Parcours PowerShell

Table of Contents

Préambule.

Dans ce document nous ferons régulièrement usage des symboles de la syntaxe Microsoft qui sera là même utilisés sur les pages d'aides avec PowerShell.

- Les crochets autour des paramètres et/ou de la valeur du paramètre indiquent des éléments **facultatifs** [ ] .
  Les crochets autour du nom du paramètre, mais pas la valeur du paramètre, indiquent que le nom du paramètre est facultatif. Ces paramètres sont appelés **paramètres positionnels**. Les valeurs de paramètre doivent être présentées dans l'ordre correct pour que les valeurs soient liées au paramètre correct.
- Les crochets ajoutés [] à un type .NET indiquent que le paramètre peut accepter une ou plusieurs valeurs de ce type. Entrez les valeurs sous la forme d'une liste séparée par des virgules.
- Les crochets < > d'angle indiquent que le texte doit être remplacé. Il peut également indiquer le type de valeur attendue : string = chaine de caractere, CommandTypes = un type de commandes, int = un entier, etc ...
- Les accolades {} indiquent une « énumération », qui est un ensemble de valeurs valides pour un paramètre.

Démonstration :

```
# cmdlet Get-Process, -name est optionnel et positionnel, le parametre attend une
valeur de type chaine de caractere
Get-Process [-Name] <string[]>
#sans parametre
PS C:\>Get-Process
# avec parametre
PS C:\>Get-Process -Name scheduler

# le parametre Option accepte les valeurs None ou ReadOnly ou Constant.
New-Alias -Option {None | ReadOnly | Constant}

# Le parametre name accepte plusieurs valeurs de type string, chaine de caractere
Get-Process [-Name] <string[]>
PS C:\>Get-Process -Name Explorer, Winlogon, Services

# cmdlet get-aduser le parametre -filter est obligatoire (mandatory), car pas de
[] qui l'entoure et attend une chaine de caractere
Get-ADUser -filter <String>
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_command_syntax?view=powershell-7.5

1. Introduction

1.1. Principes originels

PowerShell a été créé sur la base de plusieurs principes :

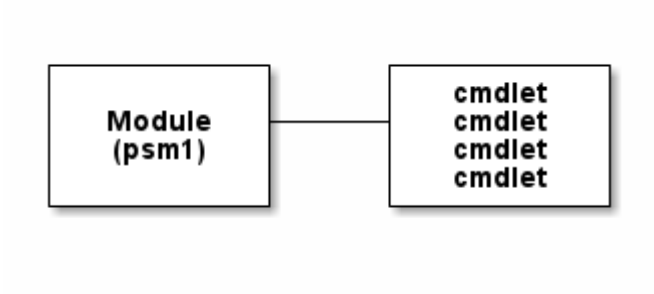
1.1.1. Une construction de cmdlet simple

Les commandes PowerShell sont appelées cmdlet, contraction de commandes et applets.
Elles sont composées d'un verbe, d'un tiret et d'un nom.

```
Verbe-Nom [-parametre <argument>]
```

1.1.2. Un langage modulaire

Un module PowerShell `.psm1`, contient un ensemble de cmdlet.
PowerShell repose sur l'utilisation de modules ce qui lui permet de "maitriser" le nombre de commandes a fournir lors de l'installation, l'ajout d'une fonctionnalité s'accompagne de l'ajout d'un module a PowerShell et donc d'un jeu de cmdlet supplémentaires.



1.1.3. Un langage facilement lisible

ChaqueMotPossedeUneMajuscule / Affiche-Commandes

Les commandes générées avec PowerShell sont parfois longues, mais le fait que chaque mot commence par une majuscule a pour but d'en faciliter la lecture.
N'oubliez pas que l'autocomplétion avec la touche "Tab" fonctionne.

```
PS C:\>Get-ADOrganizationalUnit
PS C:\>Undo-ADServiceAccountMigration
PS C:\>Show-WindowsDeveloperLicenseRegistration
```

1.1.4. Un langage non clivant

Dans le but de faciliter son adoption PowerShell propose des alias des commandes CMD de Windows et GNU/Linux les plus courantes. Néanmoins dans leur utilisation la plus simple.

La définition d'un alias en PowerShell sera évoquée plus loin. Sachez cependant que ce langage intègre dès sa création des équivalents aux commandes internes de bases provenant de CMD ou de GNU/Linux.

ls/dir/cd ... "fonctionneront" sous PowerShell mais pas leurs options : `dir /p` ne fonctionnera pas.

```
# Ces 3 commandes auront le meme effet : afficher le contenu du répertoire courant
PS C:\> dir
PS C:\> ls
PS C:\> Get-ChildItem

# A l'inverse ces commandes produiront des erreurs
PS C:\> ls -la
Get-ChildItem : Impossible de trouver un paramètre correspondant au nom « la ».
Au caractère Ligne:1 : 4
+ ls -la
+    ~~~
    + CategoryInfo          : InvalidArgument : (:) [Get-ChildItem],
ParameterBindingException
    + FullyQualifiedErrorId :
NamedParameterNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
PS C:\> dir /p
dir : Impossible de trouver le chemin d'accès « C:\p », car il n'existe pas.
Au caractère Ligne:1 : 1
+ dir /p
+ ~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\p:String) [Get-ChildItem],
ItemNotFoundException
    + FullyQualifiedErrorId :
PathNotFound,Microsoft.PowerShell.Commands.GetChildItemCommand
```

Table 1. Alias

PowerShell (Cmdlet)	PowerShell (Alias)	CMD.EXE / COMMAND.COM	Shell Unix	Description
Get-ChildItem	gci, dir, ls	dir	ls, dir	Liste les fichiers / répertoires du répertoire (courant)
Get-Content	gc, type, cat	type	cat	Obtenir le contenu d'un fichier
Get-Command	gcm	help	help, which	Liste des commandes
Get-Help	Help, man	help	man	Aide
Clear-Host	cls, clear	cls	clear	Efface l'écran
Copy-Item	cpi, copy, cp	copy	cp	Copier un ou plusieurs fichiers / l'arborescence complète
Move-Item	mi, move, mv	move	mv	Déplacer un fichier / répertoire
Remove-Item	ri, del, erase, rmdir, rd, rm	del, deltree, erase, rmdir, rd	rm, rmdir	Supprimer un fichier / répertoire
Rename-Item	rni, ren, mv	ren, rename	mv	Renommer un fichier / répertoire
Get-Location	gl, pwd	cd	pwd	Afficher le répertoire de travail courant
Set-Location	sl, cd, chdir	cd, chdir	cd	Changer le répertoire courant
Write-Output	echo, write	echo	echo	Afficher des chaînes, variables etc sur la sortie standard
Get-Process	gps, ps	tlist	ps	Liste de tous les processus en cours d'exécution
Stop-Process	spps, kill	kill	kill	Arrêter un processus en cours d'exécution
Select-String	sls, findstr	find, findstr	grep	Recherche d'une chaine de caractère

un exemple de commande PowerShell qui illustre la complexité de relecture lorsqu'on abuse des alias :

```
# Pouvez vous lire ces lignes ?
PS C:\>gci -r | ? {$_.psiscontainer -eq $false} | % {$_.fullname} | ? {$_ -match
'\.txt$'} | % {gc $_ | sls 'error' | select -f 1} | select -exp line
```

Cette commande est difficile à comprendre au premier coup d'œil en raison de l'utilisation excessive d'alias.

Voici la même commande avec les cmdlets complets, qui est beaucoup plus lisible :

```
PS C:\>Get-ChildItem -Recurse | Where-Object {$PSItem.PSIsContainer -eq $false} |
ForEach-Object {$PSItem.FullName} | Where-Object {$PSItem -match '\.txt$'} | ForEach-
Object {Get-Content $PSItem | Select-String 'error' | Select-Object -First 1} |
Select-Object -ExpandProperty Line
```

Cette commande recherche récursivement tous les fichiers .txt, puis cherche la première occurrence du mot "error" dans chacun de ces fichiers et affiche la ligne correspondante. L'utilisation d'alias rend la commande plus courte, mais beaucoup moins compréhensible, surtout pour les débutants ou lors de la relecture du code ultérieurement.

Pour rappel un des piliers fondateurs de PowerShell est la "lisibilité".
L'abus d'alias est dangereux pour la lisibilité !

Synthèse

4 principes :
. Simple
. Modulaire
. Lisible
. Adoptable

Quizz

Qu'est-ce qui rend PowerShell accessible aux utilisateurs de Windows autant qu'aux utilisateurs de Gnu/Linux ?

De quoi est composée une cmdlet ?

Faut-il user et abuser des Alias en PowerShell ?

Quels sont les 4 principes fondateurs de PowerShell ?

1.2. Historique et versions

1.2.1. Historique

- DOS MS-DOS CMD 1981 : Pour l'IBM PC, Microsoft créé MS-DOS.
- GUI et CMD 1995 : Depuis Windows95, Microsoft s'appuiera principalement sur une interface graphique. MS-DOS devient CMD, une invite de commande que l'on utilise beaucoup moins et qui est encore présente de nos jours sur les systèmes d'exploitation Microsoft.
- Monad 2004 : Inspiré par le livre Monadologie du philosophe G.W. Liebniz. le projet est initié pour répondre a un besoin grandissant d'automatisation de l'administration du système d'information. L'interface graphique montrant ses limites sur le sujet. Il est basé sur le .NET framework de Microsoft. Courant 2006 Monad change de nom pour devenir **Windows PowerShell**.
- PowerShell 1.0 - 5
  Nov 2006, PowerShell est officiellement lancé sur les systèmes : XP SP2, Serveur 2003 SP1, Vista (128 cmdlet)

2009 : PowerShell 2.0 arrive avec Windows7 (240 cmdlets)
2012 : PowerShell 3.0
2013 : PowerShell 4.0
2016 : PowerShell 5.1 (2855 cmdlet sur un client)
2016 le projet devient Open Source

- PowerShell 6 - 7.x / PowerShell Core
  2018 : Changement majeur dans le développement de ce langage : il devient OpenSource et Multiplateforme

Il est à noter que depuis la sortie de PowerShell 6.0, c'est toujours la version 5.1 qui est présente lors de l'installation d'un OS Microsoft, mais que lors du lancement d'une invite de commande PowerShelll un message invite l'utilisateur a utiliser la dernière version.

- Rétrocompatibilité v5.1 v7 ?

PowerShell 7 est largement rétrocompatible avec PowerShell 5.1, mais il existe quelques différences notamment dans les relations avec les API ou composants non disponibles.

Coexistence : PowerShell 7 est conçu pour coexister avec Windows PowerShell 5.1, permettant une utilisation côte à côte4. Cela inclut des chemins d'installation distincts, des PSModulePath séparés et des profils distincts pour chaque version

PowerShell 7 introduit un mode de compatibilité qui permet d'exécuter des scripts Windows PowerShell plus anciens sans modification. Ce mode est activé en définissant la variable d'environnement `$env:PSEdition` sur `Desktop`.

1.2.2. Connaitre sa version

```
# La variable contenant les informations de version
PS C:\>$PSVersionTable

# La cmdlet qui permet d'avoir l'information de version
PS C:\>Get-Host
```

Il est important de connaitre la version disponible sur la machine sur laquelle je dois exécuter le script, car je n'ai potentiellement pas les mêmes cmdlet disponibles en PS 1.0, 5.1 et en 7.x.

Synthèse :

- $PSVersionTable
- Get-Host
- 2006 Creation de PowerShell
- 2016 PowerShell Core

Quizz :

Que contient un module PowerShell ?

Quelle cmdlet permet de connaitre la version de PowerShell actuelle ?

Quel est le numéro de version actuelle de PowerShell Core ?

Quand PowerShell a-t-il été créé ?

1.3. Notions de base

Un Shell est une interface en ligne de commande (CLI) ou en mode graphique dans le cas de Windows qui permet d'interagir avec la machine.

1.3.1. Chemin absolu / Chemin relatif

Le **chemin absolu** est le chemin complet depuis la racine du système de fichier jusqu'a la ressource visée.

```
PS C:\>c:\Windows\System32\ping.exe
```

Le **chemin relatif** est le chemin relatif à votre position actuelle dans le système de fichier jusqu'a la ressource visée.

```
PS C:\>set-location "c:\users"
PS C:\users> ..\windows\System32\PING.EXE
# affichera l'aide de ping.exe
```

Après avoir effectué le `Set-Location`, je suis positionné dans le répertoire users de la racine C: `c:\users`, pour atteindre la racine du système de fichier je dois remonter d'un cran dans l'arborescence, ".." me permet de me déplacer "virtuellement" vers le répertoire parent de ma position actuelle donc ` `c:\`, puis le chemin, `\windows\System32\` me permet d'atteindre l'exécutable "ping.exe". Dans le système de fichier lorsque vous listez le contenu d'un répertoire (ls, dir ou Get-ChildItem) vous aurez l'indication **"."** et **".."** : * "." correspond à votre emplacement actuel, 'répertoire courant' * ".." correspond au répertoire parent de votre position, 'répertoire parent'.

1.3.2. Quotes aka Guillemet, simple ou double

Les "doubles quotes" ou guillemets doubles autorisent le Shell utilisé à interpréter les caractères spéciaux.

```
# on part du principe que le compte connecté est Administrator
PS C:\> Write-Host "Bonjour $env:username"
#Resultat :
PS C:\>Bonjour Administrator
```

Les simples "quotes" ou guillemets simples bloquent l'interprétation des caractères spéciaux

```
C:\> Write-Host 'Bonjour $env:username'
# Resultat :
PS C:\>Bonjour $env:username
```

Dans PowerShell le "back quote" (altgr+7) protège de l'interprétation le caractère suivant

```
# on part du principe que le compte connecté est Administrator
PS C:\> Write-Host "Le contenu de la variable `$env:username est $env:username"
#Resultat :
PS C:\>Le contenu de la variable $env:username est Administrator
```

En résumé :

- Double quote "" si je veux utiliser le contenu de mes variables
- Simple quote ' si je veux utiliser le nom de mes variables, mais pas leur contenu.
- Back quote ` si je veux protéger le caractère suivant uniquement.

1.3.3. Commandes internes, Commandes externes et PATH

Les commandes internes sont les commandes présentes nativement dans le Shell que vous utilisez : + `Get-Help` pour powerShell, ls pour bash, dir pour CMD, par exemple.
Les commandes externes sont des exécutables présents sur le système en dehors du Shell en cours d'utilisation ping.exe, nslookup.exe, par exemple.
Le PATH est une variable d'environnement qui indique au Shell ou aller chercher une commande externe. Lorsque je veux exécuter un ping je peux soit taper "`c:\windows\system32\ping.exe 1.1.1.1`", ce qui correspond au chemin absolu de l'exécutable ping avec ce que je souhaite "pinger" : linux.fr. Mais alors pourquoi lorsque je tape la commande ping dans mon CLI il me lance cette commande externe sans me demander le chemin ?
Si il parait logique que le Shell sache trouver ses propres commandes, ca l'est moins pour les commandes qui ne lui appartiennent pas.

Grâce a la variable d'environnement **PATH**, $env:PATH en PowerShell le système me permet de ne taper que "ping 1.1.1.1", ping n'est pas une commande interne le système ne la connaît donc pas, il va chercher dans tous les chemins présents dans le PATH si une application s'appelant ping est présente, ce qui est la cas et va donc l'exécuter et effectuer le traitement.

Cela peut être pertinent pour lancer vos scripts sans avoir a préciser leurs chemins de modifier le PATH de votre système afin d'y intégrer le(s) dossier(s) contenant vos scripts.

```
PS C:\>$env:path

PS
C:\>C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\Window
sPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Program Files\PuTTY\;C:\Program
Files\PowerShell\7\;C:\Users\Administrator\AppData\Local\Microsoft\WindowsApps;C:\User
s\Administrator\AppData\Local\Programs\Microsoft VS Code\bin
```

1.3.4. Flux

Les flux

PowerShell prend en charge les flux de sortie suivants.

