir tous les éléments dans une collection d'objets**

Une boucle FOREACH exécute une itération à partir des valeurs d'une collection.

**$lettres = "a","b","c","d"**

**foreach ($lettre in $lettres)**

**{**

**Write-Host $lettre**

**}**

**L'opérateur SWITCH est utilisé pour tester des conditions**

**$i = 3**

**switch ($i)**

**{**

**1 {"La valeur est un."}**

**2 {"La valeur est deux."}**

**3 {"La valeur est trois."}**

**4 {"La valeur est quatre."}**

**}**

**La valeur est trois.**

**$i = 3**

**switch ($i)**

**{**

**1 {"La valeur est un."}**

**2 {"La valeur est deux."}**

**3 {"La valeur est trois."}**

**4 {"La valeur est quatre."}**

**3 {"Encore trois."}**

**}**

**La valeur est trois.**

**Encore trois.**

**# Break permet d'arrêter immédiatement**

**$i = 3**

**switch ($i)**

**{**

**1 {"La valeur est un."}**

**2 {"La valeur est deux."}**

**3 {"La valeur est trois."; Break}**

**4 {"La valeur est quatre."}**

**3 {"Encore trois."}**

**}**

**La valeur est trois.**

# La commande SWITCH teste deux valeurs

**switch (4,2)**

**{**

**1 {"La valeur est un."}**

**2 {"La valeur est deux."}**

**3 {"La valeur est trois."; Break}**

**4 {"La valeur est quatre."}**

**3 {"Encore trois."}**

**}**

**La valeur est quatre.**

**La valeur est deux.**

# Default est utilisé si aucun teste fonctionne

**$i = 5**

**switch ($i)**

**{**

**1 {"La valeur est un."; Break }**

**2 {"La valeur est deux."; Break }**

**3 {"La valeur est trois."; Break}**

**4 {"La valeur est quatre."; Break }**

**Default {"Aucune valeur."}**

**}**

**Aucune valeur.**

**L'opérateur WHILE est utilisé pour effectuer une boucle**

Il ne faut pas oublier d'incrémenter la valeur de la variable dans la boucle WHILE.

**$val =1**

**while($val -le 3)**

**{**

**Write-Host $val**

**$val++**

**}**

**Les variables**

Une variable débute avec $

1. Une variable peut contenir le résultat d'une commande Windows

$resultat = ping 10.57.22.100

$resultat

1. Une variable peut contenir les propriétés d'un objet.

$col = Get-CimInstance -ClassName win32\_processor

$col

$col.NumberOfCores

$col.NumberOfLogicalProcessors

1. Normalement, une variable est en mémoire mais PowerShell permet qu'une variable soit un fichier.

${C:\Temp\Test.txt} = "Test pour écrire dans un fichier."

${C:\Temp\Test.txt} += "`n" + "Ligne 2 !!!"

**Pour afficher la liste des variables d'environnement**

* Get-ChildItem env:

**Pour afficher le contenu d'une variable d'environnement**

On doit ajouter **$env:** devant le nom de la variable d'environnement de Windows

* $env:computername

**Pour afficher la liste des variables** -> Get-Variable

**Pour créer une nouvelle variable, variable en lecture seule ou constante**

* New-Variable -Name pi -Value ([system.math]::PI) -Option Constant

**Pour effacer le contenu d'une variable** -> Clear-Variable -Name resultat

**Pour supprimer une variable et son contenu** -> Remove-Variable -Name resultat

**La variable $?**

Le contenu de la variable **$?** indique si l'exécution de la dernière commande a réussie ou échouée.

**Les variables booléennes**

* $true
* $false

**Concaténation des chaînes de caractères**

**Guillemet simple**

**La substitution de la variable $nombre ne fonctionne pas**

$nombre = 5

'Nombre = $nombre'

résultat: Nombre = $nombre

**Guillemet double**

**La substitution de la variable $nombre fonctionne**

$nombre = 5

"Nombre = $nombre"

résultat: Nombre = 5

**L'opérateur + permet de concaténer des chaînes de caractères**

$c = 'abc' + 'xyz'

$c

résultat: abcxyz

**L'opérateur + permet de concaténer des variables**

$c1 = 'abc'

$c2 = 'xyz'

$c3 = $c1 + $c2

$c3

résultat: abcxyz

**Les tableaux**

Un tableau est une variable qui contient plusieurs valeurs.

Chaque valeur est séparée par une virgule.

