



eBook Gratuit

APPRENEZ PowerShell

eBook gratuit non affilié créé à partir des
contributeurs de Stack Overflow.

#powershell

Table des matières

À propos.....	1
Chapitre 1: Démarrer avec PowerShell.....	2
Remarques.....	2
Versions.....	2
Exemples.....	2
Installation ou configuration.....	2
les fenêtres.....	2
Autres plates-formes.....	3
Autoriser les scripts stockés sur votre machine à s'exécuter sans signature.....	3
Alias et fonctions similaires.....	4
Le pipeline - Utilisation de la sortie d'une applet de commande PowerShell.....	5
Commentant.....	6
Appeler des méthodes de bibliothèque .Net.....	6
Créer des objets.....	7
Chapitre 2: Aide basée sur les commentaires.....	9
Introduction.....	9
Exemples.....	9
Aide basée sur les commentaires de fonction.....	9
Aide basée sur les commentaires de script.....	11
Chapitre 3: Alias.....	14
Remarques.....	14
Exemples.....	15
Get-Alias.....	15
Alias Set.....	15
Chapitre 4: Analyse CSV.....	17
Exemples.....	17
Utilisation de base de Import-Csv.....	17
Importer à partir de CSV et convertir les propriétés pour corriger le type.....	17
Chapitre 5: Anonymiser les adresses IP (v4 et v6) dans un fichier texte avec Powershell.....	19
Introduction.....	19

Exemples.....	19
Anonymiser l'adresse IP dans un fichier texte.....	19
Chapitre 6: Application des prérequis du script.....	21
Syntaxe.....	21
Remarques.....	21
Exemples.....	21
Appliquer une version minimale de l'hôte powershell.....	21
Exécuter le script en tant qu'administrateur.....	21
Chapitre 7: Automatisation de l'infrastructure.....	23
Introduction.....	23
Exemples.....	23
Script simple pour le test d'intégration des boîtes noires des applications de la console.....	23
Chapitre 8: Boucles.....	24
Introduction.....	24
Syntaxe.....	24
Remarques.....	24
Pour chaque.....	24
Performance.....	25
Exemples.....	25
Pour.....	25
Pour chaque.....	25
Tandis que.....	26
ForEach-Object.....	27
Utilisation de base.....	27
Utilisation avancée.....	27
Faire.....	28
Méthode ForEach ().....	28
Continuer.....	29
Pause.....	29
Chapitre 9: Classes PowerShell.....	31
Introduction.....	31

Exemples.....	31
Méthodes et propriétés.....	31
Liste des constructeurs disponibles pour une classe.....	31
Surcharge du constructeur.....	33
Obtenir tous les membres d'une instance.....	33
Modèle de classe de base.....	33
Héritage de la classe parent à la classe enfant.....	34
Chapitre 10: Cmdlet Naming.....	35
Introduction.....	35
Exemples.....	35
Verbes.....	35
Des noms.....	35
Chapitre 11: Comment télécharger le dernier artefact d'Artifactory en utilisant le script.....	36
Introduction.....	36
Exemples.....	36
Script Powershell pour télécharger le dernier artefact.....	36
Chapitre 12: Communication avec les API RESTful.....	37
Introduction.....	37
Exemples.....	37
Utilisez les Webhooks entrants Slack.com.....	37
Poster un message à hipChat.....	37
Utilisation de REST avec des objets PowerShell pour obtenir et mettre des données individu.....	37
Utilisation de REST avec des objets PowerShell pour GET et POST de nombreux éléments.....	38
Utilisation de REST avec PowerShell pour supprimer des éléments.....	38
Chapitre 13: Communication TCP avec PowerShell.....	39
Exemples.....	39
TCP listener.....	39
TCP expéditeur.....	39
Chapitre 14: Comportement de retour dans PowerShell.....	41
Introduction.....	41
Remarques.....	41
Exemples.....	41

Sortie anticipée.....	41
Je t'ai eu! Retour dans le pipeline.....	42
Je t'ai eu! Ignorer les sorties indésirables.....	42
Retour avec une valeur.....	42
Comment travailler avec les fonctions retourne.....	43
Chapitre 15: Configuration d'état souhaitée.....	45
Exemples.....	45
Exemple simple - Activation de WindowsFeature.....	45
Démarrage de DSC (mof) sur une machine distante.....	45
Importation de psd1 (fichier de données) dans une variable locale.....	45
Liste des ressources disponibles de DSC.....	46
Importation de ressources à utiliser dans DSC.....	46
Chapitre 16: Conventions de nommage.....	47
Exemples.....	47
Les fonctions.....	47
Chapitre 17: Cordes.....	48
Syntaxe.....	48
Remarques.....	48
Exemples.....	48
Créer une chaîne de base.....	48
Chaîne.....	48
Chaîne littérale.....	48
Format chaîne.....	49
Chaîne multiligne.....	49
Chaîne de caractères.....	49
Chaîne de caractères.....	49
Littéral ici-chaîne.....	50
Chaînes concaténantes.....	50
Utilisation de variables dans une chaîne.....	50
Utiliser l'opérateur +.....	50
Utiliser des sous-expressions.....	50