Les opérateurs de redirection PowerShell sont les suivants, où `n` représente le numéro de flux. Le **flux de réussite** (`1`) est la valeur par défaut si aucun flux n'est spécifié.

Flux n	Description	Introduite dans	Écrire une applet de commande
1	**Success** Flux	PowerShell 2.0	`Write-Output`
2	**Error** Flux	PowerShell 2.0	`Write-Error`
3	**Warning** Flux	PowerShell 2.0	`Write-Warning`
4	**Verbose** Flux	PowerShell 2.0	`Write-Verbose`
5	**Debug** Flux	PowerShell 2.0	`Write-Debug`
6	**Information** Flux	-PowerShell 5.0	`Write-Information`
n/a	**Progress** Flux	PowerShell 2.0	`Write-Progress`

flux (stdin) → commande → flux **1** (stdout) et flux **2** (stderr)

La notion de « flux 0 » telle qu'elle existe dans les systèmes UNIX (où le flux 0 correspond typiquement à l'entrée standard, stdin) n'existe pas formellement sous ce nom ou avec cette numérotation explicite dans PowerShell.
L'équivalent du flux d'entrée standard dans PowerShell n'est pas représenté comme « flux 0 », mais s'utilise via l'accueil de valeurs du pipeline ou via la lecture explicite (ex : Read-Host pour une interaction utilisateur directe).

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_output_streams?view=powershell-7.5

La redirection

Opérateur	Syntaxe	Description
Out-File	Out-File -FilePath <CheminFic>	redirige la sortie vers chemin
Tee-Object	Tee-Object -FilePath <CheminFic>	sortie de commande à un fichier texte, puis l'envoi au pipeline
>	n>	Envoyez le flux spécifié à un fichier
>>	n>>	**Ajoutez** le flux spécifié à un fichier
>&1	n>&1	Redirige le flux spécifié vers le **flux de réussite**

```
#
PS C:\>Get-ChildItem c:\, absentfic 2>&1
# `2>&1` pour rediriger le **flux d□erreur** vers le **flux de réussite**
PS C:\>Get-ChildItem c:\, absentfic 2>$null
#renvoi le flux d'erreur dans le vide, l'erreur n'apparait pas a l ecran.
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_operators?view=powershell-7.5#redirection-operators
https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_redirection?view=powershell-7.5

Masquer l'affichage d'une cmdlet

Le préfixe [void], viens du .NET, en début de ligne permet de ne pas avoir de sortie écran pour la commande qui suit.

```
[void]<code>
PS C:\>[void]$var #n'affichera rien a l ecran
PS C:\>[System.Collections.ArrayList]$arraylist = "0,1,2,3,4"
$arraylist.add("5")
#affiche 5, qui correspond au numero d'index du tableau pour l'entreee de la valeur 5.
PS C:\>[void]$arraylist.add("6")
#n'affichera rien a l'ecran
```

En PowerShell on pourrait le traduire par un

```
PS C:\>$something | Out-Nul
```

équivalent du "> /dev/null" sous Bash, ou du >$null en cmd.
https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/out-null?view=powershell-7.5

1.3.5. Enchainement de commandes

En PowerShell le ";" permet d'enchainer l'exécution des commandes l'une après l'autre, de façon inconditionnelle. L'une et l'autre seront exécutées, peu importe si elles réussissent ou non.

```
cmdlet1 ; cmdlet2
```

En PowerShell (à **partir de la version** 7), les opérateurs '&&' et '||' permettent d'enchaîner conditionnellement des commandes sur une seule ligne, de façon similaire à Bash ou cmd.
&& : Exécute la commande à droite seulement si la commande à gauche a réussi (retour sans erreur).
|| : Exécute la commande à droite seulement si la commande à gauche a échoué (retour avec erreur).
ils utilisent la variable '$?' (variable de 'code retour').

```
# Si le fichier existe, affiche son contenu
Test-Path .\fichier.txt && Get-Content .\fichier.txt

# Si le fichier n'existe pas, affiche un message d'erreur
Test-Path .\fichier.txt || Write-Host 'Fichier non trouvé'
```

1.3.6. Le pipeline

Le pipeline permet de transmettre chaque résultat d'une cmdlet vers une autre cmdlet.

stdin **cmd** stdout → | → stdin **cmd** → stdout

```
#lister les fichiers avec l'extension txt et lire leur contenu.
PS C:\>Get-ChildItem *.txt | Get-Content
```

Avec l'exemple ci-dessus le flux d'entrée **stdin** de la cmdlet Get-Content est nourri par le résultat d'exécution, donc le flux de sortie **stdout**, de la commandlet Get-ChildItem.
Le pipeline va traiter chaque résultat de la cmdlet Get-ChildItem et non pas un résultat global, on parle alors de "collection" de résultats.

```
# la cmdlet me retourne 2 résultats :
PS C:\>Get-ChildItem *.txt
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        05/01/2023     10:32          11085 fic1.txt
-a----        13/11/2020     10:38            404 fic2.txt

#le contenu de mes fichiers txt est :
#fic1.txt "ceci est fic1.txt"
#fic2.txt "ceci est fic2.txt"
Get-ChildItem *.txt | Get-Content
# a donc pour résultat l'affichage sur deux lignes :
PS C:\>ceci est fic1.txt
PS C:\>ceci est fic2.txt
```

Synthèse :

chemin absolu / relatif
" les caractères spéciaux ne sont pas interprétés
"" les caractères spéciaux sont interprétés
Commandes internes au Shell, externes exécutables en dehors du Shell
Variable d'environnement PATH
Les flux : 1 stdin,2 stdout,3 stderr, mais il en existe d'autres
Les flux peuvent être redirigés avec `>` ou `>>` ou `| Out-File` ou `Tee-Object`

";" enchainement de commandes inconditionnelles
le pipeline transmet la sortie d'une commande vers l'entrée d'une autre

Quizz :

Qu'est-ce qu'un chemin absolu ?

Qu'est-ce qu'un chemin relatif ?

Quelle est la différence entre :
`Write-Host "Bonjour $variable"`
`Write-Host 'Bonjour $variable'`

Quel est le résultat de la cmdlet ? Pourquoi ?
`Get-Service ; Get-Process`

Quel est le résultat des cmdlet ? Pourquoi ?
`Get-service -name "spooler" | Stop-Service`
`Get-service -name "spooler" | Stop-Process`

A quoi sert la variable `$PATH` ?

2. Débuter avec PowerShell

2.1. Une cmdlet ?

Qu'est-ce qu'une commande ?

Une commande est une instruction saisie dans un fichier ou directement dans une invite de commande. Son exécution entrainera un traitement par les ressources de la machine.

Dans PowerShell les commandes sont appelées "cmdlet", car on parle d'applet de commande et cmdlet est la contraction de ces deux termes. Applet signifiant mini-programme effectuant un traitement très ciblé.

Comme dit précédemment les cmdlets sont organisés au sein de modules. Cela a pour avantage de ne pas surcharger le système de base. Les cmdlets deviennent disponibles lorsque l'on installe un service sur un Windows Serveur par exemple, ou lorsque l'on installe manuellement un module sur un client pour faire de l'administration a distance.
Une fois installé le module peut se charger automatiquement à l'appel d'une cmdlet ou être chargé manuellement avec la cmdlet `Import-Module`.

> `$env:PSModulePath` est la variable qui stocke les chemins dans lesquels les modules peuvent être installés.

2.2. Construction

Une cmdlet est toujours composée d'un "verbe", puis d'un "tiret", puis d'un "nom" et peut avoir des paramètres

```
Verbe-Nom -parametre <argument>
```

Obtenir des commandes : `Get-Command < patern >`
Obtenir de l'aide sur une commande : `Get-Help < cmdlet >`

PowerShell regroupe les cmdlets en module, chaque module fonctionne en lien avec une bibliothèque (.dll). Les modules une fois chargés vont précharger dans PowerShell un "jeu" de cmdlet ce qui optimisera leur exécution. Ce fonctionnement modulaire, permet a PowerShell de s'adapter aux services que fournit la machine sans devoir intégrer dès son installation une quantité gigantesque de cmdlet.

> ```
> La maitrise de l'anglais basique vous permettra de (re)trouver facilement une
> cmdlet pour effectuer l'action souhaitée.
> ```

Synthèse :

cmdlet = Verbe-Nom -parametre <argument>
Get-Command permet de trouver une cmdlet

Quizz :

Comment est construite une cmdlet ?

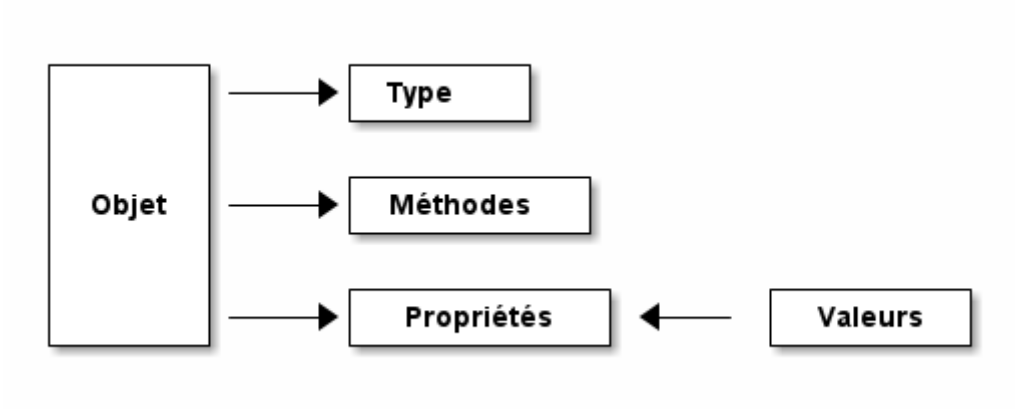
2.3. Langage Objet

Un langage objet, aussi appelé langage de programmation orientée objet, est un type de langage informatique qui s'appuie sur le principe des objets pour structurer et manipuler les données.
Pour appréhender cette notion, vous pouvez prendre l'exemple d'un crayon, d'une voiture, d'une moto… :
Prenons en exemple un objet de **type** crayon.

Ce qui le définit est un ensemble de caractéristiques : sa taille, son poids, son design (à bouchon, à pointe rétractable,….) , son type (feutre, bois, critérium, plume…), sa pointe (large, fine, biseautée …), son encre (indélébile, effaçable…), etc. On parlera en PowerShell de **propriétés**.

Cet objet crayon permet de réaliser des actions : ouvrir le bouchon, écrire, fermer le bouchon, etc… On parler en PowerShell de **méthodes**

La "construction" d'un objet repose sur une "recette" qui se doit d'être identique pour un type d'objet donné. Cela permettra d'assurer une cohérence au système. Chacun de ses objets aura donc des propriétés identiques, chaque propriété pourra contenir des valeurs différentes.



2.3.1. Type d'objet et Membres de la classe d'objet

Pour connaitre le type d'objets, on utilise la cmdlet `<cmdlet> | Get-Member`.+ Cette cmdlet permet de connaitre le type du ou des objets généré par la cmdlet précédant le pipe, leurs méthodes et leurs propriétés.

```
<cmdlet> | Get-Member
TypeName :
Name                      MemberType    Definition
----                      ----------    ----------
Property
method

#Ex : [voila un exemple avec des résultats partiels, car trop de lignes de résultats)
Get-Service | Get-Member

   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
Close                     Method        void Close()
Refresh                   Method        void Refresh()
Start                     Method        void Start(), void Start(string[] args)
Stop                      Method        void Stop()
CanPauseAndContinue       Property      bool CanPauseAndContinue {get;}
CanShutdown               Property      bool CanShutdown {get;}
CanStop                   Property      bool CanStop {get;}
DisplayName               Property      string DisplayName {get;set;}
ServiceName               Property      string ServiceName {get;set;}
ServiceType               Property      System.ServiceProcess.ServiceType ServiceType
StartType                 Property      System.ServiceProcess.ServiceStartMode
StartType Status                     Property
System.ServiceProcess.ServiceControllerStatus Status {get;}
```

> D'autres notions ne seront pas abordées dans ce cours : ScriptMethod, AliasProperty, Event…

2.3.2. Propriétés

Lorsqu'une cmdlet est exécutée, elle affiche des propriétés par défaut parmi la collection de propriétés qui compose l'objet.
Il existe plusieurs syntaxes en PowerShell pour afficher des propriétés.
Dans certains cas les entêtes seront visibles, dans d'autres non.
-property
Affiche nom de propriété et valeur

-expandproperty
N'affichera que le contenu de la propriété sans son nom.

Comment afficher des propriétés :

```
<cmdlet> | Select-Object < [ prop | prop1,prop2 ] >
#ex :
PS C:\>Get-Service
#la cmdlet affiche par defaut les proprietes Status,Name,Displayname
#afficher toutes les proprietes et leurs valeurs :
PS C:\>Get-Service | Select-object *
#afficher une propriete qui ne l'est pas par defaut en plus de name et status
PS C:\>Get-Service | Select-Object -Property name,status,starttype

#Cette syntaxe supprime les entetes
PS C:\>Get-Service | Select-Object -ExpandProperty name

#Une autre maniere existe pour afficher une proprieté sans l'entete :
PS C:\>(get-service).name
```

2.3.3. Méthodes

Une méthode est un jeu d'instructions qui permettent de modifier un objet.

`cmdlet | Get-Member` vous permet de savoir quelles sont les méthodes utilisables pour l'objet généré par cmdlet.
La syntaxe pour faire appel a une méthode en PowerShell est :

```
# pour objet de type string
< Objet >.< NomMethode >(valeur)
# pour objet autre que string
(< cmdlet >).< NomMethode >(valeur)
```

```
| get-member -force
```

Affiche des détails supplémentaires et nottamenent des informations sur les membres intrinsèques, des "méthodes cachées". Ce sont des méthodes qui sont valables pour tous les objets powerShell.

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_intrinsic_members?view#powershell-7.3

```
# ouvrir un notepad
PS C:\>notepad.exe

# utiliser la méthode kill pour fermer le process correspondant a notepad
PS C:\>(Get-Process notepad).kill()

# méthode alternative avec utilisation de variable
PS C:\>$notepad = Get-Process notepad
PS C:\>$notepad.Kill()
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_methods?view#powershell-7.3

Synthèse :

Complétez les définitions :
Types d'objet :
Propriétés :
Méthodes :

Quizz :

Donnez des exemples de types d'objets.
Donnez des exemples de propriétés d'objets.
Donnez des exemples de méthodes.

2.4. Mise à jour des fichiers d'aide

Un des atouts de PowerShell qui facilite son apprentissage est la qualité de l'aide proposé.+ Mettre a jour l'aide : `Update-Help`

Cette simple cmdlet vous permet de mettre a jour les fichiers d'aide de PowerShell.

```
update-Help
```

Il est possible de compléter cette cmdlet pour qu'elle puisse s'adapter a des contraintes architecturales, comme devoir se connecter a un depot local pour contrer l'impossibilité de se connecter au dépôt officiel Microsoft.

```
-SourcePath # précise le chemin du dépôt lorsqu'il est connu
-UIculture # précise la langue
-Credential # si le dépôt nécessite de s'authentifier
```

```
#Update-Help -SourcePath <path> -UICulture <en-us> -credential

PS C:\>Update-Help -SourcePath "c:\DepotPS" -UICulture en-us -credential
bipbip@labeni.lcl
```

Synthèse :

Update-Help

Quizz :

Comment faire pour mettre à jour à partir de sources locales ?

