Leçon 2 : PowerShell

2.1 Introduction

Powershell est le langage de script mis à disposition par Microsoft sur les systèmes d'exploitation de nouvelle génération (Windows Server 2016, Windows 10, Windows Server 2012, Windows 8.1/7). Ce langage avancé est fortement utilisé par les administrateurs systèmes pour automatiser leurs tâches. C'est pourquoi nous allons nous concentrer, dans cette leçon, à l'étude de ce langage de script.

Attention : Il s'agit d'une simple introduction à Powershell. Bien sûr, nous aborderons celui-ci souvent lorsque nous étudierons certains aspects du système. Nous allons donc d'abord commencer par les éléments simples du langage avant d'approfondir celui-ci tout au long de notre étude.

PowerShell est complet et complexe. Dans un souci pédagogique, la présentation faite ici est simplifiée. Les lecteurs souhaitant une description plus précise peuvent se référer à la référence bibliographique.

Ressource bibliographique

Cette leçon s'inspire du livre suivant : « William R. Stanek, *Windows PowerShell™ 6 – IT Pro Solutions*, Stanek & Associates, 2017 »

2.2 Démarrer le shell PowerShell

Pour démarrer le *shell* spécifique *PowerShell*, il faut simplement effectuer une recherche et celui-ci est trouvé assez facilement :

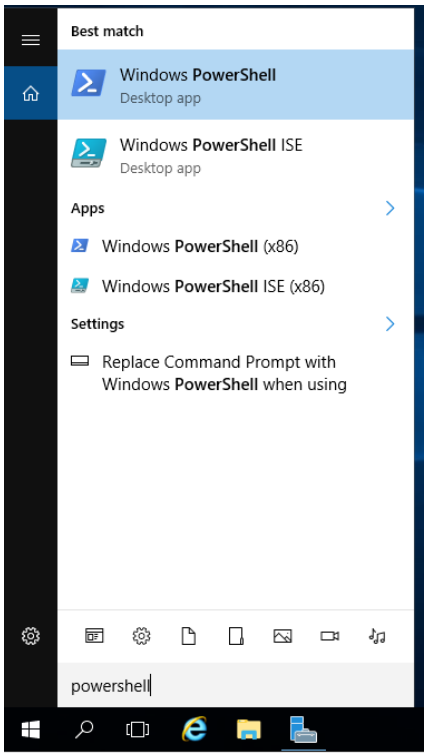

Figure 2.1 : Lancement de l'environnement Powershell

Les **scripts powershell** sont des fichiers textes portant l'extension **.ps1**. **Windows Powershell** est le terminal permettant d'exécuter des scripts Powershell. **Windows Powershell ISE** est l'outil de « développement » Powershell.

2.3 Premier script et première exécution

En utilisant un éditeur quelconque ou **Windows Powershell ISE**, il est possible de créer un premier script *PowerShell* assez simple :

```
$a = "Hello"
$b = "world !"
write-output "$a $b"
write-output $a $b
```

Script 2.1 : « Hello Word » en PowerShell

Une fois ce script entré, on va l'enregistrer sous le nom **helloworld.ps1**. Pour exécuter ce script, il suffit :

1. De démarrer le shell Powershell
2. D'aller dans le dossier contenant le script
3. Démarrer le script en entrant :
   `PS C:\Users\Administrateur\Documents> .\helloworld.ps1`

Le résultat à l'écran devrait être :

```
Hello world !
Hello
world !
```

Cependant, il *se peut* que l'erreur suivante survienne :
Impossible de charger le fichier (…) car l'exécution de scripts est désactivée sur ce système.

Pour des raisons de sécurité, les systèmes Windows Serveur peuvent ne pas exécuter directement des scripts non-signés numériquement. Si cela se produit, Il est possible de changer ce comportement en exécutant la commande suivante dans le PowerShell :

`PS C:\Users\Administrateur\Documents> Set-ExecutionPolicy unrestricted`

Une fois la commande entrée, il faut répondre *Oui* à la question posée et les scripts pourront être exécutés.

Dans ce premier exemple simple, nous avons vu comment assigner une valeur à une variable mais également comment afficher quelque chose à l'écran (`Write-Output` ou encore `Write-Host`).