Caractères spéciaux.....	51
Chapitre 18: Création de ressources DSC basées sur les classes.....	52
Introduction.....	52
Remarques.....	52
Exemples.....	52
Créer une classe de squelette de ressource DSC.....	52
Squelette de ressource DSC avec propriété de clé.....	52
Ressource DSC avec propriété obligatoire.....	53
Ressource DSC avec les méthodes requises.....	53
Chapitre 19: Déclaration de changement.....	55
Introduction.....	55
Remarques.....	55
Exemples.....	55
Commutateur simple.....	55
Instruction de changement avec le paramètre Regex.....	55
Commutation simple avec pause.....	56
Instruction de changement avec un paramètre générique.....	56
Instruction de changement avec paramètre exact.....	57
Instruction de changement avec le paramètre CaseSensitive.....	57
Instruction de changement avec paramètre de fichier.....	58
Commutateur simple avec condition par défaut.....	58
Changer de déclaration avec des expressions.....	59
Chapitre 20: Envoi d'email.....	60
Introduction.....	60
Paramètres.....	60
Exemples.....	61
Message d'envoi simple.....	61
Send-MailMessage avec des paramètres prédéfinis.....	61
SMTPClient - Mail avec un fichier .txt dans le message du corps.....	62
Chapitre 21: Executables en cours d'exécution.....	63
Exemples.....	63
Applications de console.....	63

Applications GUI.....	63
Flux de console.....	63
Codes de sortie.....	64
Chapitre 22: Expressions régulières.....	65
Syntaxe.....	65
Exemples.....	65
Match unique.....	65
Utilisation de l'opérateur -Match.....	65
Utilisation de Select-String.....	66
Utilisation de [RegEx] :: Match ().....	67
Remplacer.....	67
Utilisation de l'opérateur -Replace.....	67
En utilisant la méthode [RegEx] :: Replace ().....	68
Remplacer le texte par une valeur dynamique par un MatchEvaluator.....	68
Échapper des caractères spéciaux.....	69
Plusieurs correspondances.....	70
Utilisation de Select-String.....	70
Utiliser [RegEx] :: Matches ().....	70
Chapitre 23: Flux de travail PowerShell.....	72
Introduction.....	72
Remarques.....	72
Exemples.....	72
Exemple de workflow simple.....	72
Workflow avec des paramètres d'entrée.....	72
Exécuter le workflow en tant que job d'arrière-plan.....	73
Ajouter un bloc parallèle à un workflow.....	73
Chapitre 24: Fonctions PowerShell.....	74
Introduction.....	74
Exemples.....	74
Fonction simple sans paramètre.....	74
Paramètres de base.....	74

Paramètres obligatoires.....	75
Fonction avancée.....	76
Validation des paramètres.....	77
ValidateSet.....	77
ValiderRange.....	78
ValidatePattern.....	78
ValidateLength.....	78
ValidateCount.....	78
ValidateScript.....	78
Chapitre 25: Gestion des paquets.....	80
Introduction.....	80
Exemples.....	80
Recherchez un module PowerShell à l'aide d'un motif.....	80
Créez la tolérance de module PowerShell par défaut.....	80
Trouver un module par nom.....	80
Installer un module par nom.....	80
Désinstaller un module mon nom et sa version.....	80
Mettre à jour un module par nom.....	80
Chapitre 26: Gestion des secrets et des informations d'identification.....	82
Introduction.....	82
Exemples.....	82
Demander des informations d'identification.....	82
Accéder au mot de passe en texte clair.....	82
Travailler avec des informations d'identification stockées.....	83
Encrypter.....	83
Le code qui utilise les informations d'identification stockées:.....	83
Stockage des informations d'identification sous forme chiffrée et transmission en tant que.....	84
Chapitre 27: GUI dans Powershell.....	85
Exemples.....	85
Interface utilisateur WPF pour applet de commande Get-Service.....	85
Chapitre 28: HashTables.....	87