$tab1 = 1,2,3,4,5,6,7,8,9,10

$tab2 = 1..10 # l'opérateur .. permet de générer plusieurs valeurs

$tab3 = "python","java","PowerShell"

$tab4 = Get-Service # le résultat de Get-Service est un tableau

**Manipuler un tableau**

Le premier élément d'un tableau commence à la position d'index [0].

$tab1[0] # affiche le premier élément du tableau $tab1

$tab1[4] # affiche le cinquième élément du tableau $tab1

$tab1[-1] # affiche le dernier élément du tableau $tab1

$tab1[-4..-1] # affiche les quatre derniers éléments du tableau $tab1

Get-Member -InputObject $tab1 # affiche les méthodes disponibles pour le tableau $tab1

$tab1.SetValue(500,5) # change la valeur de l'index 5

# le nouveau contenu de $tab1 est 1,2,3,4,5,500,7,8,9,10

$tab1[5]=6 # le nouveau contenu de $tab1 est 1,2,3,4,5,500,7,8,9,10

$tab1.Contains(5) # vérifie si la valeur 5 est présente dans le tableau $tab1

$tab1.Length # affiche le nombre de valeurs dans le tableau $tab1

**Les dictionnaires (hash table)**

Un dictionnaire est une structure de données qui consiste à associer des paires [ clé = valeur ].

Un dictionnaire débute par le signe @.

Il est important de savoir que l'ordre d'affichage des éléments ne correspond pas à celui de la définition du dictionnaire (Hash Table).

**Utilisation d'une variable "hash table"**

**# Déclaration d'une "hash table" sur une ligne**

$var = @{ "cd"="ordi1";"routeur"="ordi6" }

$var.cd

$var.routeur

**Déclaration d'une "hash table" sur plusieurs lignes**

$var2 = @{ "cd"="ordi1"

"routeur"="ordi6" }

$var2.cd

$var.routeur

**Trier une "hash table"**

$hash = @{a = 1; b = 2; c = 3; d = 4; e = 5; f = 6}

foreach ($h in $hash.GetEnumerator() | Sort-Object Key)

{

Write-Host "Nom= $($h.Key) Valeur= $($h.Value)" -ForegroundColor Green

}

**Créer une "hash table" en forçant le respect de l'ordre**

$hash = **[ordered]**@{a = 1; b = 2; c = 3; d = 4; e = 5; f = 6}

foreach ($h in $hash.GetEnumerator())

{

Write-Host "Nom= $($h.Key) Valeur= $($h.Value)" -ForegroundColor Green

}

GetEnumerator() # permet de récupérer chaque combinaison clé/valeur

$hash.keys # liste les clés du dictionnaire $hash

$hask.values # liste les valeurs du dictionnaire $hash

$hash.a # récupère la valeur de la clé "**a**"

$hash["a"] # récupère la valeur de la clé "**a**"

$hash["a","d"] # récupère la valeur de la clé "**a**" et de la clé "**d**"

Get-Member -InputObject $hash # affiche les méthodes disponibles pour le dictionnaire $hash

$hash.Add("g",7) # ajoute une paire au dictionnaire $hash

$hash.Remove("a") # supprime une paire au dictionnaire $hash

$hash.Clear() # efface le contenu

$ageList = @{} # création d'une "Hast Table" vide ou efface le contenu

**Exemples**

**Voici un "Hash Table" qui contient plusieurs variables.**

$messages = @{

# Les erreurs pour la variable "PATHS".

MSG\_CHECKING\_PATHS = "Checking paths...";

MSG\_COMPUTING\_PATHS = "Computing paths...";

MSG\_CREATING\_PATHS = "Creating paths...";

# Les erreurs pour la variable "IMAGE".

MSG\_COPYING\_IMAGE = "Copying image...";

MSG\_MOUNTING\_IMAGE = "Mounting image...";

MSG\_CONVERTING\_IMAGE = "Converting image...";

MSG\_SKIPPING\_IMAGE\_CONVERSION = "Skipping image conversion."

}

# Voici deux exemples qui affichent des messages qui sont dans le "hash table"

Write-Host $messages.MSG\_CHECKING\_PATHS

Write-Host $messages.MSG\_MOUNTING\_IMAGE

**Utilisation d'une "hash table" pour initialiser les paramètres d'un cmdlet**

$params = @{ Path = "C:\\_OUTILS"; Filter = "\*" }

Get-Item @params

Pour utiliser **Sort-Object** avec plusieurs paramètres, il faut utiliser des "hash table" pour trier par ordre croissant, décroissant, ou une combinaison d’ordres de tri.