Pour lire un élément au clavier, il faut utiliser `Read-Host`. Ainsi, on peut écrire dans un script la ligne suivante pour capturer une chaîne de caractères :

```
$val = Read-Host "Entrez une chaine "
```

La commande Read-Host ajoute automatiquement un symbole « : » après le texte mentionné. On verra ainsi apparaître à l'écran ceci :

```
Entrez une chaine :
```

Si l'élément à lire est une valeur entière, il convient de forcer le type de la variable :

```
[int] $val = Read-Host "Entrez une valeur entiere "
```

2.4 Les Commandlet

PowerShell[5] est un invite de commande acceptant du scripting. Ainsi, il comprend les commandes standards héritées de MS-DOS comme *cls*, *cd*, *md*, *dir*, …

En plus des commandes standards, PowerShell ajoute des *commandlet* (appelée aussi *cmdlet*). Il s'agit de commandes internes à l'invite et qui peuvent être appelées directement. Ces *cmdlet* sont composées d'un verbe et d'un nom.

Dans le tableau 1, nous trouverons une liste des principaux verbes utilisés dans le langage PowerShell.

Verbe Cmdlet	Signification
Add	Ajoute une occurrence ou un élément
Clear	Efface le contenu d'un objet
ConvertFrom	Convertit un élément à partir d'un format vers autre
ConvertTo	Convertit un élément dans un format particulier
Disable	Désactive un élément actif
Enable	Active un élément désactivé (ou inactif)
Export	Exporte les propriétés d'un élément dans un format donné
Get	Questionne (envoie une requête à) un objet donné ou (à) un sous-ensemble d'un type d'objet
Import	Importe les propriétés d'un élément à partir d'un format donné
Invoke	Exécute une occurrence d'un objet
New	Crée une nouvelle occurrence d'un élément
Remove	Supprime l'instance d'un objet
Set	Modifie des paramètres spécifiques d'un objet
Start	Démarre une occurrence d'un élément
Stop	Arrête l'occurrence d'un élément
Test	Vérifie l'occurrence d'un élément pour un état ou une valeur donnée
Write	Réalise une opération d'écriture sur l'instance d'un objet

Tableau 2.1 : Aperçu des verbes[6]

Dans le tableau 2, nous trouvons quelques *cmdlet* utiles pour l'administrateur système.

Commande cmdlet	Description
Add-Computer, Remove-Computer	Ajoute ou supprime un ordinateur membre d'un domaine ou d'un workgroup
Checkpoint-Computer, Restore-Computer	Crée un point de restauration, restaure d'un point de restauration
Compare-Object, Group-Object, Sort-Object, Select-Object, New-Object	*Cmdlet* pour comparer, grouper, trier, sélectionner et créer des objets
ConvertFrom-SecureString, ConvertTo-SecureString	*Cmdlet* pour créer ou exporter une chaine sécurisée
Debug-Process	Débogue un processus actif sur un ordinateur
Get-Alias, New-Alias, Set-Alias, Export-Alias, Import-Alias	*Cmdlet* pour obtenir, créer, modifier, exporter ou importer des raccourcis
Get-AuthenticodeSignature, Set-AuthenticodeSignaure	Obtenir ou placer une signature à un fichier
Get-Command, Invoke-Command, Measure-Command, Trace-Command	*Cmdlet* pour obtenir des informations sur des *Cmdlet*, invoquer des commandes, mesurer le

[5] Donc si vous démarrer **Windows Powershell**
[6] Ce tableau est extrait du livre de référence