Introduction.....	87
Remarques.....	87
Exemples.....	87
Créer une table de hachage.....	87
Accédez à une valeur de table de hachage par clé.....	87
En boucle sur une table de hachage.....	88
Ajouter une paire de valeurs de clé à une table de hachage existante.....	88
Enumérer à travers les clés et les paires valeur-clé.....	88
Supprimer une paire de valeurs de clé d'une table de hachage existante.....	89
Chapitre 29: Incorporation de code géré (C # VB).....	90
Introduction.....	90
Paramètres.....	90
Remarques.....	90
Suppression de types ajoutés.....	90
Syntaxe CSharp et .NET.....	90
Exemples.....	91
C # Exemple.....	91
Exemple VB.NET.....	91
Chapitre 30: Introduction à Pester.....	93
Remarques.....	93
Exemples.....	93
Premiers pas avec Pester.....	93
Chapitre 31: Introduction à Psake.....	95
Syntaxe.....	95
Remarques.....	95
Exemples.....	95
Contour de base.....	95
Exemple FormatTaskName.....	95
Exécuter la tâche sous condition.....	96
ContinueOnError.....	96
Chapitre 32: Jeux de paramètres.....	97
Introduction.....	97

Exemples.....	97
Jeux de paramètres simples.....	97
Parameterset pour imposer l'utilisation d'un paramètre lorsqu'un autre est sélectionné.....	97
Paramètre défini pour limiter la combinaison des paramètres.....	98
Chapitre 33: La gestion des erreurs.....	99
Introduction.....	99
Exemples.....	99
Types d'erreur.....	99
Chapitre 34: Les opérateurs.....	101
Introduction.....	101
Exemples.....	101
Opérateurs arithmétiques.....	101
Opérateurs logiques.....	101
Opérateurs d'affectation.....	101
Opérateurs de comparaison.....	102
Opérateurs de redirection.....	102
Mélanger les types d'opérandes: le type de l'opérande gauche détermine le comportement.....	103
Opérateurs de manipulation de chaînes.....	104
Chapitre 35: Ligne de commande PowerShell.exe.....	105
Paramètres.....	105
Exemples.....	106
Exécuter une commande.....	106
-Command <string>.....	106
-Command {scriptblock}.....	106
-Command - (entrée standard).....	106
Exécuter un fichier script.....	107
Script de base.....	107
Utiliser des paramètres et des arguments.....	107
Chapitre 36: Logique conditionnelle.....	109
Syntaxe.....	109
Remarques.....	109

Exemples.....	109
si, sinon et sinon si.....	109
Négation.....	110
Si sténographie conditionnelle.....	110
Chapitre 37: Module ActiveDirectory.....	112
Introduction.....	112
Remarques.....	112
Exemples.....	112
Module.....	112
Utilisateurs.....	112
Groupes.....	113
Des ordinateurs.....	113
Objets.....	113
Chapitre 38: Module d'archive.....	115
Introduction.....	115
Syntaxe.....	115
Paramètres.....	115
Remarques.....	116
Exemples.....	116
Compresser les archives avec un joker.....	116
Mettre à jour le ZIP existant avec Compress-Archive.....	116
Extraire un zip avec Expand-Archive.....	116
Chapitre 39: Module de tâches planifiées.....	117
Introduction.....	117
Exemples.....	117
Exécuter un script PowerShell dans une tâche planifiée.....	117
Chapitre 40: Module ISE.....	118
Introduction.....	118
Exemples.....	118
Scripts de test.....	118
Chapitre 41: Module SharePoint.....	119
Exemples.....	119

Chargement du composant logiciel enfichable SharePoint.....	119
Itérer toutes les listes d'une collection de sites.....	119
Obtenez toutes les fonctionnalités installées sur une collection de sites.....	119
Chapitre 42: Modules Powershell.....	121
Introduction.....	121
Exemples.....	121
Créer un manifeste de module.....	121
Exemple de module simple.....	121
Exportation d'une variable à partir d'un module.....	122
Structuration des modules PowerShell.....	122
Emplacement des modules.....	123
Visibilité du membre du module.....	123
Chapitre 43: Modules, scripts et fonctions.....	124
Introduction.....	124
Exemples.....	124
Fonction.....	124
Démo.....	124
Scénario.....	125
Démo.....	125
Module.....	126
Démo.....	126
Fonctions avancées.....	126
Chapitre 44: MongoDB.....	130
Remarques.....	130
Exemples.....	130
MongoDB avec le pilote C # 1.7 utilisant PowerShell.....	130
J'ai 3 séries de tableaux dans Powershell.....	130
Chapitre 45: Opérateurs spéciaux.....	132
Exemples.....	132
Opérateur d'expression de tableau.....	132
Appel opération.....	132
Opérateur de sourcing.....	132