Get-Service | Sort-Object -Property @{Expression = "Status"; Descending = $true},

@{Expression = "DisplayName"; Ascending = $true}

**Les scripts avec PowerShell**

Les fichiers de script PowerShell doivent avoir l'extension PS1.

Par défaut, on ne peut pas exécuter des scripts avec PowerShell.

La commande **Get-ExecutionPolicy** permet d'obtenir la valeur de la stratégie d'exécution actuelle.

Pour changer la stratégie d'exécution on utilise l'instruction **Set-ExecutionPolicy**.

* Set-ExecutionPolicy -ExecutionPolicy Restricted
* Restricted: n'exécute pas de scripts, c'est la valeur par défaut
* Set-ExecutionPolicy -ExecutionPolicy Unrestricted
* Unrestricted: charge tous les fichiers de configuration et exécute tous les scripts. Si vous exécutez un script non signé à partir du réseau, vous êtes invité à autoriser l'exécution de ce script.
* Set-ExecutionPolicy -ExecutionPolicy Bypass
* Bypass: charge tous les fichiers de configuration et exécute tous les scripts.

**IMPORTANT: on ne peut pas exécuter un script PowerShell en double cliquant sur le fichier.**

Pour exécuter un script PowerShell, il faut spécifier le chemin absolu ou le chemin relatif lors de l'appel.

**Les commentaires**

**# Cette ligne est en commentaire à cause du symbole #.**

**<#**

**Ceci est un commentaire**

**sur plusieurs lignes.**

**#>**

**Caractère de continuité**

Il est possible d'écrire une ligne d'instruction sur plusieurs lignes en utilisant un caractère de continuité.

Le caractère de continuité correspond à l'accent de grave (code ASCII 96).

**L'utilisation de la variable $PSItem lorsqu'on utilise un "PIPE"**

note: "PowerShell 3.0" remplace **$\_** par **$PSItem**

PSItem contient la valeur courante d'une commande "PIPE".

# Syntaxe avant "PowerShell 3.0"

Get-Service | Where-Object { $\_.Status -eq "Stopped" }

# Syntaxe standard avec "PowerShell 3.0"

Get-Service | Where-Object { $PSItem.Status -eq "Stopped" }

# Si on utilise un seul paramètre, on peut utiliser la syntaxe simplifiée

Get-Service | Where-Object Status -eq "Stopped"

# Si on utilise deux paramètres, on ne peut pas utiliser la syntaxe simplifiée

Get-Service | Where-Object `

{ $PSItem.Status -eq "Stopped" -and $PSItem.DisplayName -like "\*Windows\*" }

**# Exemple intéressant**

**# La variable $serveurs va contenir: HV01,HV02,HV03,HV04,HV05,HV06,HV07,HV08,HV09,HV10**

$serveurs = 1..10 | ForEach-Object { "HV{0:D2}" -f $PSItem }

**Out-Host force l'affichage du résultat à l'écran**

Lors de l'exécution de plusieurs commandes, il arrive que le résultat des deux commandes soit fusionné.

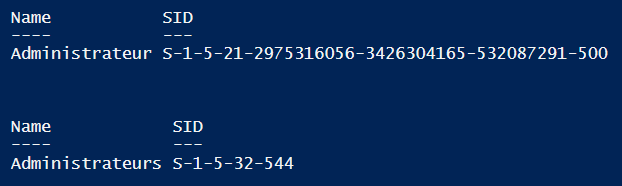
**Get-LocalUser -Name Administrateur | Select-Object Name,SID**

**Get-LocalGroup -Name Administrateurs | Select-Object Name,SID**

Le résultat des deux commandes est simplement les deux colonnes Name et SID.

**Get-LocalUser -Name Administrateur | Select-Object Name,SID | Out-Host**

**Get-LocalGroup -Name Administrateurs | Select-Object Name,SID**



**Out-Host** force l'affichage du résultat de la première commande.

Le résultat de la deuxième commande s'affiche à la suite du résultat de la première commande.