	temps d'exécution de commandes ou tracer des commandes
Get-Counter	Obtenir les données du compteur de performance
Get-Credential	Obtenir un objet *Credential* basé sur le mot de passe
Get-Date, Set-Date	Obtenir ou fixer la date et l'heure courante
Get-EventLog, Write-EventLog, Clear-EventLog	Obtenir des événements, écrire des événements ou effacer des événements d'un journal
Get-ExecutionPolicy, Set-ExecutionPolicy	Obtient ou définit la politique d'exécution pour le shell courant
Get-Host	Obtient les informations à propos du système exécutant PowerShell
Get-HotFix	Obtient les patches et autres mises à jour appliquées à l'ordinateur
Get-Location, Set-Location	Affiche ou définit des informations concernant l'espace de travail actuel
Get-Process, Start-Process, Stop-Process	Obtient, démarre ou arête un processus
Get-PSDrive, New-PSDrive, Remove-PSDrive	Obtient, crée ou supprime un lecteur PowerShell spécifié
Get-Service, New-Service, Set-Service	Obtient, créée ou définit un service système
Get-Variable, New-Variable, Set-Variable, Remove-Variable, Clear-Variable	*Cmdlet* pour obtenir, créer, définir et supprimer une variable ou sa valeur
Import-Counter, Export-Counter	Importe ou exporte le compteur de performance des fichiers journaux
Limit-EventLog	Fixe la taille et la durée d'un événement dans le journal
New-EventLog, Remove-EventLog	Créée ou supprime un événement personnalisé et sa source
Ping-Computer	Envoie des requêtes ICMP vers la destination
Pop-Location	Obtient une localisation en tête de pile
Push-Location	Sauvegarde une localisation en tête de pile
Read-Host, Write-Host, Clear-Host	Lit une entrée à partir - , écrit une sortie vers -, efface – la fenêtre
Rename-Computer, Stop-Computer, Restart-Computer	Renomme, arrête ou redémarre un ordinateur
Reset-ComputerMachinePassword	Modifie et annule le mot de passe utilisé par la machine pour s'identifier dans le domaine
Show-EventLog	Affiche un événement de l'ordinateur dans l'observateur d'événements
Show-Service	Affiche les services de l'ordinateur dans l'outil de gestion des services
Start-Sleep	Suspend l'exécution du shell ou du script pendant une période de temps définie
Stop-Service, Start-Service, Suspend-Service, Resume-Service, Restart-Service	*Cmdlet* pour arrêter, démarrer, suspendre, reprendre ou redémarrer un service
Wait-Process	Attend la fin du processus avant de continuer
Write-Output	Ecrit un objet dans le *pipeline*
Write-Warning	Affiche un message d'avertissement

Tableau 2.2: Aperçu des commandes courantes[3]

Les *cmdlets* permettent d'enrichir les options de scripts. Ainsi, comme nous l'avons vu dans le code 2.1, `Write-Output` est un cmdlet permettant d'afficher une information à l'écran.

2.5 Obtenir de l'aide

PowerShell inclut beaucoup d'aide et même des exemples d'utilisation. Grâce à la *cmdlet* `Get-Help`, il est possible d'obtenir des informations précises.

Ainsi, si vous entrer ceci :

```
PS C:\Users\Administrateur\Documents> Get-Help Get-Date
```

Vous obtenez le manuel de la *cmdlet* `Get-Date`. Sans surprise, cette commande permet d'obtenir la date et l'heure courante. Comme plusieurs formats sont possibles, la page de manuel permet facilement de comprendre le format attendu.

Enfin, **des exemples** sont également fournis. Ainsi, si vous entrez :

```
PS C:\Users\Administrateur\Documents> Get-Help Get-Date -examples
```

Vous obtenez des exemples d'utilisation de `Get-Date`. Pour être sûr de pouvoir lire tous ces exemples, il est conseillé d'ajouter la commande **more** afin de voir l'affiche de manière paginée. Ainsi la commande devient :

```
PS C:\Users\Administrateur\Documents> Get-Help Get-Date –examples | more
```

L'affichage est ainsi rendu page après page.

2.6 Types de variable en Powershell

Les variables élémentaires

Les variables élémentaires sont des variables qui peuvent contenir des chaînes de caractères, des nombres ou encore des valeurs booléennes. Pour déclarer ce type de variable, il faut simplement préfixer le nom de celle-ci par le symbole `$`. On peut **forcer le type** d'une variable en précisant son type au moment de son utilisation. Il y a beaucoup de types différents, les principaux sont :

Type	Description
[int]	Entier 32 bits signé
[long]	Entier 64 bits signé
[bool]	Booléen (True / False)
[string]	Chaîne de caractères de taille fixe Unicode
[char]	Un caractère unicode (codage sur 16 bits)
[byte]	Un caractère non-signé (codage sur 8 bits)
[decimal]	Une valeur décimale codée sur 128 bits
[single]	Un nombre en virgule flottante codé sur 32 bits
[double]	Un nombre en virgule flottante codé sur 64 bits