Chapitre 46: Opérations d'ensemble de base	133
Introduction.....	133
Syntaxe.....	133
Exemples.....	133
Filtrage: Où-Objet / où /?.....	133
Commande: Sort-Object / sort.....	134
Groupement: groupe-objet / groupe.....	135
Projection: Select-Object / select.....	135
Chapitre 47: Paramètres communs	138
Remarques.....	138
Exemples.....	138
Paramètre ErrorAction.....	138
-ErrorAction Continue	138
-ErrorAction Ignore	139
-ErrorAction Enquête	139
-ErrorAction silencieusementContinuer	139
-ErrorAction Stop	139
-ErrorAction Suspend	140
Chapitre 48: Paramètres dynamiques PowerShell	141
Exemples.....	141
Paramètre dynamique "simple".....	141
Chapitre 49: Postes de travail PowerShell	143
Introduction.....	143
Remarques.....	143
Exemples.....	143
Création d'emplois de base.....	143
Gestion de base.....	144
Chapitre 50: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output et Information	146
Remarques.....	146
Exemples.....	146
Sortie d'écriture.....	146

Préférences d'écriture	146
Chapitre 51: Powershell Remoting	148
Remarques.....	148
Exemples.....	148
Activation de la communication à distance PowerShell.....	148
Uniquement pour les environnements autres que les domaines.....	148
Activation de l'authentification de base.....	149
Connexion à un serveur distant via PowerShell.....	149
Exécuter des commandes sur un ordinateur distant.....	149
Avertissement de sérialisation à distance	150
Utilisation des arguments	151
Une bonne pratique pour nettoyer automatiquement les sessions PSSession.....	152
Chapitre 52: Profils Powershell	153
Remarques.....	153
Exemples.....	154
Créer un profil de base.....	154
Chapitre 53: Propriétés calculées	155
Introduction.....	155
Exemples.....	155
Afficher la taille du fichier en Ko - Propriétés calculées.....	155
Chapitre 54: PSScriptAnalyzer - Analyseur de script PowerShell	156
Introduction.....	156
Syntaxe.....	156
Exemples.....	156
Analyse de scripts avec les ensembles de règles prédéfinis intégrés.....	156
Analyse des scripts par rapport à chaque règle intégrée.....	157
Liste toutes les règles intégrées.....	157
Chapitre 55: Reconnaissance Amazon Web Services (AWS).....	158
Introduction.....	158
Exemples.....	158
Détection des étiquettes d'image avec AWS Rekognition.....	158

Comparer la similarité faciale avec AWS Rekognition	159
Chapitre 56: requêtes sql powershell	160
Introduction	160
Paramètres	160
Remarques	160
Exemples	162
SQLExample	162
SQLQuery	162
Chapitre 57: Scripts de signature	164
Remarques	164
Politiques d'exécution	164
Exemples	165
Signer un script	165
Modification de la stratégie d'exécution à l'aide de Set-ExecutionPolicy	165
Contournement de la politique d'exécution pour un seul script	165
Autres politiques d'exécution:	166
Obtenir la politique d'exécution actuelle	166
Obtenir la signature d'un script signé	167
Création d'un certificat de signature de code auto-signé pour le test	167
Chapitre 58: Sécurité et cryptographie	168
Exemples	168
Calcul des codes de hachage d'une chaîne via la cryptographie .Net	168
Chapitre 59: Service de stockage simple Amazon Web Services (AWS) (S3)	169
Introduction	169
Paramètres	169
Exemples	169
Créer un nouveau seau S3	169
Télécharger un fichier local dans un compartiment S3	169
Supprimer un seau S3	170
Chapitre 60: Splatting	171
Introduction	171

Remarques.....	171
Exemples.....	171
Paramètres de splatting.....	171
Passage d'un paramètre Switch à l'aide de la division.....	172
Tuyauterie et éclaboussure.....	172
Division de la fonction de niveau supérieur en une série de fonctions internes.....	173
Chapitre 61: Travailler avec des fichiers XML.....	174
Exemples.....	174
Accéder à un fichier XML.....	174
Création d'un document XML à l'aide de XmlWriter ().....	176
Ajout d'extraits de XML à XmlDocument en cours.....	177
Données d'échantillon.....	177
Document XML.....	177
Nouvelles données.....	178
Modèles.....	179
Ajouter les nouvelles données.....	179
Profit.....	181
Améliorations.....	181
Chapitre 62: Travailler avec des objets.....	182
Exemples.....	182
Mise à jour des objets.....	182
Ajout de propriétés.....	182
Suppression de propriétés.....	182
Créer un nouvel objet.....	183
Option 1: Nouvel objet.....	183
Option 2: Sélectionner un objet.....	183
Option 3: accélérateur de type pscustomobject (PSv3 + requis).....	184
Examiner un objet.....	184
Création d'instances de classes génériques.....	185
Chapitre 63: Travailler avec le pipeline PowerShell.....	187
Introduction.....	187