2.5. Trouver une cmdlet

`Get-Command` est la cmdlet que tout débutant se doit de connaitre, car elle lui permettra de trouver la cmdlet qui répondra a son besoin. Si tant est qu'il connaisse un peu d'anglais et qu'il fasse preuve d'un peu de logique.
Ses paramètres sont :
-name
-verb
-noun
-module

```
Get-Command <pattern>
```

il est possible d'utiliser des métacaractères comme " **pour la recherche.** " signifiant ici 0 ou n caractères

la cmdlet **Get-Verb** vous permet de lister l'ensemble des verbes présent dans PowerShell.

```
# chercher les commandes dont le verb est get et dont le nom service :
PS C:\>Get-Command -Verb get -noun service
# Resultat : Get-Service

# chercher les commandes dont le verb est get et dont le nom contient servi :
PS C:\>Get-Command -Verb get -noun *servi*
# Resultat :
PS C:\>Get-NetfirewallServiceFilter
PS C:\>Get-Service
```

Comment trouver la commande qui crée un utilisateur ? :

```
# Quel est le verb en anglais qui permet de creer ?
# get-verb : create, new, add pourrait correspondre
# Comment dit on utilisateur en anglais ? user
# Quel metacaractere peut servir de joker ? *
PS C:\>Get-Command -verb new -noun *users*
#OU Get-Command new-*users*
# Resultat :
PS C:\>
CommandType     Name                                               Version    Source
-----------     ----                                               -------    -------
Cmdlet          New-ADUser                                         1.0.0.0
ActiveDirectory
Cmdlet          New-LocalUser                                      1.0.0.0
Microsoft.PowerShell.LocalAccounts
Cmdlet          New-WinUserLanguageList                            2.0.0.0
```

```
International
# New-LocalUser parait le plus pertinent, comment en etre sur ?
# RTFM
PS C:\>Get-Help New-LocalUser
# Resultat :
PS C:\>
DESCRIPTION
    The `New-LocalUser` cmdlet creates a local user account.
# Felicitations vous avez trouvé votre premiere cmdlet !
```

Synthèse :

Get-Command [-module | -verb | -noun]

Quizz :

Quel verbe "anglais", avec PowerShell, me permet d'afficher des résultats ?
Quel verbe "anglais", avec PowerShell, me permet de modifier ?
Quelle cmdlet permet d'afficher l'état des services ?
Quelle cmdlet permet d'afficher la configuration réseau actuelle ?

2.6. Trouver de l'aide sur une cmdlet

Une fois la mise a jour des fichiers d'aide effectuée, il sera possible d'aller chercher de l'aide sur une cmdlet. Obtenir de l'aide sur une cmdlet : Get-Help < cmdlet >

```
# Afficher l'aide de la cmdlet get-service
PS C:\>Get-Help Get-Service
```

Différents paramètres sont à disposition :
`-ShowWindow`
Sans doute le paramètre qui apporte le plus de plus value : il ouvre l'aide dans une fenêtre en dehors de la console dans laquelle elle a été lancée, elle permet la recherche et le filtrage par paramètres.
`-Examples`
Affiche la section exemples uniquement.
`-Online`
Affiche la version en ligne de la page d'aide dans un navigateur internet.
`-Full`
Affiche toutes les sections.
`-Detailed`
Affiche le détail des sections.

```
# Afficher l'aide de la cmdlet get-service
PS C:\>Get-Help Get-Service -ShowWindow
```

Les alias de "Get-Help"; "**Man**" et "**Help**" sont disponibles.

Informations détaillées de l'aide PowerShell : https://learn.microsoft.com/fr-fr/powershell/scripting/learn/ps101/02-help-system?view=powershell-7.5
Comprendre la syntaxe présente dans l'aide :

```
PS C:\>Get-Help About_Command_Syntax
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_command_syntax?view#powershell-5.1

2.7. Trouver de l’aide sur un concept de fonctionnement

En plus de l’aide sur les cmdlets PowerShell fournit de la documentation sur ses concepts.
Ils permettent de trouver des informations de fond sur divers aspects de PowerShell, tels que la syntaxe du langage, les concepts de scripting, les politiques d’exécution…

```
#liste des fichiers d'aide traitant des concepts
PS C:\>Get-Help About_*
#exemple : aide sur le concept de la syntaxe des cmdlet
PS C:\>Get-Help About_Variables
```

Synthèse :

Get-Help [-ShowWindow | -Online] Get-Help About_*

Quizz :

Quelle cmdlet permet d’obtenir de l’aide d’une cmdlet PowerShell ?
Quelle sont les sections présentent dans l’aide PowerShell ?
Quelle cmdlet permet d’obtenir de l’aide sur un concept de fonctionnement de PowerShell ?
Quel cmdlet permet d’afficher l’aide dans une nouvelle fenêtre ?

2.8. Les stratégies d'exécution

La stratégie d'exécution est un élément de sécurité. Son rôle est d'empêcher des scripts d'être exécutés sans validation ou contrôle de sécurité. Elle a un rôle équivalent à l'UAC dans l'intention : empêcher une violation non souhaitée des règles de sécurité.

>Sur des systèmes d'exploitation **non-Windows** elle est **Unrestricted** et ne peut pas être modifiée, en réalité elle équivaut plus au fonctionnement en mode bypass, car ces systèmes d'exploitation n'intègrent pas les zones de sécurité Windows.

```
# Afficher la strategie d'execution actuelle
PS C:\>Get-ExecutionPolicy

# Affiche les strategies d'execution par étendue
PS C:\>Get-Executionpolicy -list

# Modifier la strategie d'execution
PS C:\>Set-ExecutionPolicy <[ AllSigned | Bypass | RemoteSigned | Restricted |
Undefined | Unrestricted ]>

# obtenir de l'aide sur le sujet :
PS C:\>Get-Help about_executionpolicy
```

Les options sont :

- **AllSigned** : Les scripts peuvent être exécutés, si ils sont signés par un fournisseur reconnu. Vous serez interrogé lorsque vous tenterez d'exécuter un script dont le fournisseur n'est pas encore connu et classifié en reconnu ou non-reconnu.
- **Bypass** : Rien n'est bloqué et il n'y a aucun avertissement. Utilisé dans des environnements qui possèdent déjà une sécurité propre.
- **Default** : Fixe le paramètre a sa valeur par défaut. **Restricted** pour les Windows Clients, **RemoteSigned** pour les Windows Serveurs.
- **RemoteSigned** : C'est l'option par défaut pour les versions serveur. Les scripts provenant d'internet peuvent s'exécuter si ils sont signés par un fournisseur reconnu. Pour les scripts non-signés, provenant d'internet, la cmdlet `Unblock-File` est utilisable. Ne requiers pas de signature pour les scripts provenant de cet ordinateur.
- **Restricted** : Stratégie par défaut pour les Clients Windows, ne permet pas l'exécution de scripts, ni de fichier de configuration, ni de profiles.
- **Undefined** : cette stratégie applique en fait la stratégie par défaut.
- **Unrestricted** : Stratégie par défaut pour les systèmes non-Windows, averti l'utilisateur avant d'exécuter un script ne provenant pas de cet ordinateur.

Sur les systèmes qui ne font pas la différence entre les chemins UNC et les chemins internet, les scripts identifiés par un chemin UNC ne seront pas autorisés à s'exécuter avec une stratégie RemoteSigned.

Les stratégies d'exécution ont une portée qui peut être : MachinePolicy, UserPolicy, Process, CurrentUser, and LocalMachine. LocalMachine est la portée par défaut quand on définit une stratégie d'exécution.

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view#powershell-7.3

Synthèse :

Get-ExecutionPolicy Set-ExecutionPolicy

Quizz :

Quelle cmdlet permet de connaitre l'état de la stratégie d'exécution ? Citez 3 des 6 états possibles pour la stratégie d'exécution ?

2.9. Notions de base sur les variables

2.9.1. Déclarer une variable

Une variable est un espace de stockage mémoire nommé, dans lequel on peut stocker de l'information dans le but de la réutiliser.
L'intérêt de la variable réside dans le fait d'appeler par le même nom une boite dont le contenu peut varier au fur et à mesure du traitement effectué.
Dans PowerShell il n'est pas obligatoire de déclarer une variable au début de votre script pour pouvoir l'utiliser.
Une variable se reconnait au fait que son nom soit précédé d'un `'$'`.
Elle va pouvoir contenir une ou plusieurs chaines de caractère, un ou des nombres, un ou des objets.

Pour déclarer une nouvelle variable, on peut soit utiliser la cmdlet appropriée, soit lui attribuer une valeur directement.

```
#declarer une variable :
PS C:\>New-Variable -name <NomDeLaVariable> -Value <valeur>
#ou plus simplement :
PS C:\>$<NomDeLaVariable> = <valeur>
```

2.9.2. Affichage de valeurs de propriétés et variable

```
# Affiche le contenu de la variable
PS C:\>$<NomDeLaVariable>
# Lorsque ma variable contient un objet avec des proprietees
PS C:\>$<NomDeLaVariable>.<NomDeLaPropriete>

#Ex :
PS C:\>$MaVariable = "ceci est ma variable"
#affiche le contenu de ma variable
PS C:\>Write-Host "$MaVariable"
#affiche le contenu de ma variable
PS C:\>$mavariable

#Lorsque ma variable contient un objet, par exemple ici une liste d'objet utilisateur
locaux
PS C:\>$MaVariable = Get-LocalUser
# Affiche la propriete name de chacun des objets du tableau
PS C:\>$MaVariable.name
```

2.9.3. Les variables tableaux

Les tableaux ont une propriété length ou count. La méthode `Clear` ne réinitialise pas la taille du tableau, elle définit toutes les valeurs sur la valeur par défaut.

```
PS C:\>$a = 1,2,3
PS C:\>$a.lenght #affiche 3, la longueur du tableau est de 3.
PS C:\>$a #affiche 1 2 3 avec des retours a la ligne
PS C:\>$a.clear()
PS C:\>$a # affiche rien
PS C:\>$a.length #affiche 3, la longueur du tableau n'a pas changé.
```

On parlera de "array", il existe plusieurs types de variables tableaux en powerShell : array, arraylist, Hashtable, genericlist.

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_arrays?view=powershell-7.5#count-or-length-or-longlength

Synthèse :

Une variable se reconnait au signe "$" qui précède son nom
variable locale
Variable d'environnement
Variable tableau

Quizz :

2.10. Afficher un message

`Write-Host` est la cmdlet qui permet d'afficher un résultat à l'écran. `-NoNewline` : ne fera pas de retour à la ligne `-Separator <Object>` : `-ForegroundColor <ConsoleColor>` `-BackgroundColor <ConsoleColor>`

```
#affichage a titre informatif
PS C:\>Write-Host "mon message"
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/write-host?view=powershell-5.1

2.11. Demander une saisie utilisateur

`read-host` permet de demander la saisir par l'utilisateur du script. '-prompt' est positionnel et optionnel, omis la plupart du temps

```
# a noter que la cmdlet read-host ajoutera automatiquement ":" a la fin de la phrase
PS C:\>$choix = read-host -prompt "quel message voulez vous entrer ? "
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/read-host?view=powershell-5.1

2.12. Nettoyer l'affichage

`Clear-Host` permet de nettoyer l'écran, autrement dit de 'reset' l'affichage à l'écran.

```
Clear-Host
```

Synthèse :

Write-Host : Affiche à l'écran
Read-Host : Demande une saisie utilisateur
Clear-Host : Reset l'affichage

Quizz :

Quelle instruction permet d'afficher le message "Hello World" ?
Quelle instruction permet de demander "Quel est votre choix ?" et de stocker le résultat dans la variable choix ?

2.13. Sélection et Filtrage

2.13.1. Sélectionner les propriétés d'un objet

La cmdlet `Select-Object` permet de modifier un objet en déterminant quelles sont les propriétés que l'on souhaite conserver sur celui-ci.

```
<cmdlet> | Select-Object <property>[,<property>] [ -Unique | -First | -Last |
-ExpandProperty ]
```

Ses principales fonctionnalités sont :

- Sélection de propriétés spécifiques, permets de choisir les propriétés à afficher : `Get-Process | Select-Object -Property Name, Id`
- Possibilité de renommer des propriétés pour plus de clarté : `Get-Service | Select-Object @{Name='ServiceName';Expression={$PSItem.Name}}, Status`
- Filtre les doublons avec le paramètre `-Unique`
- Limitation du nombre d'objets Sélectionne un nombre spécifique d'objets avec `-First` et `-Last`
- Expansion de propriétés Développe les propriétés contenant des objets ou des tableaux avec `-ExpandProperty`
- Création de propriétés calculées Permets de créer de nouvelles propriétés basées sur des expressions
- Optimisation des performances Arrête le traitement une fois le nombre d'objets requis atteint (sauf avec `-Wait`)
- Combinaison avec d'autres cmdlets S'utilise souvent en conjonction avec `Where-Object` pour un filtrage avancé
- Indexation Sélectionne des objets à des positions spécifiques dans un tableau avec `-Index`

`Select-Object` est essentiel pour affiner les résultats des commandes PowerShell, permettant une manipulation précise des données et une présentation claire des informations. Il s'utilise a travers un pipeline, les propriétés non sélectionnées ne pourront plus être utilisées après son exécution, car elles disparaitront de l'objet. Le choix de son placement dans un enchainement de pipeline a donc son importance.

```
# La commande suivante provoquera une erreur, expliquez pourquoi ?
PS C:\>Get-Service | Select-Object name | Where-Object status -like "stopped"
```

2.13.2. Filtrer un objet

Pour filtrer les résultats en PowerShell nous utiliserons la cmdlet `Where-Object` après un pipeline.
Sa Syntaxe est la suivante :

```
cmdlet1 | Where-Object -FilterScript { < cond1 > [-And \| -Or] < cond2 > }
```

Cette syntaxe fonctionne toujours en PowerShell, que j'ai une ou plusieurs conditions.

2.13.3. Opérateurs de comparaison

Les opérateurs de comparaison en Powershell sont les suivants :

Table 2. Operateur de comparaison

Operator	signifie	case sensitive
-eq	egal	-ceq
-ne	non egal	-cne
-gt	superieur a	-cgt
-ge	superieur ou egal	-cge
-lt	inferieur a	-clt
-le	inferieur ou egal	-cle
-Like	comparaison avec des jokers (meta)	-clike
-NotLike	inverse de like	-cnotlike
-Match	compare a string to regular expr	-cmatch
-NotMatch	inverse de match	-cnomatch
-Contains	test si la collection contient une valeur	
-NotContains	inverse de Contains	
-In	test si objet existe dans la collection	
-NotIn	inverse de in	
-Replace	remplace via regex	
-as	test si un objet est de type	

2.13.4. Filtrage simple

Le filtrage simple signifie que nous ne poserons qu'une seule condition lors du filtrage.
La syntaxe d'origine de la cmdlet `Where-Object` est la forme que nous verrons au point filtrage avancé, cette forme simplifiée est plus facile a appréhender, mais impose une limite d'une seule et

unique condition a respecter.

```
cmdlet1 | Where-Object < cond >

# Ex: obetnir la liste des services dont l'etat est en cours d'execution
PS C:\>Get-Service | Where-Object status -like "running"
```

2.13.5. Filtrage avancé

Le filtrage "avancé" possède une syntaxe un peut plus complète avec `-filterscript { }` elle peut être utilisée pour une ou plusieurs conditions. Autrement dit elle s'adaptera quoi que vous souhaitiez faire.

```
cmdlet1 | Where-Object [-FilterScript] { < cond1 > [-And \| -Or] < cond2 > }

# Afficher les services dont l'etat est running et dont le nom commence par "W"
PS C:\>Get-Service | Where-Object -FilterScript { $PSItem.status -like "running" -and
$PSItem.name -like "w*" }
```

-Object de Where-Object est optionnel
-FilterScript est optionnel

PowerShell se veut être le plus lisible possible faire l'omission volontaire de ces indications ne respecte pas les **bonnes pratiques**.

Synthèse :

cmdlet | Select-Object [-unique | -ExpandProperty]
cmdlet | Where-Object [-FilterScript] { < cond1 > [-And \| -Or] < cond2 > }

Quizz :

Quel est le résultat de la cmdlet `Get-Service | Select-Object *` ?
Pourquoi la cmdlet `Get-Service | Where-Object name -like "Spooler" -and status -like "stopped"` génere t'elle une erreur ?

2.14. Formats

2.14.1. Liste

Modifie l'affichage d'une cmdlet pour la présenter sous forme de liste.
`-property` : permets de choisir les propriétés à afficher, comme avec Select-Object.
`-groupby` : permets de regrouper les résultats à afficher.