Les opérations sur **les chaînes de caractères** :

Concaténation	La concaténation entre des chaînes de caractères s'obtient également avec l'opérateur « + » Ex: Write-output ($a + $b)

Comparaison	Pour effectuer des comparaisons : -eq vérifie si 2 chaines sont identiques -ne vérifie si 2 chaines sont différentes -ge vérifie si la première est plus grande ou égale à la seconde -gt vérifie si la première est plus grande que la seconde -lt vérifie si la première est plus petite que la seconde -le vérifie si la première est plus petite ou égale à la seconde -match effectue une vérification sur base d'une expression régulière -replace permet d'effectuer un remplacement
Modification[7]	$a = "The Scriptign Guys" $a = $a.Replace("Scriptign", "Scripting") *Remplace une chaîne par une autre* $b1 = $a.Substring(4) *Extrait une sous-chaine d'une chaîne à partir de la 5ème position. $b1 contient donc la chaîne « Scripting Guys ».* $b2 = $a.Substring(4,9) *Extrait une sous-chaine de 9 caractères d'une chaîne à partir de la 5ème position. $b2 contient donc la sous-chaîne « Scripting ».* $c1 = $a.ToLower() *Convertit toutes les lettres en minuscule. La chaîne $c1 contiendra la valeur « the scripting guys »* $c2 = $a.ToUpper() *Convertit toutes les lettres en majuscule. La chaîne $c2 contiendra la valeur « THE SCRIPTING GUYS »* $d = $b2.ToCharArray() *Convertit une chaîne de caractères en tableau de caractères. Le tableau $d contiendra les éléments @("S","c","r","i","p","t","i","n","g")* $tab = $a -split " " *Découpe la chaîne de caractères en utilisant le séparateur mentionné (espace dans cet exemple). Le résultat est un tableau comprenant les différents éléments. Dans notre exemple, $tab contiendra @("The", "Scripting", "Guys")* $e = $tab -join "*" *Applatit tous les éléments du tableau dans une chaîne de caractères. Les éléments seront séparés par le caractère mentionné (« * » dans cet exemple). La chaîne $e contiendra la valeur « The*Scripting*Guys »*

Les opérations sur **les données numériques :**

[7] Ces exemples sont extraits du site *Technet* de Microsoft : http://technet.microsoft.com/en-us/library/ee692804.aspx

Arithmétique	Les opérations arithmétiques sont conformes à celles que l'on trouve dans le langage C. Ainsi :
	`$a = $a + 1 ; $a +=1 ; $a++`
	`$a = $a - 1 ; $a =-1 ; $a--`
	`$a = $a * 3 ; $a *= 3`
	`$a = $a / 5 ; $a /= 5`
	
	Le reste de la division entière s'obtient avec le symbole `%`
	`$reste = 5 % 2  # Le reste vaut 1`

Comparaison	Pour effectuer des comparaisons : -eq vérifie si 2 nombres sont identiques -ne vérifie si 2 nombres sont différentes -ge vérifie si le premier est plus grand ou égale au second -gt vérifie si le premier est plus grand que le second -lt vérifie si le premier est plus petit que le second -le vérifie si le premier est plus petit ou égale au seconde
Affichage d'une expression	S'il faut afficher le résultat d'une expression directement à l'intérieur d'un cmdlet, il convient d'entourer l'expression de parenthèses afin que Powershell identifie clairement celle-ci : `Write-Output ("Le reste de la division entière de 5 par 2 vaut " + (5 % 2) )`

Les tableaux

Powershell permet à l'utilisateur d'utiliser des tableaux. Les tableaux sont des variables *simples* qui contiennent des éléments indicés. Ainsi, ils sont également créés en utilisant le symbole `$`. Pour initialiser les éléments d'un tableau, il suffit de lister ses éléments :

```
$montableau = @(3,5,9,10)
$montableau2 = @("Bonjour", "Salut", "Hello")
$montableau3 = @()
```

Il est possible d'accéder à un élément du tableau via son indice. Le premier élément se trouve à la position 0. Par exemple :

```
Write-output ("Le second élément se trouve à la position 1 et vaut " +
       $montableau[1])
```

Dans ces exemples, on déclare et initialise un tableau avec un nombre fixe d'élément. Ainsi, il n'est pas possible d'accéder à l'indice 3 du tableau `$montableau2` puisque celui-ci contient 3 éléments (stockés aux indices 0, 1 et 2). Si l'on souhaite « étendre » un tableau (ajouter dynamiquement un élément supplémentaire), il faut utiliser l'opérateur `+=`.