Syntaxe.....	187
Remarques.....	187
Exemples.....	187
Fonctions d'écriture avec cycle de vie avancé.....	188
Prise en charge de base du pipeline dans les fonctions.....	188
Concept de travail du pipeline.....	189
Chapitre 64: URL encoder / décoder.....	190
Remarques.....	190
Exemples.....	190
Démarrage rapide: encodage.....	190
Démarrage rapide: décodage.....	191
Encode Query String avec `[uri] :: EscapeDataString ()`.....	191
Encoder une chaîne de requête avec `[System.Web.HttpUtility] :: UriEncode ()`.....	192
Décoder l'URL avec `[uri] :: UnescapeDataString ()`.....	192
Décoder l'URL avec `[System.Web.HttpUtility] :: UriDecode ()`.....	194
Chapitre 65: Utilisation du système d'aide.....	197
Remarques.....	197
Exemples.....	197
Mise à jour du système d'aide.....	197
Utilisation de Get-Help.....	197
Affichage de la version en ligne d'une rubrique d'aide.....	198
Exemples d'affichage.....	198
Affichage de la page d'aide complète.....	198
Affichage de l'aide pour un paramètre spécifique.....	198
Chapitre 66: Utiliser des classes statiques existantes.....	199
Introduction.....	199
Exemples.....	199
Créer un nouveau GUID instantanément.....	199
Utiliser la classe Math .Net.....	199
Ajouter des types.....	200
Chapitre 67: Utiliser la barre de progression.....	201
Introduction.....	201

Examples.....	201
Utilisation simple de la barre de progression.....	201
Utilisation de la barre de progression interne.....	202
Chapitre 68: Utiliser ShouldProcess.....	204
Syntaxe.....	204
Paramètres.....	204
Remarques.....	204
Examples.....	204
Ajout de la prise en charge de -WhatIf et -Confirm à votre applet de commande.....	204
Utiliser ShouldProcess () avec un argument.....	204
Exemple d'utilisation complète.....	205
Chapitre 69: Variables Automatiques.....	207
Introduction.....	207
Syntaxe.....	207
Examples.....	207
\$ pid.....	207
Valeurs booléennes.....	207
\$ null.....	207
\$ OFS.....	208
\$ _ / \$ PSItem.....	208
\$?.....	209
\$ erreur.....	209
Chapitre 70: Variables automatiques - partie 2.....	210
Introduction.....	210
Remarques.....	210
Examples.....	210
\$ PSVersionTable.....	210
Chapitre 71: Variables d'environnement.....	211
Examples.....	211
Les variables d'environnement Windows sont visibles en tant que lecteur PS appelé Env:.....	211
Appel instantané des variables d'environnement avec \$ env:.....	211
Chapitre 72: Variables dans PowerShell.....	212

Introduction.....	212
Exemples.....	212
Variable simple.....	212
Supprimer une variable.....	212
Portée.....	212
Lecture d'une sortie CmdLet.....	213
Affectation de liste de variables multiples.....	214
Tableaux.....	214
Ajout à un arry.....	215
Combiner des tableaux ensemble.....	215
Chapitre 73: Variables intégrées.....	216
Introduction.....	216
Exemples.....	216
\$ PSScriptRoot.....	216
\$ Args.....	216
\$ PSItem.....	216
\$?.....	217
\$ erreur.....	217
Chapitre 74: WMI et CIM.....	218
Remarques.....	218
CIM vs WMI.....	218
Ressources additionnelles.....	218
Exemples.....	219
Interroger des objets.....	219
Liste tous les objets pour la classe CIM.....	219
En utilisant un filtre.....	219
En utilisant une requête WQL:.....	220
Classes et espaces de noms.....	221
Liste des classes disponibles.....	221
Rechercher un cours.....	221
Liste des classes dans un autre espace de noms.....	222

Liste des espaces de noms disponibles.....223

Crédits.....224

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [powershell](#)

It is an unofficial and free PowerShell ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official PowerShell.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec PowerShell

Remarques

Windows PowerShell est un composant de shell et de script de Windows Management Framework, une structure de gestion de l'automatisation / configuration de Microsoft basée sur .NET Framework. PowerShell est installé par défaut sur toutes les versions prises en charge des systèmes d'exploitation client et serveur Windows depuis Windows 7 / Windows Server 2008 R2. Powershell peut être mis à jour à tout moment en téléchargeant une version ultérieure de [Windows Management Framework](#) (WMF). La version "Alpha" de PowerShell 6 est multi-plateforme (Windows, Linux et OS X) et doit être téléchargée et installée à partir de [cette page](#) .