```
<cmdlet> | Format-List
-property
-groupby

Get-Service | Select-Object -first 5 | Format-List -property status, name

Get-Service | Select-Object -first 5 | Sort-Object status | Format-List -GroupBy
status
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-list?view=powershell-7.5

2.14.2. Tableau

Modifie l'affichage d'une cmdlet pour la présenter sous forme de tableau.

```
<cmdlet> | Format-Table
-Autosize : redimensionne la largeur des colonnes et leur nombre
-RepeatHeader : repete les entete de colonne a chaque page/ecran
-HideTableHeaders : masque les entetes
-Wrap : permet de ne pas tronquer l'affichage avec des ...
-property : permet de n'afficher que certaines proprietes
-GroupBy : regroupe les données selon la valeur de proprietes

Get-Service | Select-Object -first 5 | Format-table -property status, name

Get-Service | Select-Object -first 5 | Format-table
Status   Name               DisplayName
------   ----               -----------
Running  AdobeARMservice    Adobe Acrobat Update Service
Stopped  AJRouter           Service de routeur AllJoyn
Stopped  ALG                Service de la passerelle de la couc...
Running  AppHostSvc         Application Host Helper Service
Stopped  AppIDSvc           Identité de l□application
# le displayname de ALG est tronqué

Get-Service | Select-Object -first 5 | Format-table -wrap

Status   Name               DisplayName
```

```
------   ----                -----------
Running  AdobeARMservice     Adobe Acrobat Update Service
Stopped  AJRouter            Service de routeur AllJoyn
Stopped  ALG                 Service de la passerelle de la couche
                             Application
Running  AppHostSvc          Application Host Helper Service
Stopped  AppIDSvc            Identité de l□application
# -wrap permet d'avoir un affichage complet avec le retour a la ligne.
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-table?view=powershell-7.5

2.14.3. Colonne

Modifie l'affichage d'une cmdlet pour la présenter sous forme de colonnes.

```
<cmdlet> | Format-Wide
-Property
-Autosize
-Column
-GroupBy

Get-Service | Format-wide -Column 5
```

Le traitement du formatage de résultat de cmdlet doit toujours se faire en fin de ligne, car celui-ci **change le type d'objet.**

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/format-wide?view=powershell-7.5

Synthèse :

| Format-List : résultat en liste
| Format-Table : résultat en tableau
| Format-Wide : résultat en colonne
Les cmdlets Format-X changent le type de l'objet.

Quizz :

Quelle cmdlet affiche les services sous forme de tableau ?
Quelle cmdlet affiche les services sous forme de liste ?
Quelle cmdlet affiche les services sous forme de colonne ?

2.15. Tri

La cmdlet `Sort-Object` permet de trier les résultats obtenus.
Par défaut la cmdlet tri par ordre croissant ou ordre alphabétique.

```
<cmdlet> | Sort-Object
# Tri des utilisateurs locaux en fonction de la valeur de la propriete enabled
PS C:\>Get-LocalUser | Sort-Obect -Property enabled
  -descending
  -unique
  -CaseSensitive

# affiche la propriete "status" pour tous les services
Get-Service | Select-Object status
# affiche sans doublons, les types de valeurs de "status" present dans la liste.
Get-Service | Select-Object status | Sort-Object -unique status
```

Synthèse :

| Sort-Object : permets de trier les résultats

Quizz :

Quelle cmdlet permet d'afficher les 5 fichiers les plus volumineux du répertoire courant dans l'ordre décroissant ?

2.16. Compter

La cmdlet `| Measure-Object` permet de faire des calculs sur les résultats.

-Average
-Character
-Line
-Maximum
-Minimum
-Property
-Sum
-Word

```
# sur une liste de fichiers
PS C:\>Get-ChildItem | Measure-Object -Property length -Minimum -Maximum -Sum -Average

PS C:\>Get-ChildItem | Measure-Object -Sum {$_.Length/1MB}

# dans un fichier texte
PS C:\>"One", "Two", "Three", "Four" | Set-Content -Path C:\Temp\tmp.txt

PS C:\>Get-Content C:\Temp\tmp.txt | Measure-Object -Character -Line -Word
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-object?view#powershell-7.3

Il est également possible d'utiliser :

```
# comptera ce qui compose la variable var.
PS C:\>$var.count
```

Synthèse :

| Measure-Object

Quizz :

Quelle cmdlet permet d'afficher la moyenne du poids des fichiers du répertoire courant ?

2.17. Horodatage

La cmdlet `Get-Date` est la cmdlet qui permet d'avoir la date et l'heure sous PowerShell. Pour un administrateur système, il est primordial de maitriser les commandes permettant de créer l'horodatage.

```
#affichage de la date
PS C:\>Get-Date
PS C:\>lundi 1 janvier 0001 00:00:00
#utilisation d'une méthode pour ajouter 2 jours
PS C:\>(Get-Date).AddDays(2)
PS C:\>mercredi 3 janvier 0001 00:00:00
#format personnalisé
PS C:\>Get-Date -Format "yyyy-MM-dd HH:mm:ss"
#recuperation de l'année uniquement
PS C:\>(Get-Date).Year
#au format Unix
PS C:\>Get-Date -UFormat %D
```

-UFormat permet d'utiliser les balises du bash %y %d %m etc... ce paramètre permet plus de liberté dans la présentation
Liste complete des balises pour UFormat

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/get-date?view=powershell-7.5

Synthèse :

Get-Date : permets de gérer la date
Le format Unix est possible.

Quizz :

Quelle cmdlet me permet d'afficher la date ?
Quelle cmdlet me permet d'afficher la date au format " JJ/MM/AA HH:MM" ?
Quelle cmdlet me permet d'afficher la date précise dans 3600 heures ?
Pourquoi est-il important pour un sys de maitriser la génération de date ?

2.18. Log et historique de commande

- Log

PowerShell intégré les cmdlets `Start-Transcript` et `Stop-Transcript` qui permettent de démarrer un fichier de log qui va récupérer l'ensemble des retours du script. Cette méthode est pratique, mais sans filtre. La cmdlet `Out-File` quant a elle permet d'être plus précis dans ce que l'on souhaite conserver.

```
Start-Transcript -path <FilePath> [-append]

Stop-Transcript
## OU
| Out-File
```

- Historique

Lorsque vous entrez une commande à l'invite de commandes, PowerShell enregistre la commande dans l'historique des commandes.
PowerShell a deux fournisseurs d'historique différents : l'historique intégré et l'historique géré par le module PSReadLine .

Cmdlet d'historique.

Applet de commande	**Alias**	**Description**
Get-History	h	Obtient l'historique des commandes.
Invoke-History	r	Exécute une commande dans l'historique des commandes.
Add-History		Ajoute une commande à l'historique des commandes.
Clear-History	clhy	Supprime les commandes de l'historique des commandes.

Interagir avec l'historique au clavier
- UpArrow : affiche la commande précédente.
- DownArrow : affiche la commande suivante.
- F7 : affiche l'historique des commandes.
- ESC - Pour masquer l'historique.
- F8 - Recherche une commande. Tapez un ou plusieurs caractères, puis appuyez sur F8. Appuyez de nouveau sur F8 pour l'instance suivante.
- F9 : recherchez une commande par ID d'historique. Tapez l'ID d'historique, puis appuyez sur F9. Appuyez sur F7 pour rechercher l'ID.
- # <string> Onglet : recherchez et <**string**> retournez la correspondance la plus récente. Si vous appuyez sur Tab à plusieurs reprises, il parcourt les éléments correspondants dans votre historique.

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_history?view=powershell-7.5

Synthèse :

Start-Transcript -path <FilePath> -append : démarre le log dans le fichier

Stop-Transcript : arrête le log

Quizz :

Quel est l'intérêt de garder une trace de ce qui est effectué ?
Puis je le faire dans un script ?
Puis je le faire dans une CLI ?
Quels sont les 2 mécanismes qui me permettent de gérer des logs avec PowerShell ?

2.19. Les cmdlet Active Directory

Le module Active Directory n'est disponible que sur les serveurs ou le rôle AD DS a été installé.
Ces cmdlets doivent s'interfacer avec un annuaire de type LDAP et ont donc des particularités comme le fait d'utiliser le `-filter` pour la cmdlet `Get-ADuser`

```
Get-ADuser -Filter * -Properties <prop>

PS C:\>Get-ADUser -Identity "AxelRios" -Properties * | Set-ADUser -Description repouet
PS C:\>Get-ADUser -Identity "AxelRios"
```

Attention IDENTITY correspond au SamAccountName.
https://learn.microsoft.com/en-us/powershell/module/activedirectory/?view=windowsserver2025-ps
https://learn.microsoft.com/en-us/powershell/module/activedirectory/get-aduser?view=windowsserver2025-ps

Synthèse :

Le module Active Direcory de PowerShell n'est disponible que sur le serveur ou le rôle AD DS a été installé.
Les cmdlet sont de type <verb>-AD<noun>

Quizz :

Quelle cmdlet me permet d'avoir la liste complète de toutes les cmdlets Active Directory ?
Quelle cmdlet me permet d'avoir la liste des utilisateurs Active Directory ?
Quelle cmdlet me permet d'avoir la liste des utilisateurs Active Directory du service Informatique uniquement ?

2.20. L'import / Export de données

PowerShell offre des fonctionnalités puissantes pour l'import et l'export de données, facilitant la manipulation et l'analyse des informations dans divers formats.

2.20.1. Import de données

PowerShell permet d'importer des données à partir de plusieurs types de fichiers :

- Fichiers CSV : Utilisez la cmdlet `Import-Csv` pour importer des données à partir de fichiers CSV.
- Fichiers XML : La cmdlet `Import-Clixml` permet d'importer des données XML.
- Fichiers JSON : Utilisez `ConvertFrom-Json` pour traiter des données JSON.
- Fichiers PowerShell Data (.psd1) : La cmdlet `Import-PowerShellDataFile` importe en toute sécurité des paires clé-valeur à partir de fichiers .psd.
- Fichiers Excel : Avec le module ImportExcel, vous pouvez importer des données depuis des fichiers Excel8.

2.20.2. Export de données

Pour l'export de données, PowerShell propose plusieurs options :

- Fichiers CSV : La cmdlet `Export-Csv` permet de convertir des objets en format CSV47.
- Fichiers XML : Utilisez `Export-Clixml` pour exporter des données au format XML.
- Fichiers JSON : La cmdlet `ConvertTo-Json` permet de convertir des objets en format JSON.

Personnalisation et options avancées

- Vous pouvez spécifier des délimiteurs personnalisés lors de l'export CSV avec le paramètre `-Delimiter`.
- L'option `-NoTypeInformation` permet d'omettre les informations de type dans les fichiers CSV exportés.
- Pour les fichiers Excel, vous pouvez sélectionner des colonnes spécifiques à importer ou exporter.

Synthèse :

Import-CSV
Import-CliXML
Export-CSV
Export-CliXML
ConvertTo-Json ... | Out-File <file>
ConvertTo-HTML | Out-File <file>

Quizz :

3. Scripting

Avant toutes choses il convient de s'assurer que le besoin est : - réel
- bien compris
- validé par le demandeur et l'exécutant

3.1. La construction d'un script

Méthode : algo → blocs structurés → recherche des commandes nécessaire (sans la syntaxe exacte) → création du fichier de script en .PS1, ajout du bloc d'aide intégrant le cartouche et insertion des commentaires depuis l'algo → création de la structure avec uniquement de l'affichage et des tests (if|while|foreach…/Write-Host) → insertion des cmdlets de traitement → validation → optimisation (gestion d'erreur, logs…) → validation

3.1.1. Les consoles et outils de scripting

applications	**remarques**
Le CLI	permets de lancer des commandes en direct et des scripts
PowerShell ISE	intégré a Windows limité a un langage
Visual Studio Code	open source multiplateforme, multilangages
VS Codium	idem sans couche Microsoft

3.1.2. L'algorithmie

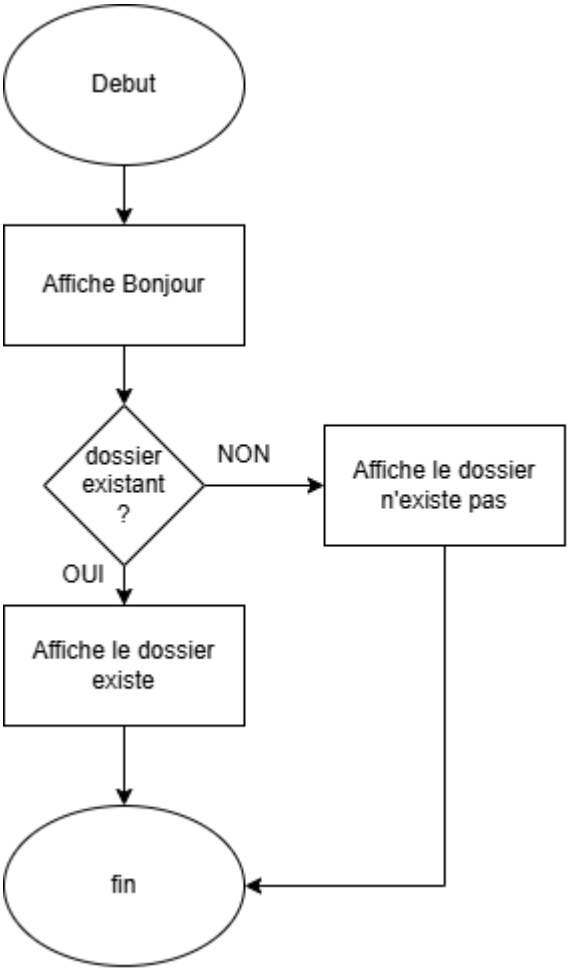
Un algorithme est la description d'une suite d'étapes permettant d'obtenir un résultat à partir d'éléments fournis en entrée. En développement, il permet de schématiser le fonctionnement souhaité d'un programme.

- Planification : Il permet de visualiser la structure logique du script avant de commencer à coder, aidant à identifier les étapes clés et les flux de contrôle.
- Clarification : Il simplifie la compréhension des processus complexes en les décomposant en étapes plus petites et plus gérables.
- Débogage : Il facilite l'identification des erreurs potentielles dans la logique du script avant même de commencer à coder.
- Documentation : Il sert de documentation visuelle du script, utile pour la maintenance future et pour expliquer le fonctionnement du script à d'autres personnes.
- Optimisation : Il aide à identifier les inefficacités potentielles dans le flux du script, permettant d'optimiser la structure avant l'implémentation.

les symboles :

- Rond : pour le début et la fin du script.
- Rectangle : pour les actions.

- Losange : pour les tests.