Quelques opérateurs sur les tableaux :

```
$montableau.count
```
Retourne le nombre d'éléments dans le tableau (4)

```
$montableau2 = $montableau2 | Sort-Object
```
Permet de trier les éléments du tableau (Bonjour,Hello,Salut).

```
$montableau3 += 5
```

Etend le tableau `$montableau3` (actuellement vide) en lui ajoutant l'élément « 5 » à la position 1 (donc à l'indice 0).

Les tables hachées

Powershell permet une utilisation simple des tables hachées (ou tableaux associtatifs). Pour rappel, il s'agit de faire correspondre à une clé (unique donc) une valeur déterminée. Les tables hachées sont utilisées dans beaucoup de cas, mais notamment, pour garantir l'unicité d'un ensemble.

```
$capitales = @{"Belgique" = "Bruxelles"; "France" = "Paris"; "Allemagne" =
"Berlin"; "Suisse" = "Berne" }

$hash = @{}
```

Nous pouvons effectuer les opérations suivantes :

```
$capitales.Add("Pays-Bas", "Amsterdame")
```
Dans cet exemple nous ajoutons un élément à la table.

```
$capitales.Remove("Allemagne")
```
Dans cet exemple nous supprimons de la table la valeur associée à la clé Allemagne.

```
$capitales.Set_Item("Pays-Bas", "Amsterdam")
```
Dans cet exemple nous modifions la valeur associée à la clé « Pays-Bas ».

```
$capitales.Get_Item("Belgique")
$capitales["Belgique"]
```
Dans cet exemple nous récupérons la valeur associée à la clé « Belgique ». La valeur récupérée est donc « Bruxelles ».

```
if($capitales.ContainsKey("Belgique")) {
```
Dans cet exemple nous vérifions si la clé « Belgique » est définie dans la table.

```
if($capitales.ContainsValue("Berlin")) {
```
Dans cet exemple nous vérifions si la valeur « Berlin » est stockée dans la table.

```
$capitales.keys
```
Cette propriété de la table hachée donne la liste des clés définies.

```
$capitales.values
```
Cette propriété de la table hachée donne la liste des valeurs stockées dans la table.

2.7 Quelques opérateurs

Powershell propose, comme dans tous les langages, des connecteurs logiques pour lier des expressions. Il y a également d'autres opérateurs comme les opérations bits à bits. Voici une liste de ces opérateurs :

Opérateurs	Description
-and	Effectue un ET logique entre les deux expressions. Vrai si chaque expression est vraie.

-or	Effectue un OU logique entre les deux expressions. Vrai si au moins une des expressions est vraie.
-xor	Effectue un OU EXCLUSIF entre les deux expressions. Vrai si une seule expression est vraie.
-not	Effectue une NEGATION de l'expression. Vrai si le résultat de l'expression est fausse.
-band	Effectue un ET bit à bit entre deux valeurs
-bor	Effectue un OU bit à bit entre deux valeurs
-bnot	Effectue un NOT bit à bit (inverse tous les bits) de la valeur donnée
-bxor	Effectue un OU EXCLUSIF bit à bit entre deux valeurs

2.8 Les sélections

Powershell permet la sélection au moyen des structures `if` et `switch`. Il faut **être vigilant à bien utiliser les opérateurs vus** précédemment.

```
[int] $val = Read-Host "Entrez une valeur "

if($val -gt 0) {
  Write-Output "La valeur est positive !"
} elseif($val -lt 0) {
  Write-Output "La valeur est négative !"
} else {
  Write-Output "La valeur est nulle !"

}
```