Ressources additionnelles:

- Documentation MSDN: <https://msdn.microsoft.com/en-us/powershell/scripting/powershell-scripting>
- TechNet: <https://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx>
 - [A propos des pages](#)
- Galerie PowerShell: <https://www.powershellgallery.com/>
- Blog MSDN: <https://blogs.msdn.microsoft.com/powershell/>
- Github: <https://github.com/powershell>
- Site communautaire: <http://powershell.com/cs/>

Versions

Version	Inclus avec Windows	Remarques	Date de sortie
1.0	XP / Server 2008		2006-11-01
2.0	7 / Server 2008 R2		2009-11-01
3.0	8 / serveur 2012		2012-08-01
4.0	8.1 / Server 2012 R2		2013-11-01
5.0	10 / Server 2016 Tech Preview		2015-12-16
5.1	10 édition anniversaire / Serveur 2016		2017-01-27

Exemples

Installation ou configuration

les fenêtres

PowerShell est inclus avec Windows Management Framework. L'installation et la configuration ne sont pas requises sur les versions modernes de Windows.

Les mises à jour de PowerShell peuvent être effectuées en installant une version plus récente de Windows Management Framework.

Autres plates-formes

La version "Beta" de PowerShell 6 peut être installée sur d'autres plates-formes. Les packages d'installation sont disponibles [ici](#).

Par exemple, PowerShell 6, pour Ubuntu 16.04, est publié pour empaqueter les référentiels afin de faciliter l'installation (et les mises à jour).

Pour installer, procédez comme suit:

```
# Import the public repository GPG keys
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -

# Register the Microsoft Ubuntu repository
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee
/etc/apt/sources.list.d/microsoft.list

# Update apt-get
sudo apt-get update

# Install PowerShell
sudo apt-get install -y powershell

# Start PowerShell
powershell
```

Après avoir enregistré le référentiel Microsoft en tant que superutilisateur, il vous suffit d'utiliser `sudo apt-get upgrade powershell` pour le mettre à jour. Alors lancez simplement `powershell`

Autoriser les scripts stockés sur votre machine à s'exécuter sans signature

Pour des raisons de sécurité, PowerShell est configuré par défaut pour autoriser uniquement l'exécution des scripts signés. L'exécution de la commande suivante vous permettra d'exécuter des scripts non signés (vous devez exécuter PowerShell en tant qu'administrateur pour ce faire).

```
Set-ExecutionPolicy RemoteSigned
```

Une autre façon d'exécuter des scripts PowerShell consiste à utiliser `Bypass` comme `ExecutionPolicy` :

```
powershell.exe -ExecutionPolicy Bypass -File "c:\MyScript.ps1"
```

Ou à partir de votre console PowerShell ou de votre session ISE existante en exécutant:

Une solution de contournement temporaire pour la stratégie d'exécution peut également être obtenue en exécutant le fichier exécutable Powershell et en transmettant toute stratégie valide en tant que paramètre `-ExecutionPolicy`. La stratégie est en vigueur uniquement pendant la durée de vie du processus, de sorte qu'aucun accès administratif au registre n'est nécessaire.

```
C:\>powershell -ExecutionPolicy RemoteSigned
```

Plusieurs autres stratégies sont disponibles et les sites en ligne vous encouragent souvent à utiliser `Set-ExecutionPolicy Unrestricted`. Cette stratégie reste en place jusqu'à ce qu'elle soit modifiée et diminue la position de sécurité du système. Ceci n'est pas conseillé. L'utilisation de `RemoteSigned` est recommandée car elle permet de stocker et d'écrire du code en local et requiert que le code acquis à distance soit signé avec un certificat provenant d'une racine approuvée.