3.1.3. Les commentaires

```
# commente une ligne
<cmdlet> #commentaire possible en fin de ligne
<#
commente un bloc
#>
```

- Commenter un script, bonne pratique ou perte de temps ?
  Pour le vous du futur.
  Pour faciliter le fait, de maintenir le script a l'avenir.
  Pour permettre à vos collègues de monter en compétences.
- le cartouche Au début de chaque script, la bonne pratique veut que l'on retrouve des informations relatives à la vie du script.
  Par Exemple :

```
Nom du script +
Version du script +
Auteur +
Date de création +
Dernier modificateur +
```

```
Date de derniere modification ``
```

- l'aide intégrée au script PowerShell permet au créateur du script d'intégrer des balises commentées en début de script qui peuvent être interprétées par la cmdlet `Get-Help`.
  Certaines se complète automatiquement :
  .Nom
  .Syntaxe
  .Liste de paramètres
  .Paramètres communs
  .Table d'attributs de paramètre
  .Remarques

Les autres sont :
.SYNOPSIS
.DESCRIPTION
.PARAMETER
.EXAMPLE
.INPUTS
.OUTPUTS
.NOTES
.LINK
.COMPONENT
.ROLE
.FUNCTIONALITY
.FORWARDHELPTARGETNAME
.FORWARDHELPCATEGORY
.REMOTEHELPRUNSPACE
.EXTERNALHELP

Si j'insère ce bloc de code au début du script alors je rends possible l'utilisation de `Get-Help monscript.ps1`. Tous les paramètres de `Get-Help` seront utilisables.

```
<#
.SYNOPSIS
auteur :
date de création :
dernier modificateur :
date de derniere modif :
.DESCRIPTION
un script qui dit bonjour
.EXAMPLE
exemple1 : .\demohelp.ps1
.EXAMPLE
exemple 2 : l'exemple 2
.LINK
https://learn.microsoft.com/fr-
fr/powershell/module/microsoft.powershell.core/about/about_comment_based_help?view=pow
ershell-7.3
```

```
.NOTES
Les notes apparaissent lors de l'usage de get-help avec -full
#>
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_comment_based_help?view=powershell-7.4&viewFallbackFrom=powershell-7.3

! Attention ! Si la stratégie d'exécution ne permet pas d'exécuter un script, il n'est pas possible de faire un `get-help .\monscript.ps1`, même si celui-ci contient les bonnes balises.

Synthèse :

Il est possible en ajoutant au début du script un bloc de code pour faire en sorte que celui-ci soit compatible avec la cmdlet Get-Help.
Les commentaires servent a :
permettre une relecture et une maintenance plus efficace.
faciliter le début en insérer des lignes de contrôles (ex : afficher les valeurs de variables)

3.2. Les tests

Un test conditionnel est une expression qui se résout en une valeur booléenne : vrai ou faux ('true' ou 'false').
Ces tests conditionnels contrôlent le flux de l'exécution, en permettant d'exécuter des blocs de code uniquement si la condition est vérifiée (true) ou pas (false) (IF, SWITCH), ou de répéter des actions tant que (while) ou jusqu'à (do until) ce que la condition change.
En PowerShell on utilisera des tests avec des opérateurs de comparaison (-eq, -lt, ...), des opérateurs logiques (-and, -or, -not...), des opérations booléennes (présence d'un objet, existence d'un fichier/répertoire)

```
PS >10 -gt 100
PS >False

PS >"test" -like "test"
PS >True

PS >Test-Path ".\fic.txt"
PS >True
```

Dans le cas où le test est intégré à une structure, le code retour ('$?') déterminera si la condition est vraie ou fausse.

```
# Demonstration
# un if condition avec affichage du code retour
$nombre = 15
if ($nombre -gt 10)
{
Write-Host "le code retour est `$? = $?"
Write-Host "$nombre est plus grand que 10"
}
PS >le code retour est $? = True
PS >15 est plus grand que 10


$nombre = 5
if ($nombre -gt 10)
{
Write-Host "le code retour est `$? = $?"
Write-Host "$nombre est plus grand que 10" }
else
{
Write-Host "le code retour est `$? = $?"
Write-Host "$nombre est plus petit que 10"
}
PS >le code retour est $? = False
PS >5 est plus petit que 10
```

```
# exemple avec une commande
if (test-path ".\script.ps1") {Write-Host "le fichier existe"} else {Write-Host "le
fichier n'existe pas"}
```

```
# Attention a la facilité de PowerShell
# 042 est converti en 'int'
PS >42 -eq "042"
PS >True
# 042 est une chaine de caractere 'string'
PS >"042" -eq 42
PS >False
PS >42 -like "042"
PS >False
```

Le "typage" est une notion qui sera abordée plus loin dans le document.

3.3. Structures

PowerShell dispose de plusieurs structures pour effectuer des traitements selon conditions.

3.3.1. IF

Exécute un bloc de code si une condition est vraie.
Peut être combiné avec else et elseif pour des conditions multiples.

```
#Syntaxe
if (<test1>)
    {<statement list 1>}
[elseif (<test2>)
    {<statement list 2>}]
[else
    {<statement list 3>}]

# Exemple :
$x = 10
if ($x -gt 5) {
    Write-Host "La valeur de x est supérieure à 5."
}
else {
    Write-Host "La valeur de x est inférieure ou égale à 5."
}
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_if?view=powershell-7.4

3.3.2. SWITCH

Alternative à plusieurs if-elseif pour comparer une valeur à plusieurs cas.
Permets d'exécuter différents blocs de code selon la valeur testée, souvent lorsque l'on connait les cas de figure possibles et pour des menus.
`-Exact` : Utilisé par défaut, correspondance sans casse.
`-CaseSensitive` :
`-Wildcard` :
`-Regex` :
`-File` :

`default` : La clause default est déclenchée lorsque la valeur ne correspond à aucune des conditions. Ne peut être présente qu'une fois.

```
#Syntaxe
Switch (<test-expression>)
{
    <result1-to-be-matched> {<action>}
    <result2-to-be-matched> {<action>}
}

# Exemple :
$value = 2

switch ($value) {
    1 { Write-Host "La valeur est 1" }
    2 { Write-Host "La valeur est 2" }
    3 { Write-Host "La valeur est 3" }
    Default { Write-Host "La valeur n'est pas dans la liste" }
}
PS >La valeur est 2

# Exemple -wildcard:
$value = 123

switch -Wildcard ($value) {
    *1* { Write-Host "La valeur est 1" }
    2 { Write-Host "La valeur est 2" }
    3 { Write-Host "La valeur est 3" }
    Default { Write-Host "La valeur n'est pas dans la liste" }
}
PS >"La valeur est 1"

#Exemple -File
$value = "C:\temp\tmp.txt"
#le fichier contient 1 et one sur deux lignes

switch -file ($value) {
    1 { Write-Host "La valeur est 1" }
    2 { Write-Host "La valeur est 2" }
```

```
    3 { Write-Host "La valeur est 3" }
    Default { Write-Host "La valeur n'est pas dans la liste" }
}
PS >La valeur est 1
PS >La valeur n'est pas dans la liste
```

Le SWITCH a la différence du CASE en Bash ne sort pas après une condition vraie

```
$value = 15
switch -Wildcard ($value) {
    { $value -gt 10 } {
        Write-Host "$PSItem est supérieur à 10"
        # pas de break, on continue à tester les autres cas
    }
    { $value % 3 -eq 0 } {
        Write-Host "$PSItem est un multiple de 3"
    }
    { $value -lt 20 } {
        Write-Host "$PSItem est inférieur à 20"
    }
    Default {
        Write-Host "$PSItem ne correspond à aucune condition spécifique"
    }
}
PS >15 est supérieur à 10
PS >15 est un multiple de 3
PS >15 est inférieur à 20
# Defaut n'est pas executé car une des valeurs precedentes était vraie.
# Si je veux me rapprocher du mode de fonctionnement de CASE en bash je devrais
utiliser 'break' dans mes bloc d'actions, qui aura pour effet de quitter le switch.
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_switch?view=powershell-7.5

3.3.3. WHILE

Exécute un bloc de code tant qu'une condition est vraie.
Vérifie la condition avant chaque itération.

```
#Syntaxe
while (<condition>)
{
    <statement list>
}

# Exemple :
$val=0
while($val -ne 3)
{
    $val++
    Write-Host $val
}
```

3.3.4. DO WHILE

Similaire à while, mais exécute le code au moins une fois.
Vérifie la condition après chaque itération.

```
do {
<statement list>
} while (<condition>)

    Do # fait
    {
        Write-Host "1) Affichage des ordinateurs du domaine."
        Write-Host "2) Affichage des groupes de domaines locaux"
        Write-Host "3) Import des utilisateurs « AD » à partir d'un fichiers « CSV »"
        Write-Host "4) Quitter"
            $choix # Read-Host "Bienvenue dans l'outil d'inventaire, faîtes votre
choix parmi les menus suivants "
            Switch($choix)
            {
                "1" {write-host "choix 1"}
                "2" {write-host "choix 2"}
                "3" {write-host "choix 3"}
                "4" {exit}
            }
    # jusqu'a ce que le contenu de choix ne soit pas egal a 4
    } While ($choix -ne '4')
```

3.3.5. DO UNTIL

Exécute un bloc de code jusqu'à ce qu'une condition devienne vraie.
Vérifie la condition après chaque itération.

```
do {
<statement list>
} until (<condition>)

    Do # fait
    {
        Write-Host "1) Affichage des ordinateurs du domaine."
        Write-Host "2) Affichage des groupes de domaines locaux"
        Write-Host "3) Import des utilisateurs « AD » à partir d'un fichiers « CSV »"
        Write-Host "4) Quitter"
            $choix # Read-Host "Bienvenue dans l'outil d'inventaire, faîtes votre
choix parmi les menus suivants "
            Switch($choix)
            {
                "1" {write-host "choix 1"}
                "2" {write-host "choix 2"}
                "3" {write-host "choix 3"}
                "4" {exit}
            }
    # tant que le contenu de choix est egal a 4
    } until ($choix -eq '4')
```

3.3.6. FOREACH

Fait une itération sur chaque élément d'une collection.
Très utilisé pour traiter des tableaux ou des listes d'objets.

```
foreach ($<item> in $<collection>)
{
<statement list>
}

$letterArray = 'a','b','c','d'
foreach ($letter in $letterArray)
{
  Write-Host $letter
}
```

3.3.7. FOR

Boucle avec un compteur, utile quand le nombre d'itérations est connu.
Permet un contrôle précis sur l'initialisation, la condition et l'incrémentation.

```
for (<Init>; <Condition>; <Repeat>)
{
    <Statement list>
}

nombre=0
max=10
for ($i = 1; $i -le $max; $i++) {
    $resultat = $nombre + 1
    Write-Host "$nombre `+ 1 = $résultat"
}
```

Faites vos propres essais simples pour tester le comportement des structures, partez des exemples et explorez.

Synthèse :

Les structures de contrôle à disposition en PowerShell sont :
IF
Switch
While
Do While
Do Until
Foreach
For

Quizz :

4. Synthèse des notions essentielles

Voici la boite à outils du survivaliste PowerShell, vous l'emmènerez lors de vos voyages dans cette zone pas si hostile ou règne PowerShell. Complétez la colonne "commentaire" avec vos mots pour valider que vous maitrisez ces notions et leurs définitions.

Cmdlet	**commentaire**
notions indispensables	
Update-Help	
Get-Command	
Get-Help	
Get-Help About_*	
| Get-Member	
| Select-Object <prop>	
| Where-Object <cond>	
| Where-Object -FilterScript { < cond1 > [-And | -Or] < cond2 > }	
Get-ExecutionPolicy	
notions plus avancées	
$var	
$var.< prop >	
(Get-Service).ServiceName	
(Get-date).adddays(7)	
Sort-Object	
Measure-Object	
Format-List	
Format-Table	
Format-Wide	
Get-PSDrive	
Get-Help About_providers	
Get-Help About_profiles	
Les cmdlets AD	
Get-ADuser -Filter * -Properties < prop>	

Quizz :

5. Notions avancées

5.1. Profil utilisateur

Les profils utilisateur en PowerShell sont des scripts qui permettent de personnaliser l'environnement PowerShell pour chaque utilisateur.
Types de profils pour la CLI :
Tous les utilisateurs, hôte actuel - $PSHOME\Microsoft_profile.ps1
Utilisateur actuel, hôte actuel - $HOME\Documents\PowerShell\Microsoft_profile.ps1
Pour VSCode :
Tous les utilisateurs, hôte actuel - $PSHOME\Microsoft.VSCode_profile.ps1
Utilisateur actuel, hôte actuel - $HOME\Documents\PowerShell\Microsoft.VSCode_profile.ps1

Création et emplacement
Par défaut, aucun profil n'est créé
Exemple de création pour l'utilisateur actuel :

```
PS C:\>if (!(Test-Path -Path $PROFILE)) { New-Item -ItemType File -Path $PROFILE
-Force }
```

Ce profil permet de personnaliser l'environnement PowerShell : modifier le prompt, ajouter des commandes, alias, fonctions, variables, modules, automatiser le chargement de modules, configurer des préférences utilisateur.

`Get-Help About_profiles`

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_prompts?view=powershell-7.5

D'autres programmes qui hébergent PowerShell peuvent prendre en charge leurs propres profils. Par exemple, Visual Studio Code (VS Code) prend en charge les profils spécifiques à l'hôte.

Tous les utilisateurs, hôte actuel - $PSHOME\Microsoft.VSCode_profile.ps1
Utilisateur actuel, hôte actuel - $HOME\Documents\PowerShell\Microsoft.VSCode_profile.ps1

Synthèse :

Quizz :

5.2. Les modules

PowerShell est un langage modulaire, cela permet de limiter le nombre de cmdlet a embarquer lors d'une première installation. D'autres parts le fait de charger les modules contenant les cmdlets dont vous avez besoin permet d'adapter de manière granulaire powerShell a vos besoins. Un module contient des cmdlets, des providers, des fonctions, des variables, des alias.

Les modules sont stockés dans des emplacements spécifiques du système, ces emplacements sont stockés dans la variable système **$env:PSModulePath**. Chaque module a son propre dossier a son nom. Le point d'arborescence dans lequel vous déciderez d'installer le module déterminera pour qui il sera disponible. Il est possible d'automatiser son chargement avec l'utilisation du profil powerShell.

```
PS C:\>$env:PSModulePath
C:\Users\<UserName>\Documents\WindowsPowerShell\Modules;
C:\ProgramFiles\WindowsPowerShell\Modules;
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules
```

5.2.1. Installer un module

Un module peut être installé à partir de sources locales ou à partir de dépôts.
Le dépôt, par défaut, pour PowerShell est la PSGallery.

```
# connaitre les depots declares
PS C:\>Get-PSRepository
# chercher un module dans les depots declares
PS C:\>Find-Module <ModuleName>
# une fois le nom exact trouve on rappel la commande que l'on pipe dans un install
PS C:\>Find-Module <ModuleName> | Install-Module
# Telecharge le module et l'installe
PS C:\>Install-Module -Name <ModuleName> -Scope <[CurrentUser | AllUsers]>
```

Il est possible d'installer un module ou on le souhaite dans l'arborescence, néanmoins respecter les chemins proposés par défaut permettent de faciliter la maintenance des modules.

5.2.2. Importer un module

Comment trouver les modules installés ?

```
# obtenir la liste des modules disponibles sur l'OS
PS C:\>Get-Module -ListAvailable
# obtenir les liste des modules chargés dans le CLI PowerShell
PS C:\>Get-Module
```

Comment charger un module dans votre CLI ?

```
Import-Module <ModuleName>
```

5.3. Module (fonctionnement et création)

Différentes cmdlet sont a notre disposition pour la gestion des modules :

cmdlet	description
Find-Module	trouver un module dans les dépôts validés
Install-Module	installer un module
Import-Module	charger un module dans la session powerShell actuelle
Save-Module	sauvegarder localement un module
Update-Module	mettre a jour un module
Uninstall-Module	désinstaller un module de la machine
Remove-Module	retire le module de la session powerShell actuelle

Une variable d'environnement contient les chemins dans lequel PowerShell va chercher les modules disponibles pour l'utilisateur connecté spécifiquement ou globalement pour les utilisateurs de la machine.