La structure du `if` est conforme à celle du langage C. La partie `elseif` est bien sûr facultative.

```
[string]$val = Read-Host "Entrez un chiffre en lettre "
switch($val) {
  "zero" { $val_num = 0}
  "un" { $val_num = 1 }
  "deux" { $val_num = 2 }
  "trois" { $val_num = 3 }
  "quatre" { $val_num = 4 }
  "cinq" { $val_num = 5 }
  "six" { $val_num  = 6 }
  "sept" { $val_num = 7 }
  "huit" { $val_num = 8 }
  "neuf" { $val_num = 9 }
  default { $val_num = -1 }
}

if($val_num -ge 0) {
  Write-Output ("Le chiffre entré est " + $val_num + " et son carre vaut: "
    + ($val_num * $val_num))
}
```

La structure `switch` est assez simple à écrire. Elle admet l'utilisation de chaîne de caractères ou des données numériques.

2.9 Les répétitions

Powershell admet plusieurs formes de répétitions. On dénombre ainsi les boucles de type *while* et *for* mais également des boucles de type *foreach*.

Les boucles while

```
$i = 0
while($i -le 5) {
  Write-Output ("Tour dans la boucle = " + $i)
  $i++
}
```

Dans cet exemple, il y aura 6 tours dans la boucle (puisque i va évoluer de 0 à 5).

Les boucles for

```
for($i=0 ; $i -le 5 ; $i++) {
  Write-Output ("Tour dans la boucle = $i")
}
```

Dans cet exemple, il y aura également 6 tours dans la boucle. La structure de la boucle *for* est très proche de celle utilisée en C. **Attention aux opérateurs de comparaison !**

Les boucles foreach

```
$tableau = @(1,4,9,0,3,1)
foreach($element in $tableau) {
  Write-Output ("Elément courant = " + $element)
}
```

Dans cet exemple, on parcourt tous les éléments d'un tableau un à un.

La boucle *foreach* est particulièrement adaptée au parcours d'une table hachée :

```
$capitales = @{"Belgique" = "Bruxelles"; "France" = "Paris"; "Allemagne" =
"Berlin"; "Suisse" = "Berne" }
foreach ($pays in $capitales.keys) {
  Write-Output ("La capitale de $pays est $($capitales[$pays])")
}
```

Nous utilisons ici l'interpolation de chaines en Powershell. Il faut noter l'écriture particulière lorsqu'il s'agit d'aller retrouver la valeur associée à un tableau associatif avec un $() encadrant l'ensemble.

2.10 Parcourir un fichier texte

Powershell permet de parcourir un fichier texte et d'en traiter chaque ligne. C'est particulièrement utile s'il faut **créer des utilisateurs** à partir d'une liste, **créer des dossiers**, … Le fonctionnement est assez simple, il faut utiliser la cmd-let `Get-Content` et ensuite, utiliser des expressions régulières pour traiter le résultat.

Si le fichier est conforme au schéma suivant (`C:\monfichier.txt`) :

```
Element1;Element2;Element3
Element1;Element2;Element3
Element1;Element2;Element3
Element1;Element2;Element3
```

Voici comment il est possible d'en traiter les différentes lignes :

```
$contenu = Get-Content "C:\monfichier.txt"
foreach($ligne in $contenu) {
  if($ligne –match '^(.*);(.*);(.*)$') {
    write-output ("Element 2  = $($matches[2])")
  }
}
```

Dans cet exemple, nous commençons par lire le contenu du fichier grâce au cmd-let `Get-Content`. Le tableau contenant les différentes lignes du fichier s'appelle `$contenu`. Grâce à une boucle *foreach*, il est possible de parcourir chaque ligne et d'analyser celle-ci grâce à une expression régulière. Enfin, grâce à la variable `$matches`, il est possible de récupérer les éléments capturés dans l'expression.