Veillez également à ce que la stratégie d'exécution soit appliquée par la stratégie de groupe, de sorte que la stratégie de groupe, même modifiée, `Unrestricted` pour l'ensemble du système (généralement 15 minutes). Vous pouvez voir la stratégie d'exécution définie sur les différentes étendues à l'aide de `Get-ExecutionPolicy -List`

Documentation TechNet:

[Set-ExecutionPolicy](#)
[about_Execution_Policies](#)

Alias et fonctions similaires

Dans PowerShell, il existe plusieurs manières d'obtenir le même résultat. Ceci peut être bien illustré avec l'exemple simple et familier de `Hello World` :

Utilisation de `Write-Host` :

```
Write-Host "Hello World"
```

Utilisation de `Write-Output` :

```
Write-Output 'Hello world'
```

Il est à noter que bien que `Write-Output` & `Write-Host` écrivent tous deux sur l'écran, il y a une différence subtile. `Write-Host` écrit *uniquement* dans `stdout` (c.-à-d. L'écran de la console), alors que `Write-Output` écrit à la fois sur `stdout` *AND* et sur le flux de sortie `[success]` permettant la [redirection](#). La redirection (et les flux en général) permet de diriger la sortie d'une commande en entrée vers une autre, y compris l'affectation à une variable.

```
> $message = Write-Output "Hello World"  
> $message  
"Hello World"
```

Ces fonctions similaires ne sont pas des alias, mais peuvent produire les mêmes résultats si l'on veut éviter de "polluer" le flux de réussite.

Write-Output est associé à Echo ou Write

```
Echo 'Hello world'  
Write 'Hello world'
```

Ou, simplement en tapant "Hello world"!

```
'Hello world'
```

Tout cela entraînera la sortie de console attendue

```
Hello world
```

Un autre exemple d'alias dans PowerShell est le mappage commun des commandes d'invite de commandes plus anciennes et des commandes BASH aux applets de commande PowerShell. Tous les éléments suivants produisent une liste de répertoires du répertoire en cours.

```
C:\Windows> dir  
C:\Windows> ls  
C:\Windows> Get-ChildItem
```

Enfin, vous pouvez créer votre propre alias avec l'applet de commande Set-Alias! Par exemple, testons Test-NetConnection, qui est essentiellement l'équivalent PowerShell de la commande ping de l'invite de commandes, pour "ping".

```
Set-Alias -Name ping -Value Test-NetConnection
```

Maintenant, vous pouvez utiliser ping au lieu de Test-NetConnection ! Sachez que si l'alias est déjà utilisé, vous écraserez l'association.

L'alias sera vivant jusqu'à ce que la session soit active. Une fois que vous fermez la session et essayez d'exécuter l'alias que vous avez créé lors de votre dernière session, cela ne fonctionnera pas. Pour résoudre ce problème, vous pouvez importer tous vos alias depuis Excel dans votre session une fois, avant de commencer votre travail.

Le pipeline - Utilisation de la sortie d'une applet de commande PowerShell

L'une des premières questions que se posent les utilisateurs lorsqu'ils commencent à utiliser PowerShell pour créer des scripts est la manipulation de la sortie d'une applet de commande pour effectuer une autre action.

Le symbole du pipeline | est utilisé à la fin d'une applet de commande pour prendre les données exportées et les transmettre à la cmdlet suivante. Un exemple simple consiste à utiliser Select-Object pour afficher uniquement la propriété Name d'un fichier affiché dans Get-ChildItem:

```
Get-ChildItem | Select-Object Name
#This may be shortened to:
gci | Select Name
```

L'utilisation plus avancée du pipeline nous permet de canaliser la sortie d'une applet de commande dans une boucle foreach:

```
Get-ChildItem | ForEach-Object {
    Copy-Item -Path $_.FullName -destination C:\NewDirectory\
}

#This may be shortened to:
gci | % { Copy $_.FullName C:\NewDirectory\ }
```

Notez que l'exemple ci-dessus utilise la variable automatique \$_. \$_ est le court alias de \$PSItem qui est une variable automatique qui contient l'élément en cours dans le pipeline.

Commentant

Pour commenter les scripts d'alimentation en ajoutant la ligne au symbole # (dièse)

```
# This is a comment in powershell
Get-ChildItem
```

Vous pouvez également avoir des commentaires sur plusieurs lignes en utilisant <# et #> au début et à la fin du commentaire, respectivement.

```
<#
This is a
multi-line
comment
#>
Get-ChildItem
```

Appeler des méthodes de bibliothèque .Net

Les méthodes de bibliothèque .Net statiques peuvent être appelées à partir de PowerShell en encapsulant le nom de classe complet dans le troisième cadre et en appelant ensuite la méthode en utilisant ::

```
#calling Path.GetFileName()
C:\> [System.IO.Path]::GetFileName('C:\Windows\explorer.exe')
explorer.exe
```

Les méthodes statiques peuvent être appelées à partir de la classe elle-même, mais l'appel de méthodes non statiques nécessite une instance de la classe .Net (un objet).