**Exemple de code**

**Voici le code qui affiche le nom de la carte réseau et sa vitesse de transmission.**

**note: le nom de la carte réseau doit être le même sur chaque serveur**

**Clear-Host**

**$carte = "Ethernet"**

**# La variable $serveurs va contenir: HV01,HV02,HV03,HV04,HV05,HV06,HV07,HV08,HV09,HV10**

**$serveurs = 1..10 | ForEach-Object { "HV{0:D2}" -f $PSItem }**

**foreach ($serveur in $serveurs)**

**{**

**Write-Host $serveur -ForegroundColor Yellow**

**Get-NetAdapter -Name $carte -CimSession $serveur | Format-Table Name,LinkSpeed**

**"-"\*100**

**}**

**Détails sur les boucles**

1..10 | ForEach-Object { "HV{0:D2}" -f $PSItem }

# Dans cet exemple ForEach est l'alias de ForEach-Object

1..10 | ForEach { "HV{0:D2}" -f $PSItem }

# Dans cet exemple, foreach est une méthode de la collection

(1..10).foreach({"HV{0:D2}" -f $PSItem})

**Exemple de code**

**Voici le code qui affiche la liste complète des fonctions, cmdlet, alias de tous les modules disponibles.**

**Clear-Host**

**$modules = (Get-Module -ListAvailable).Name**

**foreach ($module in $modules)**

**{**

**Write-Host "Nom du module: $module" -ForegroundColor Green**

**Get-Command -All -Module $module**

**"\*" \* 80**

**}**

**Utilisation d'un workflow et d'une boucle FOREACH et du paramètre -Parallel**

**Ce script envoie la commande Restart-Computer en parallèle aux ordinateurs.**

**Ce script n'est pas ralenti par le fait qu'il peut y avoir des ordinateurs qui sont fermés.**

**# La variable $computers va contenir les noms 407P01 à 407P32**

**$computers = 1..32 | ForEach-Object { "407P{0:D2}" -f $PSItem }**

**Workflow Restart-AllComputers**

**{**

**param([string[]]$Computers)**

**ForEach-Object -Parallel ($computer in $computers)**

**{**

**Restart-Computer -PSComputerName $computer -Force -Verbose**

**}**

**}**

**Restart-AllComputers -Computers $computers**

**IMPORTANT: On ne peut pas prédire l'ordre des résultats lorsqu'on exécute des tâches en parallèles.**

**Comment afficher le nom de l'ordinateur**

Il existe plusieurs manières d'afficher le nom de l'ordinateur.

**Méthode 1: utilisation de hostname.exe** ->hostname.exe

**Méthode 2: utilisation de la variable d'environnement** -> $env:COMPUTERNAME

**Méthode 3: utilisation d'un objet WMI** ->(Get-WMIObject Win32\_ComputerSystem).Name

**Méthode 4: utilisation d'une instance CIM** ->(Get-CIMInstance CIM\_ComputerSystem).Name

**Méthode 5: utilisation d'une méthode ".Net Framework"** ->[system.environment]::MachineName

**Méthode 6: utilisation d'une méthode ".Net Framework"**->[system.net.dns]::GetHostName()

**Méthode 7: utilisation du cmdlet Get-ComputerInfo** -> (Get-ComputerInfo).CsName

**Comment trouver la méthode la plus rapide**

Le cmdlet Measure-Command permet de mesurer la **vitesse d'exécution** d'une commande.

exemple: Measure-Command { hostname.exe }

La propriété TotalMilliseconds permet de comparer facilement la vitesse d'exécution d'une méthode par rapport à une autre.

Le cmdlet Measure-Object permet d'effectuer des calculs comme la moyenne

exemple:

**1..100 | Foreach-Object { Measure-Command { hostname.exe } } | Measure-Object -Average TotalMilliseconds**

On exécute 100 fois le cmdlet Measure-Command et le cmdlet Measure-Object calcule la moyenne de la propriété "TotalMilliseconds".

Effectuons des tests pour déterminer la différence dans le temps d'exécution.