```
#variable d'environnement pour les chemins des modules :
PS C:\>$env:PSModulePath
C:\Users\<utilisateur>\Documents\WindowsPowerShell\Modules;
C:\Program Files\WindowsPowerShell\Modules;
C:\Windows\system32\WindowsPowerShell\v1.0\Modules

#tester si le chemin existe sinon le creer
if (!(test-path "$env:HOMEPATH\Documents\WindowsPowerShell\Modules")){new-item -type
file -path "$env:HOMEPATH\Documents\WindowsPowerShell\Modules" -force}
```

Le répertoire contenu dans le profil utilisateur permettra de rendre disponibles les modules uniquement pour lui. Les répertoires contenus dans 'program files' et 'windows' seront disponibles pour tous les utilisateurs de la machine.

Installer un module, créé le répertoire a son nom dans le répertoire 'modules'. Il pourra par la suite être importé quand l'utilisateur en fera la demande. L'import ne se fait pas automatiquement a chaque démarrage de session powerShell. Si c'est le souhait de l'utilisateur alors il devra modifier son profil powerShell et y intégrer les cmdlets d'import des modules souhaites.

https://learn.microsoft.com/fr-fr/powershell/scripting/learn/ps101/10-script-modules?view#powershell-7.3

Synthèse :

Quizz :

5.4. Notions avancées sur les variables

5.4.1. Les types de variables

Avec PowerShell il n'est pas obligatoire de typer les variables lors de leurs déclarations. Les variables peuvent avoir des types différents : string, int, bool, array...

Attention tout de même, il est parfois "plus facile", "pratique" de laisser faire le système, il ne faut pas néanmoins oublier que celui-ci applique purement et simplement ce qu'on lui demande de faire. Dans certains cas il pourrait appliquer un nouveau type lors d'enchainement de traitement et donc provoquer des erreurs dont le début ne sera pas intuitif.

```
#variable de type int
PS C:\>[int]$nombre = 1
PS C:\>$nombre.gettype()
# variable de type string
PS C:\>[string]$chaine = "Bonjour"
PS C:\>$chaine.gettype()
# variable de type bool
PS C:\>[bool]$bool = $true
PS C:\>$bool.gettype()
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_variables?view=powershell-7.3

5.4.2. Les variables automatiques

```
# Premier jeton de la derniere ligne recue par la session
$^
# Dernier jeton de la dernière ligne reçue par la session.
$$
# Status d'exécution de la dernière commande.
$?
# tableau d'objets d'erreur qui représentent les erreurs les plus récentes. L'erreur la
plus récente est le premier objet d'erreur dans le tableau `$Error[0]`.
$error
# chemin d'accès complet du répertoire de base de l'utilisateur.
$HOME
# objet qui représente l'application hôte actuelle pour PowerShell.
$HOST
# objet path qui représente le chemin d'accès complet de l'emplacement du répertoire
actif pour l'espace d'exécution PowerShell actuel.
$PWD
# Tableau de valeurs pour les paramètres non déclarés qui sont passés à une fonction,
un script ou un bloc de script.
$args
```

```
#Booleens
$false
$true
# Variable null
$null
```

Le cas particulier de $PSItem : $PSItem vs $_

> They are exactly the same. In PowerShell 2, when we use $_, it started to cause some confusion, so when PowerShell 3 was released the creators of PowerShell decided, “Let’s go ahead and rename it, or provide an alternate name for it, PSItem.” For some individuals, it’s a little more clear as to what it represents.
>
> — Rūmī

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_psitem?view=powershell-7.5

Autrement dit $_ est un alias pour $PSItem qui a été implémenté en v3 pour "faciliter" la lecture, ce qui est, pour rappel, un des principes fondateur de PowerShell. Ce n’est pas interdit de l’utiliser, ca ne met pas en péril un script, ils font la même chose , c’est juste une **mauvaise pratique**.

Lorsqu’un script est appelé avec à PowerShell , $LASTEXITCODE est défini sur :

- 1 si le script s’est arrêté en raison d’une exception ou si le résultat de la dernière commande a la valeur `$false`
- 0 si le script s’est terminé correctement et que le résultat de la dernière commande est défini sur `$true `

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_automatic_variables?view#powershell-7.3

5.4.3. Les variables tableaux

Les tableaux ont une propriété length ou count. La méthode Clear ne réinitialise pas la taille du tableau, elle définit toutes les valeurs sur la valeur par défaut.

```
PS C:\>$a = 1,2,3
PS C:\>$a.lenght #affiche 3, la longueur du tableau est de 3.
PS C:\>$a #affiche 1 2 3 avec des retours a la ligne
PS C:\>$a.clear()
PS C:\>$a # affiche rien
PS C:\>$a.length #affiche 3, la longueur du tableau n'a pas changé.
```

- Array

```
PS C:\>$array = "PC1,PC2,PC3"
PS C:\>$array.gettype()
```

le type Array n'autorise pas la modification, Il n'existe aucune méthode add ou remove. PowerShell "bidouille" ; en réalité il détruit le tableau pour le recréer.

- ArrayList

```
PS C:\>[System.Collections.ArrayList]$dept # (" ","Rennes","Angers","Quimper")

PS C:\>$dept.remove($dept[0]) #supprime la premiere ligne du tableau index 0, la ligne
vide.

PS C:\>$dept.removeat(0) #supprime la premiere ligne du tableau index 0, la ligne
vide.

PS C:\>$dept.add("Angers") #ajoute angers comme enregistrement dans le tableau
```

les méthodes pour les tableaux de type ArrayList sont :
- add()
Ajoute une valeur au tableau et affiche l'index du tableau contenant l'ajout.
- remove()
Supprime une valeur du tableau.
Dans les parenthèses est attendue une valeur du tableau
- removeat()
Supprime une valeur du tableau en fonction de son index
- count()

- Hashtable

Une table de hachage ou tableau associatif est une collection reposant sur la notion clé valeur.
il sera possible de rappeler une valeur par rapport a la clé associé dans le tableau lors de son insertion.

```
PS C:\>[HashTable]$hash = @{ "S1" = "Windows" ; "S2" = "Linux" ; "S3" = "Admin Linux"}
PS C:\>[HashTable]$hasho = [ordered]@{ "S1" = "Windows" ; "S2" = "Linux" ; "S3" =
"Admin Linux"}
PS C:\>$hash.keys #affiche S1 S2 S3
PS C:\>$hash.values #affiche Windows Linux Admin Linux
PS C:\>$hash.S1 #affiche Windows
PS C:\>$hash.S3 #affiche Admin Linux

#ajout de valeur au tableau
PS C:\>$hash["S4"] = "Gestion de projet"
PS C:\>$hash.add( "S5" , "MSP" )
```

```
PS C:\>$hash

# utilisation de la méthode foreach
PS C:\>$hash.ForEach({"The value of '$($_.Keys)' is: $($_.Values)"})
```

- Genericlist

Il s'agit d'un type spécifique de tableau emprunté au C#, mais utilisable en PowerShell. Ce type de tableau a la particularité de spécifier le type de données qu'ils vont stocker.

```
# création d'une liste de string
PS C:\>$mylist = [System.Collections.Generic.List[string]]::new()
# création d'une liste d'entier
PS C:\>$mylist = [System.Collections.Generic.List[int]]::new()
# création sans initialisation :
PS C:\>$mylist = [System.Collections.Generic.List[int]]@(1,2,3)
PS C:\>$myList.Add(10) #pas d'affichage ecran
```

Il est possible de créer une liste pouvant accepter n'importe quel type de donnée

```
# comment le creer
PS C:\>using namespace System.Collections.Generic
PS C:\>$list = [List[PSObject]]::new()
# ou
PS C:\>$myList = [List[string]]@('Zero','One','Two','Three')

# dans le cas de l'utilisation du remove sans le [void] il affiche true ou false
PS C:\>[void]$myList.Remove("Two")
```

System.Collections est apparu en premier, il accepte tout type d'objet. System.Collections.Generic est apparu bien plus tard, il permet de spécifier le type d'objet accepté par la collection. Son utilisation et le fait de connaitre précisément le type de données permettent d'optimiser le traitement, les traitements spécifiques aux autres types ne seront pas envisagés par le moteur. Add() ne renvoie pas d'information à l'écran.

A noter qu'a la différence du ArrayList l'utilisation de la méthode .Add() ne génère pas d'affichage de l'index de position de la donnée insérée a l'écran.

https://powershellexplained.com/2018-10-15-Powershell-arrays-Everything-you-wanted-to-know/#generic-list
https://www.reddit.com/r/PowerShell/comments/9wr6h8/collectionsgenericlistobject_vs/

lien contenant l'explication sur quand utiliser quoi ?
https://gist.github.com/kevinblumenfeld/4a698dbc90272a336ed9367b11d91f1c
When to use what

Synthèse pour les tableaux :
- Arrays si le type d'éléments est connu et a une taille fixe.
- ArrayList si la taille de la liste est fluctuante et/ou le type d'éléments sera mixte.
- Generic List si le type est connu, mais pas la taille de la collection.
- HashTable si vous avez besoin de clés valeurs, dont vous ignorez le type.

5.4.4. Méthode disponible pour les tableaux

1. Foreach
   La méthode foreach peut être utilisée avec les tableaux pour convertir le type de donnée, pour récupérer ou définir des valeurs de propriété pour chaque élément de la collection, pour exécuter une commande sur chaque élément de la collection.

```
# type string
PS C:\>$myvar = ("one", "two", "three")
PS C:\>$myvar.ForEach("ToUpper")
# type int
PS C:\>$myvar = (0..9)
PS C:\>$myvar.ForEach({ $PSItem * $PSItem})
```

1. where
   Permet de filtrer des éléments de la collection selon un ou des critères.

```
PS C:\>$myvar = (0..9)
PS C:\>$myvar.Where{ $PSItem % 2 }
```

1. split
   Le split fractionne ou regroupe les éléments d'une collection en deux collections distinctes.

```
PS C:\>$running, $stopped = (Get-Service).Where({$PSItem.Status -eq 'Running'},
'Split')
PS C:\>$running
```

Supprimer une variable ? n'est pas possible en powershell car elles sont la propriété du système.

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_arrays?view=powershell-7.3

5.4.5. La portée des variables

- GLOBAL
  il s'agit de l'étendue parente racine d'une session PowerShell. Elle va contenir toutes les variables et les fonctions présentes au démarrage de PowerShell.

- LOCAL
  Il s'agit de l'étendue actuellement en utilisation, elle peut correspondre a global, script, ou a celle d'une fonction.
- SCRIPT
  Il s'agit de l'étendue créée lors de l'exécution d'un script.

Vous pouvez créer une étendue enfant en appelant un script ou une fonction. L'étendue appelante est l'étendue parente. Le script ou la fonction appelé est l'étendue enfant. Les fonctions ou scripts que vous appelez peuvent appeler d'autres fonctions, créant une hiérarchie d'étendues enfants dont l'étendue racine est l'étendue globale. Il y'a une notion d'héritage entre les étendues parents et leurs enfants.

Il est possible de fixer l'étendue d'une variable lors de sa déclaration

```
#Ex :
PS C:\>$Global:var
PS C:\>$Script:var
PS C:\>$local:var
PS C:\>$Private:var
```

- PRIVATE
  Il ne s'agit pas d'une étendue, mais d'une option qui spécifie que la variable est privée et visible uniquement dans l'étendue actuelle.

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_scopes?view=powershell-7.3

Synthèse :

GLOBAL
LOCAL
SCRIPT

Quizz :

Citez et donnez les particularités des différents types de portées de variables.

5.4.6. Les propriétés calculées

Permettent de rajouter une propriété a un objet. Cela peut s'avérer pertinent pour changer d'échelle pour une valeur par exemple passer de l'octet à des notions de Giga octet ou Mega Octet.
Cela peut également servir a rappeler la valeur d'une propriété existante dans un autre intitulé, un autre nom de propriété.

```
Select-Object @{n='NomDeLaPropriete';e=expression}
```

Les propriétés calculées ne modifient pas durablement un objet, elle n'existera que le temps d'exécution du script.

→ les formats '{0;N2} -f'

CF : notes sur les formats en annexe

5.5. Fonction

Les fonctions PowerShell sont des **blocs de code**, **nommés**, **réutilisables** qui permettent d'encapsuler des instructions pour effectuer des tâches.
Une fonction doit, pour pouvoir être appelée, avoir été préalablement déclarée. Par convention elle sera donc déclarée en début de script. Sa syntaxe est :

```
function <Nom-Fonction> {
    param(
        [type]$parametre1,
        [type]$parametre2
    )
    # Instructions de la fonction
}
```

1. Déclaration d'une fonction
   Une fonction en PowerShell est :
   Un bloc de code réutilisable, nommé, que l'on peut appeler plusieurs fois dans un script ou une session.
   Toujours déclarée par le mot-clé function :

```
function MonAddition {
    param(
        [int]$a,
        [int]$b
    )
    return $a + $b
}
# Appel de la fonction :
MonAddition -a 2 -b 3   # Renvoie 5
```

Déclaration préalable : Une fonction doit être définie (déclarée) avant d'être appelée, donc généralement au début du script.

1. Dot Sourcing Dot sourcing = charger dans sa session ou dans un autre script les fonctions stockées dans un fichier séparé.

Exemple :

On a un fichier MesFonctions.ps1 :

```
function Hello {
    param($Name)
    Write-Host "Bonjour, $Name"
}
```

On souhaite utiliser Hello dans un autre script ou une session :

```
. .\MesFonctions.ps1  # Remarquez l'espace entre les deux points !
Hello -Name "Alice"   # Affiche 'Bonjour, Alice'
```

Utilité :

Réutilisation facile de fonctions entre différents scripts.

Permet l'organisation du code.

1. Scope des variables Le scope détermine où une variable est accessible.

Scope local : Une variable définie dans une fonction n'est, par défaut, pas visible à l'extérieur de la fonction.

```
function Test-Local {
    $x = 42
}
Test-Local
Write-Host $x  # Erreur : $x n'est pas défini hors de la fonction
```

Scope global : Si on veut qu'une variable soit accessible partout :

```
function Test-Global {
    $global:x = 42
}
```

Le scope du script : En scripts et modules, utilise $script:variable.

Résumé :

Par défaut : scope local à la fonction.

Préfixes utilisables : $global:, $script:, $local: pour préciser la portée.

1. Les paramètres d'une fonction On déclare les paramètres via le bloc param() juste après l'accolade ouvrante.

Chaque paramètre peut être typé :

```
function Greet {
    param([string]$Name, [int]$Repeat)
    1..$Repeat | ForEach-Object { Write-Host "Bonjour, $Name" }
}
Greet -Name Alice -Repeat 2
```

Les paramètres sont obligatoires sauf s'ils ont une valeur par défaut :

```
function Talk {
    param([string]$Message = "Hello")
    Write-Host $Message
}
Talk                # Affiche "Hello"
Talk -Message "Salut !"  # Affiche "Salut !"
```

PowerShell supporte des paramètres positionnels et nommés.

Si vous ne réutilisez pas le bloc de code aucun intérêt d'en faire une fonction.

Synthèse :

`Select-Object @{n='NomDeLaPropriete';e=expression}`
Fonction : utile quand un bloc de code est appelé plusieurs fois dans le script, doit être déclarée avant d'être appelée.

Quizz :

Que fait une propriété calculée par rapport a un objet ?
Citez une utilité des propriétés calculées.
Peut-on appeler une fonction avant sa déclaration ?
Une fonction peut-elle fonctionner avec des paramètres ?

5.6. Exit / Break / Continue

En PowerShell, Exit, Break, et Continue sont trois commandes de contrôle de flux, mais elles ont des usages distincts.

Exit-Break.Différences entre Exit et Break

Commande	Effet	Utilisation principale
Exit	Stoppe tout le script immédiatement	Quitter un script, renvoyer un code de sortie
Break	Stoppe uniquement la boucle ou le switch en cours	Sortir prématurément d'une boucle ou d'un switch
Continue	Ignore le reste du code et passe à l'itération suivante	Sauter une itération dans une boucle

Synthèse :

Exit : sort du script Break : sort de la boucle en cours Continue : ignore le reste du traitement pour passer à la prochaine itération.

Quizz :

5.7. Gestion d'erreurs

5.7.1. $?

la variable système `$?` permet de connaitre le code retour de la dernière commande effectuée par le système. Elle peut être nourrie par l'instruction `exit` au sein d'un script. Il s'agit d'une variable de type booleen : True si la commande s'est exécutée normalement, False si la commande a rencontré une erreur.