Quelques éléments d'une **expression régulière** :

[...]	1 caractère compris dans la séquence mentionnée. Ex : `[a-z]` *N'importe quelle lettre*
[^...]	1 caractère non compris dans la séquence mentionnée. Ex : `[^;]` *N'importe quel caractère qui n'est pas le signe de ponctuation « ; »*
.	N'importe quel caractère
\w	Une lettre
\W	N'importe quel caractère qui n'est pas une lettre
\d	Un chiffre
\D	N'importe quel caractère qui n'est pas un chiffre
\s	Un espace, une tabulation
\S	N'importe quel caractère qui n'est pas un espace ou une tabulation
*****	Répétition 0-N Ex : `.*` Séquence éventuelle de n'importe quel caractères
+	Répétition 1-N Ex : `.+` *Séquence d'au moins un caractère quelconque*
^	Correspond au début de la chaîne
$	Correspond à la fin de la chaîne
()	Permet de capturer un élément accessible par après au moyen de la variable `$matches[]`. **Attention !** `$matches[0]` reprend l'ensemble de la ligne. Les éléments capturés commencent donc à l'indice 1.

2.11 Commentaires en Powershell

Les commentaires peuvent, comme en C, être limité à une seule ligne. Dans ce cas, il suffit de préfixer la ligne par le symbole #

```
# Ceci est un commentaire sur une ligne
```

Si le commentaire s'étale sur plusieurs lignes, il faut utiliser les éléments <# et #> comme délimiteur.

```
<# Voici un commentaire
qui se trouve
sur plusieurs lignes #>
```

2.12 Quelques commandes utiles

Créer un dossier

```
PS C:\...> New-Item C:\Scripts -type directory
```

Se placer dans un dossier

```
PS C:\...> Set-Location C:\Scripts
```

Exécuter un programme ou une commande (résultat affiché)

```
PS C:\...> Invoke-expression "C:\Windows\system32\tasklist.exe"
```

Exécuter un programme ou une commande (résultat caché)

```
PS C:\...> Invoke-expression "C:\Windows\system32\tasklist.exe" | out-null
```

Exécution directe et récupération du résultat dans une variable

```
PS C:\...> $resultat = &"C:\Windows\system32\tasklist.exe"
```

2.13 Pour finir …

Il est bien clair que cette leçon constitue **une introduction** à PowerShell. Des éléments importants utilisables à l'intérieur de scripts seront vus par la suite. N'oubliez pas de combiner tous ces éléments afin de constituer vos scripts !

La plupart des cmdlet accepte bon nombre d'options. Ainsi, par exemple, le cmdlet de lecture d'une information au clavier `Read-Host` admet une option `-AsSecureString` qui permet de lire les données sensibles (comme un mot de passe). Les options sont détaillées dans l'aide associée à chaque cmdlet (`Get-Help Read-Host` par exemple).

Internet et, en particulier, le site de Microsoft regorge d'information et d'exemples sur l'utilisation de PowerShell (par exemple : http://technet.microsoft.com/en-us/scriptcenter/powershell.aspx). N'oubliez donc pas d'aller y jeter un coup d'œil pour trouver des solutions à des problèmes rencontrés.

2.14 Exercices

1. Sur base du fichier « `liste-users.csv` », on vous demande, pour chaque ligne présente dans le fichier, **d'afficher à l'écran** le contenu de celle-ci en ajoutant deux nouveaux champs : le *login* et le *mot de passe* de l'utilisateur.
    a. Le login sera formé comme suit : la 1ère et la dernière lettre de la catégorie (**a**dministrati**f** → **af**). Ces lettres seront suivies d'un numéro de séquence par catégorie. Ainsi, `sl0073` identifie le 73ème utilisateur de la catégorie *social* alors que `pl0040` identifie le 40ème utilisateur de la catégorie *personnel*.
    b. Le mot de passe comptera exactement 8 caractères et comprendra 2 chiffres, deux lettres majuscules et 4 lettres minuscules. Pour composer ce mot de passe, vous pouvez faire appel à la cmdlet `Get-Random` qui fournit un nombre aléatoire. On vous demande de coder vous-même la génération du mot de passe. Pensez que la position des lettres et chiffres dans le mot de passe est aléatoire (il serait incorrect de supposer que le mot de passe commence par une minuscule puis un chiffre, ...). **Attention ! Tous les utilisateurs numérotés 50 (af0050, pl0050, ie0050, …) doivent avoir « P@ssw0rd » comme mot de passe**.

    Les lignes commençant par un '#' doivent être ignorées, ce sont des commentaires.

(Solution : le script fait moins de 25 lignes)