**Clear-Host**

**# IMPORTANT: on ne doit pas utiliser des variables qui contiennent les commandes**

**# Foreach-Object possède deux alias: foreach et %**

**# Les tests vont du plus rapide au plus lent.**

**1..100 | % {Measure-Command {[system.environment]::MachineName}} | `**

**Measure-Object -Average TotalMilliseconds**

**0,008832 ms**

**1..100 | % {Measure-Command {$env:computername}} | `**

**Measure-Object -Average TotalMilliseconds**

**0,024194 ms**

**1..100 | % {Measure-Command {[system.net.dns]::GetHostName()}} | `**

**Measure-Object -Average TotalMilliseconds**

**0,047328 ms**

**1..100 | % {Measure-Command {(Get-WMIObject Win32\_ComputerSystem).Name}} | `**

**Measure-Object -Average TotalMilliseconds**

**8,362251 ms**

**1..100 | % {Measure-Command {hostname.exe}} | `**

**Measure-Object -Average TotalMilliseconds**

**9,12406 ms**

**1..100 | % {Measure-Command {(Get-CIMInstance CIM\_ComputerSystem).Name}} | `**

**Measure-Object -Average TotalMilliseconds**

**9,330143 ms**

**1..100 | % {Measure-Command {(Get-ComputerInfo).CsName}} | `**

**Measure-Object -Average TotalMilliseconds**

**1659,054235 ms**

**Comparaison de la vitesse d’exécution des commandes**

La commande 1 est toujours la plus rapide avec un temps d’exécution d’environ **0.008 ms**

Les commandes 2 et 3 sont environ 5 fois plus lentes que la commande 1

Les commandes 4, 5 et 6 sont environ 1000 fois plus lentes que la commande 1

La commande 7 est environ 200 000 fois plus lente que la commande 1

**La commande la plus rapide**

**"[system.environment]::MachineName" est toujours extrêmement rapide.**

**La commande la plus lente**

**"(Get-ComputerInfo).CsName" est toujours extrêmement lente.**

**Les fichiers de configuration pour PowerShell**

* Un profil applicable à tous les utilisateurs et aux consoles powershell.exe et powershell\_ise.exe.
* **$PSHOME\profile.ps1**
* Un profil applicable à tous les utilisateurs et à la console powershell.exe.
* **$PSHOME\Microsoft.PowerShell\_profile.ps1**
* Un profil applicable à tous les utilisateurs et à la console powershell\_ise.exe.
* **$PSHOME\Microsoft.PowerShellISE\_profile.ps1**
* Un profil applicable à l'utilisateur courant et aux consoles powershell.exe et powershell\_ise.exe.
* **$HOME\Documents\WindowsPowerShell\profile.ps1**
* Un profil applicable à l'utilisateur courant et à la console powershell.exe.
* **$HOME\Documents\WindowsPowerShell\Microsoft.PowerShell\_profile.ps1**
* Un profil applicable à l'utilisateur courant et à la console powershell\_ise.exe
* **$HOME\Documents\WindowsPowerShell\Microsoft.PowerShellISE\_profile.ps1**

**L'ordre d'exécution des fichiers de configuration pour "powersell.exe"**

$PROFILE | Select-Object \*

AllUsersAllHosts : $PSHOME\profile.ps1

AllUsersCurrentHost : $PSHOME\Microsoft.PowerShell\_profile.ps1

CurrentUserAllHosts : $HOME\Documents\WindowsPowerShell\profile.ps1

CurrentUserCurrentHost : $HOME\WindowsPowerShell\Microsoft.PowerShell\_profile.ps1

**L'ordre d'exécution des fichiers de configuration pour "powershell\_ise.exe"**

$PROFILE | Select-Object \*

AllUsersAllHosts : $PSHOME\profile.ps1

AllUsersCurrentHost : $PSHOME\Microsoft.PowerShellISE\_profile.ps1

CurrentUserAllHosts : $HOME\Documents\WindowsPowerShell\profile.ps1

CurrentUserCurrentHost : $HOME\Documents\WindowsPowerShell\Microsoft.PowerShellISE\_profile.ps1

# Commande pour afficher les variables dont le nom débute par **ps** donc des variables de PowerShell

**Get-Variable ps\***

La variable **$psISE** permet de configurer l'environnement ISE de PowerShell.

**Exemple d'un fichier $PSHOME\profile.ps1**

**# Modification de la variable BufferSize**

**$InfoHost = Get-Host**

**$InfoWindow = $InfoHost.UI.RawUI**

**$NewSize = $InfoWindow.BufferSize**

**$NewSize.Height = 8192**

**$NewSize.Width = 512**

**$InfoWindow.BufferSize = $NewSize**

**# Force l'affichage du volet de script dans "PowerShell ISE"**

**if ($psISE.CurrentPowerShellTab.ExpandedScript -eq $false)**

**{**

**$psISE.CurrentPowerShellTab.ExpandedScript = $true**

**}**

**# Modification du texte dans la barre de titre**

**$InfoWindow.WindowTitle = "Console de " + [system.environment]::UserName**

**Exemple d'un fichier $HOME\Documents\WindowsPowerShell\profile.ps1**

**# Change le dossier actif**

**Set-Location E:\scriptPS1**