5.7.2. Try Catch Finally

La structure Try/Catch/finally permet de faire de la gestion d'erreur

```
#essaye
try { get-toto }
#le catch ne sera exécuté que si on rencontre l'erreur qu'il référence
catch [System.Management.Automation.CommandNotFoundException]
{"Commande non reconnue" }
#le bloc finally est optionnel et sera exécuté quoi qu'il se passe
finally {"ceci est effectué quoi qu'il se passe"}

#essaye
try { get-service -name spooler }
#le catch ne sera exécuté que si on rencontre l'erreur qu'il référence
catch [System.Management.Automation.CommandNotFoundException]
{"Commande non reconnue" }
#le bloc finally est optionnel et sera exécuté quoi qu'il se passe
finally {"ceci est effectué quoi qu'il se passe"}
```

Des propriétés supplémentaires sont accessibles avec la variable $PSItem dans le catch
`$PSItem.ScriptStackTrace`
`$PSItem.Exception`
`$PSItem.ErrorDetails`

```
try { NonsenseString }
catch {
  Write-Host "An error occurred:"
  Write-Host $PSItem
  $PSItem.ScriptStackTrace
  $PSItem.Exception
  $PSItem.ErrorDetails
}
```

Ainsi on va pouvoir cibler précisément les différentes erreurs via leur type en superposant les blocs catch dans notre structure.

On peut empiler les blocs catch pour gérer différents types d'erreurs, il effectue les tests de manière

séquentiel et sortira des catchs sans effectuer les suivants.

```
try { NonsenseString }
#test le type d'erreur commande non trouvée
catch [System.Management.Automation.CommandNotFoundException]
{"Inherited Exception" }
#test le type d'erreur commande non trouvée et d'autres exceptions
catch [System.SystemException] {"Base Exception" }
```

5.7.3. ErrorAction

Le paramètre -ErrorAction va permettre de déterminer quel sera le comportement de powerShell lorsqu'une commande génère une erreur. Les valeurs possibles de -ErrorAction :

ErrorAction.

Valeur	Description
Continue (par défaut)	Affiche l'erreur et continue l'exécution du script.
Stop	Arrête immédiatement le script si une erreur survient.
SilentlyContinue	Ignore l'erreur sans afficher de message.
Ignore (PowerShell 3+)	Comme SilentlyContinue, mais ne stocke pas l'erreur dans $Error.
Inquire	Demande une confirmation avant de poursuivre l'exécution.
Suspend	Mets en pause le script pour un dépannage interactif (utile en mode débogage).

Remove-Item "C:\FichierInexistant.txt" -ErrorAction SilentlyContinue Write-Host "Le script continue malgré l'erreur."

Get-Item "C:\FichierInexistant.txt" -ErrorAction Stop Write-Host "Cette ligne ne sera jamais exécutée si l'erreur survient."

demande confirmation Get-Item "C:\FichierInexistant.txt" -ErrorAction Inquire

Synthèse :

Quizz :

5.8. Pipeline et objet

PowerShell comme évoqué précédemment permet de faire de la redirection de flux de cmdlet. Un type de redirection de flux possible est le pipeline. Dans les langages non objet ce qui sera produit par une commande sera du texte il faut donc lorsque j'utilise un pipeline que ma commande accepte d'avoir pour entrée du texte. Dans le cas du langage objet cette notion se complique, car les objets vont avoir des types définis et il devient donc nécessaire au Shell de mettre en place un ensemble de règles pour s'assurer que l'exécution demandée soit cohérente. Une cmdlet qui arrête un service ne saura pas traiter une collection de processus, cela n'aurait aucun sens.

```
PS C:\>Get-Process | Stop-Service
```

5.8.1. Fonctionnement du pipeline dans le détail

PowerShell tente automatiquement d'associer les objets transmis dans le pipeline aux paramètres appropriés de la commande suivante. Ce processus, appelé "liaison de paramètres", se fait selon des règles spécifiques :
Le paramètre doit accepter l'entrée du pipeline
Le type d'objet transmis doit correspondre ou être convertible au type attendu par le paramètre
Le paramètre ne doit pas avoir déjà été utilisé dans la commande

Les paramètres peuvent accepter l'entrée du pipeline de deux façons principales :

1. ByValue : Le paramètre accepte directement les valeurs du type attendu. Par exemple, le paramètre Name de Start-Service peut accepter des chaînes de caractères directement.
2. ByPropertyName : Le paramètre accepte des objets ayant une propriété du même nom que le paramètre. Cela permet une correspondance basée sur les noms de propriétés plutôt que sur le type d'objet entier.

Processus de diagnostic
Connaitre le type d'objet généré par la cmdlet 1 :
`cmdlet1 | Get-Member` pour connaitre le type ou `cmdlet1.gettype()`
Consulter l'aide de cmdlet2, on vérifie les paramètres qui accepte des entrées de pipeline :
accept pipeline input On vérifie le type de méthode accepté pour cette entrée : méthode byvalue,bypropertyname, ou les deux ? On vérifie le type d'objet accepté par le paramètre byvalue bypropertyname

cmd1	**Pipeline Binding**	**cmd2**
get-member		get-help
type d'objet property type ? (string/int/…)		Pipeline Input \ Accepted:True parameter type
-	By Value	-
Type d'objet		Pipeline input accepted:True \ Type d'objet = ok

cmd1	Pipeline Binding	cmd2
-	By Value = nogo ⇒ By propertyname	-
property name	=	param name
property type	=	property type \ (!!Type PSObject = tout type d'objets!!)
-	by propertyname = nogo Pipeline Input Accepted:True	-
cmd1 Property	calculated property for matching cmd2 property	cmd2 Property

!!Type PSObject = tout type d'objets!!

Synthèse :

Quizz :

5.9. Providers et PSDrives

Les PSDrives et PSProviders sont des concepts fondamentaux dans PowerShell qui permettent d'accéder et de manipuler différents types de données de manière cohérente.

5.9.1. PSProviders

Les PSProviders sont des programmes .NET qui fournissent un accès uniforme à différents types de magasins de données. Ils présentent les données dans un format cohérent, similaire à un système de fichiers.
PowerShell inclut plusieurs fournisseurs intégrés, notamment :
- FileSystem : pour accéder aux fichiers et répertoires
- Registry : pour accéder au registre Windows
- Alias : pour gérer les alias PowerShell
- Environment : pour accéder aux variables d'environnement
- Function : pour gérer les fonctions PowerShell
- Variable : pour accéder aux variables PowerShell
- Certificate : pour accéder aux certificats

5.9.2. PSDrives

Les PSDrives sont des lecteurs virtuels créés par PowerShell pour accéder aux données exposées par les PSProviders.
Ils peuvent être :
Temporaires : existant uniquement dans la session PowerShell actuelle.
Persistants : mappés à des ressources réseau et disponibles entre les sessions.

Les PSDrives permettent d'accéder aux données comme s'il s'agissait de lecteurs du système de fichiers (`set-location`, `get-childitem`,...possible), en utilisant des chemins familiers.
Caractéristiques clés :
Les PSDrives peuvent être créés avec la cmdlet `New-PSDrive`.
`Get-PSDrive` permet de lister tous les lecteurs disponibles.
Les PSDrives peuvent représenter des lecteurs logiques Windows, des partages réseau, ou des espaces de noms abstraits comme le registre.
Ils facilitent la navigation et la manipulation des données à travers différents magasins de données de manière cohérente.

Cette architecture permet à PowerShell d'offrir une interface unifiée pour interagir avec divers types de données, simplifiant ainsi l'automatisation et la gestion des systèmes.

```
PS C:\>Get-PSDrive
```

`Get-Help About_providers`

Synthèse :

Quizz :

5.10. Signer un script

La signature de scripts PowerShell est un processus important pour garantir l'intégrité et l'authenticité du code.
Le certificat peut être émis par une autorité de certification ou être auto-signé.
Le certificat doit être stocké dans le magasin de certificats de l'utilisateur ou de l'ordinateur.
Pour que cela fonctionne en AllSigned le certificat doit être aussi présent dans les magasins "Autorités de certification racine de confiance" et "Éditeurs approuvés".

```
# Création d'un certificat auto-signé
PS C:\>$SelfSignedCert = New-SelfSignedCertificate -Subject ScriptPowerShell -Type
CodeSigningCert -CertStoreLocation Cert:\LocalMachine\My -FriendlyName
"SelfSignCert_scripts_PowerShell" -NotAfter (Get-Date).AddYears(5)
#stockage du certificat dans une variable
PS C:\>$cert = Get-ChildItem Cert:\CurrentUser\My -CodeSigningCert | Select-Object
-First 1

#signature du script
PS C:\>Set-AuthenticodeSignature -FilePath <chemin_du_script> -Certificate $cert
```

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_signing?view=powershell-7.5

https://www.it-connect.fr/chapitres/powershell-signer-un-script-ps1-avec-un-certificat-auto-signe/

https://www.it-connect.fr/comment-signer-un-script-powershell/

Synthèse :

Quizz :

5.11. PSRemoting

Le PowerShell Remoting est une fonctionnalité de PowerShell qui permet d'exécuter des commandes et des scripts sur des ordinateurs distants.
Ce principe utilise le protocole WS-Management (WS-Man) et le service Windows Remote Management (WinRM) Il fonctionne sur les ports HTTP 5985 ou HTTPS 5986 (recommandé pour plus de sécurité)

Il est activé par défaut sur les versions serveur de Windows récentes.
Sur les versions clientes, nécessite l'activation via la commande Enable-PSRemoting
Il requiert des configurations de pare-feu et de sécurité appropriées

Son Utilisation peut prendre deux formes : - Enter-PSSession pour une session interactive sur un ordinateur distant - Invoke-Command pour exécuter des commandes ou des scripts sur un ou plusieurs ordinateurs distants Possibilité d'utiliser des sessions persistantes avec New-PSSession

En termes de sécurité il utilise l'authentification Windows par défaut, il supporte l'authentification CredSSP pour les scénarios de double saut.
il peut être configuré pour utiliser HTTPS pour un chiffrement renforcé.

Synthèse :

Quizz :

5.12. SSH vers du linux

Avec Windows 10/11 ou Windows server 2019+ SSH est préinstallé.
deux possibilités pour ouvrir une session en SSH sur un Linux :
- `Enter-PSSession -HostName <adresse_ip_ou_nom_hote> -UserName <nom_utilisateur>`
- `Invoke-Command -HostName <adresse_ip_ou_nom_hote> -UserName <nom_utilisateur> -ScriptBlock { <commande_linux> }`

1. Générez une paire de clés SSH si vous n'en avez pas déjà : `ssh-keygen -t ed25519`
2. Copiez la clé publique sur le serveur distant dans le fichier `~/.ssh/authorized_keys`.
3. Assurez-vous que l'authentification par clé est activée sur le serveur SSH en vérifiant que le fichier sshd_config contient : `PubkeyAuthentication yes` .
   Pour vous connecter, utilisez la commande Enter-PSSession ou New-PSSession en spécifiant le chemin vers votre clé privée.
4. `Enter-PSSession -HostName serveur.exemple.com -UserName utilisateur -IdentityFilePath ~\.ssh\id_ed25519`
   ou
   `$session = New-PSSession -HostName serveur.exemple.com -UserName utilisateur -IdentityFilePath ~\.ssh\id_ed25519`
5. Pour exécuter une commande unique sans entrer dans une session interactive, utilisez `Invoke-Command` :
   `Invoke-Command -HostName serveur.exemple.com -UserName utilisateur -IdentityFilePath ~\.ssh\id_ed25519 -ScriptBlock { commande_linux }`

Synthèse :

Quizz :

5.13. WMI / CIM / DOTNET

Windows Management Instrumentation (WMI) est l'ancien outillage Windows, il est déprécié depuis PowerShell 6+. Common Information Model (CIM) l'a remplacé depuis PowerShell 3.0 et est "rétrocompatible" avec WMI.
`Get-Command -Module CimCmdlets` vous permet d'avoir la liste des commandes disponibles.

https://learn.microsoft.com/fr-fr/powershell/scripting/learn/ps101/07-working-with-wmi?view#powershell-7.3

Synthèse :

Quizz :

6. Annexes sur des Points techniques

6.1. Write-Host ou Write-Output

`Write-Host` et `Write-Output` sont deux cmdlets PowerShell qui servent à afficher des informations, mais leurs fonctionnements et utilisations diffèrent significativement :

`Write-Host` Écrit directement sur la console ou l'hôte PowerShell.
Ne produit pas de sortie dans le pipeline.
Permets de personnaliser l'affichage (couleurs, arrière-plan).
Utilisé principalement pour des messages destinés à l'utilisateur.
Ne peut pas être capturé dans une variable ou redirigé facilement.

`Write-Output` Envoie des objets au pipeline (flux de réussite ou stdout).
Peut être capturé dans une variable, redirigé ou pipeline vers d'autres cmdlets.
Est le comportement par défaut en PowerShell pour renvoyer des données.
Utilisé pour produire des données que le script ou d'autres commandes peuvent utiliser.

Choix d'utilisation Utilisez `Write-Host` pour des messages purement informatifs destinés à l'utilisateur.
Préférez `Write-Output` pour des données que vous voulez manipuler ou stocker dans le script.
En résumé, `Write-Host` est pour l'affichage pur, tandis que `Write-Output` est pour la manipulation de données dans le pipeline PowerShell

6.2. Caractères spéciaux en PowerShell

Caracteres speciaux	signification
"	The beginning(or end) of quoted text. Used to represent strings
'	The beginning(or end) of quoted text. Used to represent strings. special characters lose their significance within single quotes
#	The beginning of a comment
$	The beginning of a variable
&	Reserved for future use
()	Parentheses used for subexpressions
{}	Script block
;	Statement Separator
|	Pipeline seperator
`	Escape Character
?	Alias operator for Where-Object
%	Alias Operator for for-loop

6.3. ETS Properties

Les propriétés ETS (Extended Type System) en PowerShell sont des membres qui peuvent être traités comme des propriétés d'objet. Voici une synthèse de leurs caractéristiques principales :

- Types de propriétés ETS Alias properties : créent un alias pour une propriété existante Code properties : implémentées en code .NET PowerShell properties : définies dans PowerShell Note properties : stockent des valeurs simples Script properties : calculées par un script PowerShell
- Fonctionnalités Permettent d'étendre les types d'objets existants Accessibles via la propriété Members des objets PSObject Peuvent apparaître du côté gauche d'une expression Offrent une flexibilité pour adapter et personnaliser les objets
- Utilisation Définies dans des fichiers XML de types (*.ps1xml) Importées avec la cmdlet Update-TypeData Permettent d'ajouter de la cohérence entre différents types d'objets Utiles pour adapter des objets .NET aux besoins de PowerShell

Les propriétés ETS sont un élément clé du système de type étendu de PowerShell, offrant aux développeurs et administrateurs un moyen puissant de manipuler et d'étendre les objets dans l'environnement PowerShell

https://learn.microsoft.com/fr-fr/powershell/scripting/developer/ets/properties?view=powershell-7.4

6.4. Travailler avec les chaines de caractères

Le travail avec les chaînes de caractères en PowerShell offre de nombreuses possibilités grâce à diverses méthodes et techniques. Voici une synthèse des principaux aspects :

- Méthodes de manipulation
  PowerShell propose plusieurs méthodes pour manipuler les chaînes : Length : compte le nombre de caractères
  ToUpper() : converti en majuscules
  ToLower() : converti en minuscules
  Replace() : remplace des caractères ou sous-chaînes
  Split() : découpe une chaîne en tableau
  Opérateurs de fractionnement et de jointure
  -split fractionne une chaîne en sous-chaînes.
  -join concatène plusieurs chaînes en une seule chaîne.
- Opérations courantes
  Concaténation : utilisation de l'opérateur '+' ou de chaînes entre guillemets doubles
  Substitution de variables : insertion de variables dans des chaînes avec $variable
  Formatage : utilisation de -f pour formater des chaînes complexes
  Recherche : Select-String pour chercher des motifs dans des chaînes ou fichiers
- Techniques avancées
  Expressions régulières pour des recherches et manipulations complexes
  Exécution de commandes dans des chaînes avec $()
  Utilisation de chaînes de format pour une mise en forme avancée

- Considérations importantes
  Les guillemets simples empêchent la substitution de variables, ou le caractère d'échappement " `".
  L'encodage des fichiers peut affecter le traitement des chaînes.
  Les méthodes .NET sont disponibles pour des manipulations plus poussées.
- ConvertToString
  Méthode statique de la classe Convert pour convertir différents types en chaînes
  Syntaxe : [Convert]::ToString(valeur, [baseOptionnelle])
  Permets de convertir des nombres, dates, et autres types en chaînes
  Accepte un paramètre optionnel pour spécifier la base (pour les nombres)
  Exemple : [Convert]::ToString(123, 2) convertit 123 en binaire ("1111011")
- Substring
  Méthode permettant d'extraire une partie d'une chaîne de caractères
  Syntaxe : $chaine.Substring(indexDébut, longueur)
  Peut être utilisé avec un seul paramètre pour extraire jusqu'à la fin de la chaîne
  Utile pour extraire des sous-chaînes de longueur fixe ou variable
  Exemple : $texte.Substring(0, 3) extrait les 3 premiers caractères

https://learn.microsoft.com/fr-fr/powershell/scripting/learn/deep-dives/everything-about-string-substitutions?view=powershell-7.5

https://www.it-connect.fr/powershell-et-substring-extraire-une-chaine-dune-chaine/?utm_content=cmp-true

6.5. Operateurs

- Opérateurs arithmétiques
  Utilisez des opérateurs arithmétiques (+, , -, */, %) pour calculer des valeurs dans une commande ou une expression. Les opérateurs au niveau du bit (-band, , -bor, -bxor-bnot, -shl) -shrmanipulent les modèles de bits dans les valeurs.
- Opérateurs de fractionnement et de jointure
  -split fractionne une chaîne en sous-chaînes.
  -join concatène plusieurs chaînes en une seule chaîne.
- Opérateurs d'assignation
  Utilisez des opérateurs d'affectation (=, , +=, -=*=, /=%=) pour affecter, modifier ou ajouter des valeurs à des variables. Vous pouvez combiner des opérateurs arithmétiques avec une affectation pour affecter le résultat de l'opération arithmétique à une variable.
- Opérateurs unaires
  Utilisez l'unaire ++ et — les opérateurs pour incrémenter ou décrémenter des valeurs et - pour la négation.

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_operators?view#powershell-7.3#dot-sourcing-operator-

6.5.1. Opérateur de sous-expression $( )

Retourne le résultat d'une ou plusieurs instructions. Pour un résultat unique, retourne un scalaire. Pour plusieurs résultats, retourne un tableau. Utilisez cette option lorsque vous souhaitez utiliser une expression dans une autre expression. Par exemple, pour incorporer les résultats de la commande dans une expression de chaîne.

```
PS C:\> "Today is $(Get-Date)"
PS C:\>Today is 12/02/2019 13:15:20
```

6.5.2. Opérateur de format -f

Met en forme des chaînes à l'aide de la méthode de format des objets string. Entrez la chaîne de format sur le côté gauche de l'opérateur et les objets à mettre en forme sur le côté droit de l'opérateur.

- N - Numbers
- C - Currency
- P - Percentage
- X - Hex
- D - Decimal

```
# Afficher 1 puis Hello puis Pi avec 2 chiffres apres la virgule
PS C:\>"{0} {1,-10} {2:N}" -f 1,"hello",[math]::pi
PS C:\>1 hello      3.14

$objets = 123
$rep = 'documents'
"il y a {0} objets dans le répertoire {1}." -f $objets, $rep
PS >il y a 123 objets dans le répertoire documents.

#pour un affichage en pourcentage de la taille restante d'un volume en utilisant N
pour numeric
Get-Volume | Select-Object DriveLetter, @{Name='PercentFree'; Expression={ '{0:N2}%'
-f (($PSItem.SizeRemaining / $PSItem.Size) * 100) }}
#pour un affichage en pourcentage de la taille restante d'un volume en utilisant P
pour le pourcentage
Get-Volume | Select-Object DriveLetter, @{Name='PercentFree'; Expression={ '{0:P0}' -f
(($PSItem.SizeRemaining / $PSItem.Size)) }}

#Exemple en Hexa
Write-Host ("Hexadécimal : {0:X}" -f 255)
PS >Hexadécimal : FF
Write-Host ("Hexadécimal : {0:X}" -f 10)
PS >Hexadécimal : A
Write-Host ("Hexadécimal : {0:X6}" -f 4095)
```

```
PS >Hexadécimal : 000FFF
```

Détail de {0:N2}
{0:...} :
Ici, 0 est l'index du paramètre qui va être placé dans la chaîne. C'est-à-dire, si tu écris : '{0:N2}' -f 123.45678
alors le 0 fait référence au premier élément qui suit le -f (ici, 123.45678).

N signifie Number (nombre) :
C'est un spécificateur de format numérique standard utilisé pour afficher un nombre avec des séparateurs de milliers (si nécessaire) et un nombre fixe de décimales.

2 après N précise qu'on veut deux décimales.

6.5.3. Opérateurs de chaîne de && pipeline et ||

Exécutez de manière conditionnelle le pipeline de droite en fonction de la réussite du pipeline de gauche.
Les opérateurs de chaîne de pipeline \&\& et `||` **ont été introduits dans PowerShell** 7 pour permettre l'exécution conditionnelle de commandes.

- Fonctionnement
  - `&&` (ET) : Exécute la commande de droite uniquement si celle de gauche réussit.
  - `||` (OU) : Exécute la commande de droite uniquement si celle de gauche échoue.
- Caractéristiques
  - Utilisent les variables `$?` et `$LASTEXITCODE` pour déterminer le succès ou l'échec d'une commande.
  - Fonctionnent avec des cmdlets, des fonctions et des commandes natives.
  - Ont une priorité inférieure au pipeline ('|') et à la redirection ('>'), mais supérieure aux opérateurs de travail ('&') et d'affectation ('=').
- Avantages
  - Permettent d'écrire du code plus concis et lisible.
  - Facilitent l'enchaînement conditionnel de commandes sans utiliser de structures if-else explicites.

```
# Exécute la seconde commande si la première réussit
Command1 && Command2

# Exécute la seconde commande si la première échoue
Command1 || Command2
```

Ces opérateurs offrent une syntaxe similaire à celle des Shell POSIX (bash, zsh) et de l'interpréteur de commandes Windows, améliorant ainsi la flexibilité et l'expressivité de PowerShell pour

l'exécution conditionnelle de commandes.

6.5.4. Opérateur de plage ..

L'opérateur de plage peut être utilisé pour représenter un tableau d'entiers séquentiels ou de caractères. Les valeurs jointes par l'opérateur de plage définissent les valeurs de début et de fin de la plage.

```
PS C:\>5..-5 | ForEach-Object {Write-Output $_}
```

6.5.5. Operateur d'appel

`&` Permets d'exécuter des commandes stockées dans des variables et représentées par des chaînes ou des blocs de script. L'opérateur d'appel s'exécute dans une étendue enfant. (voir la notion de scope)

```
PS C:\>$c = "Get-ExecutionPolicy"
PS C:\>$c
PS C:\>Get-ExecutionPolicy
PS C:\>& $c
PS C:\>AllSigned
```

6.5.6. Opérateur d'accès aux membres

L'opérateur d'accès aux membres en PowerShell, représenté par le point (.), permet d'accéder aux propriétés et méthodes des objets.

```
#Accède aux propriétés et méthodes d'un objet.

PS C:\>(Get-Process PowerShell).kill()
PS C:\>(Get-Service).gettype()
PS C:\>$var.length
```

6.5.7. Opérateur membre statique '::'

Appelle les propriétés statiques et les méthodes d'une classe .NET Framework. Pour rechercher les propriétés et méthodes statiques d'un objet, utilisez le paramètre Static de l'applet `Get-Member` de commande. Le nom du membre peut être une expression.

```
[datetime]::Now
'MinValue', 'MaxValue' | Foreach-Object { [int]:: $_ }
```

6.5.8. Opérateur d'approvisionnement par points '.'

Exécute un script dans l'étendue actuelle afin que toutes les fonctions, alias et variables créées par le script soient ajoutées à l'étendue actuelle, en remplaçant les fonctions existantes. Les paramètres déclarés par le script deviennent des variables. Les paramètres pour lesquels aucune valeur n'a été donnée deviennent des variables sans valeur. Toutefois, la variable `$args` automatique est conservée.

```
. c:\scripts\sample.ps1 1 2 -Also:3
<#Le point (.) au début est un opérateur de "dot-sourcing" qui exécute le script dans
le contexte actuel #plutôt que dans un nouveau contexte.

#Les arguments "1" et "2" sont passés comme arguments positionnels non nommés. Ils
seront accessibles dans le script via $args et $args1 respectivement.

#L'argument "-Also:3" est un argument nommé. Dans le script, il sera accessible via un
paramètre nommé #"Also" s'il est défini, sinon il sera traité comme un argument
supplémentaire.

#Cette syntaxe permet de passer à la fois des arguments positionnels et des arguments
nommés à un #script PowerShell
#>
```

6.6. Travail en arrière plan

- PowerShell offre plusieurs méthodes pour exécuter des tâches en arrière-plan :
  - `Start-Job` : Permets de lancer une tâche en arrière-plan dans un processus séparé.
  - BackgroundJob : Utilise des processus multiples pour l'exécution de tâches5.
  - ThreadJob : Basé sur des threads multiples, offrant de meilleures performances que BackgroundJob5.
- Avantages du travail en arrière-plan :
  Exécution de tâches longues sans bloquer la console PowerShell.
  Possibilité de paralléliser des tâches pour améliorer les performances.
- Commandes utiles :
  - `Get-Job` : Affiche le statut des tâches en cours.
  - `Receive-Job` : Récupère les résultats d'une tâche terminée.

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_jobs?view=powershell-7.5

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/invoke-command?view=powershell-7.5

namespace

PowerShell 7 amène une nouveauté sur ce sujet avec l'utilisation possible de `| ForEach-Object -Parallel`.

```
$collection | ForEach-Object -Parallel {
    # Code à exécuter en parallèle
} -ThrottleLimit <nombre>
```

- Exécution parallèle : Chaque élément de la collection est traité dans un espace de travail PowerShell (Runspace) distinct, permettant une exécution simultanée.
- ThrottleLimit : Ce paramètre contrôle le nombre maximal de tâches parallèles exécutées simultanément. Par défaut, il est fixé à 53.
- Variables partagées : Pour utiliser des variables externes dans le bloc parallèle, utilisez le mot-clé $using:

```
$sharedVariable = "Valeur partagée"
1..10 | ForEach-Object -Parallel {
    Write-Output "$using:sharedVariable : $PSItem"
} -ThrottleLimit 3
$ip = 1..254 | ForEach-Object -parallel { test-netconnection "192.168.1.$PSItem" }
-ThrottleLimit 3
```

C'est particulièrement indiqué dans les cas suivant : * Traitement de grandes quantités de données : Idéal pour les opérations longues sur de nombreux éléments.

- Tâches avec attente : Efficace pour les opérations impliquant des délais, comme les requêtes réseau.

6.6.1. Opérateur en arrière-plan &

Equivalent à Start-Job, en moins lisible.

```
PS C:\>Get-Process -Name pwsh &
PS C:\>Start-Job -ScriptBlock {Get-Process -Name pwsh}
```

est équivalent a un `start-job` donc créé un Shell enfant.

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_operators?view#powershell-7.3#call-operator-

6.7. Regex : Expression régulieres

Une expression régulière est un modèle ou template utilisé pour faire correspondre le texte.
PowerShell utilise le moteur regex .NET .
PowerShell a plusieurs opérateurs et applets de commande qui utilisent des expressions régulières :
- `Select-String`
- `-match`, `-split` et `-replace`, pour les opérateurs
- Instruction switch avec l'option -regex

La casse n'est pas prise en compte par défaut mais peut l'être en le précisant avec `-CaseSensitive`, ou en ajoutant un 'c' devant l'operateur ex : `-cmatch`.

Le caractère d'échappement en PowerShell est le '\'.

Caracteres	Descrption
*	Zéro ou plus de fois.
+	Une ou plusieurs fois.
?	Zéro ou une fois.
[aze]	'a', 'z' ou 'e'
[^aze]	pas 'a', 'z' ou 'e'
[a-z]	ensemble de lettres de 'a' à 'z'
.	tout caractère sauf '\n'
\	Caractère d'échappement pour le caractère suivant
\w	Un mot
\d	Decimal
\t	Correspond à une tabulation
\n	Correspond à une ligne de nouvelle ligne
\r	Correspond à un retour chariot
\s	Correspond à un espace
^a	ligne qui commence par 'a'
a$	ligne qui finit par 'a'
{n,m}	Au moins n, mais pas plus que m de fois.
{n}	Correspond exactement n au nombre de fois.
{n,}	Mettre en correspondance au MOINS n le nombre de fois.
{n,m}	Correspondance entre n et m nombre de fois.

Groupes, captures et substitutions sont possible avec les regex.
`$Matches` est une variable automatique de hachage stockant l'intégralité de la correspondance sous

forme de tableau avec un index automatique ou nommé.
$& dans les substitutions, représente tout le texte mis en correspondance.

```
# Ceci renvoi la valeur true pour tout serveur un nom qui est composé de lettres suivi
de deux
#chiffres, avec ou sans tiret les séparant.
PS >'SERVER01' -match '^[A-Z]+-?\d\d'
PS >True

# Autre exemple en utilisant la variable de hashage
PS >"Bienvenue a vous $env:username" -match '(.+vous )(.+)'
True
PS > $matches
Name                           Value
----                           -----
2                              Administrateur
1                              Bienvenue a vous
0                              Bienvenue a vous Administrateurr
```

https://learn.microsoft.com/fr-fr/powershell/module/microsoft.powershell.core/about/about_regular_expressions?view=powershell-7.5

https://learn.microsoft.com/fr-fr/dotnet/standard/base-types/regular-expressions
https://learn.microsoft.com/fr-fr/dotnet/standard/base-types/the-regular-expression-object-model

Cheat Sheet regex microsoft .NET

6.8. Classe, Object, Typename .net

Type .NET
Un type .NET est comme un modèle qui définit la structure et le comportement d'un objet.
Il détermine les propriétés et méthodes disponibles pour un objet.
Les types .NET sont organisés de manière hiérarchique, héritant les uns des autres.

- Classe
  Une classe est un type .NET qui sert de plan pour créer des objets.
  Elle définit les attributs (données) et les méthodes (comportements) que les objets de cette classe auront.
  En C#, une classe est déclarée avec le mot-clé "class" suivi de son nom.
- Objet
  Un objet est une instance concrète d'une classe.
  Il est créé à partir d'une classe en utilisant le mot-clé "new" suivi du nom de la classe.
  Chaque objet a ses propres valeurs pour les attributs définis dans la classe.
- Constructeur
  Un constructeur est une méthode spéciale appelée lors de la création d'un objet.

Il porte le même nom que la classe et n'a pas de type de retour.
Son rôle est d'initialiser les attributs de l'objet nouvellement créé.
Une classe peut avoir plusieurs constructeurs (surcharge) avec différents paramètres.

- Informations
  Le constructeur est appelé automatiquement lors de l'instanciation d'un objet avec "new".
  Si aucun constructeur n'est défini, un constructeur par défaut sans paramètre est fourni.
  Les constructeurs peuvent avoir des paramètres pour initialiser l'objet avec des valeurs spécifiques.

Pour aller plus loin formation .NET https://learn.microsoft.com/fr-fr/training/dotnet/

INDEX