Par exemple, la méthode AddHours ne peut pas être appelée à partir de la classe System.DateTime elle-même. Il nécessite une instance de la classe:

```
C:\> [System.DateTime]::AddHours(15)
Method invocation failed because [System.DateTime] does not contain a method named 'AddHours'.
At line:1 char:1
+ [System.DateTime]::AddHours(15)
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [], RuntimeException
+ FullyQualifiedErrorId : MethodNotFound
```

Dans ce cas, nous **créons d'abord un objet** , par exemple:

```
C:\> $Object = [System.DateTime]::Now
```

Ensuite, nous pouvons utiliser des méthodes de cet objet, même les méthodes qui ne peuvent pas être appelées directement depuis la classe `System.DateTime`, comme la méthode `AddHours`:

```
C:\> $Object.AddHours(15)

Monday 12 September 2016 01:51:19
```

Créer des objets

La cmdlet `New-Object` est utilisée pour créer un objet.

```
# Create a DateTime object and stores the object in variable "$var"
$var = New-Object System.DateTime

# calling constructor with parameters
$sr = New-Object System.IO.StreamReader -ArgumentList "file path"
```

Dans de nombreux cas, un nouvel objet sera créé afin d'exporter des données ou de le transmettre à une autre commande. Cela peut être fait comme ça:

```
$newObject = New-Object -TypeName PSObject -Property @{
    ComputerName = "SERVER1"
    Role = "Interface"
    Environment = "Production"
}
```

Il y a plusieurs façons de créer un objet. La méthode suivante est probablement le moyen le plus court et le plus rapide de créer un objet `PSCustomObject` :

```
$newObject = [PSCustomObject]@{
    ComputerName = 'SERVER1'
    Role = 'Interface'
    Environment = 'Production'
}
```

Si vous avez déjà un objet, mais que vous n'avez besoin que d'une ou deux propriétés supplémentaires, vous pouvez simplement ajouter cette propriété à l'aide de `Select-Object` :

```
Get-ChildItem | Select-Object FullName, Name,
```

```
@{Name='DateTime'; Expression={Get-Date}},  
@{Name='ProprieteName'; Expression={'CustomValue'}}
```

Tous les objets peuvent être stockés dans des variables ou passés dans le pipeline. Vous pouvez également ajouter ces objets à une collection, puis afficher les résultats à la fin.

Les collections d'objets fonctionnent bien avec Export-CSV (et Import-CSV). Chaque ligne du fichier CSV est un objet, chaque colonne une propriété.

Les commandes de format convertissent les objets en flux de texte pour l'affichage. Évitez d'utiliser les commandes Format- * jusqu'à la dernière étape d'un traitement de données, afin de préserver la convivialité des objets.

Lire Démarrer avec PowerShell en ligne: <https://riptutorial.com/fr/powershell/topic/822/demarrer-avec-powershell>

Chapitre 2: Aide basée sur les commentaires

Introduction

PowerShell dispose d'un mécanisme de documentation appelé aide basée sur les commentaires. Il permet de documenter des scripts et des fonctions avec des commentaires de code. L'aide basée sur les commentaires est la plupart du temps écrite dans des blocs de commentaires contenant plusieurs mots-clés d'aide. Les mots-clés d'aide commencent par des points et identifient les sections d'aide qui seront affichées en exécutant la cmdlet `Get-Help`.

Exemples

Aide basée sur les commentaires de fonction

```
<#  
  
.SYNOPSIS  
    Gets the content of an INI file.  
  
.DESCRIPTION  
    Gets the content of an INI file and returns it as a hashtable.  
  
.INPUTS  
    System.String  
  
.OUTPUTS  
    System.Collections.Hashtable  
  
.PARAMETER FilePath  
    Specifies the path to the input INI file.  
  
.EXAMPLE  
    C:\PS>$IniContent = Get-IniContent -FilePath file.ini  
    C:\PS>$IniContent['Section1'].Key1  
    Gets the content of file.ini and access Key1 from Section1.  
  
.LINK  
    Out-IniFile  
  
#>  
function Get-IniContent  
{  
    [CmdletBinding()]  
    Param  
    (  
        [Parameter(Mandatory=$true,ValueFromPipeline=$true)]  
        [ValidateNotNullOrEmpty()]  
        [ValidateScript({(Test-Path $_) -and ((Get-Item $_).Extension -eq ".ini")})]  
        [System.String]$FilePath  
    )  
  
    # Initialize output hash table.  
    $ini = @{}  
}
```

```

switch -regex -file $FilePath
{
    "^\[([.+])\]" # Section
    {
        $section = $matches[1]
        $ini[$section] = @{}
        $CommentCount = 0
    }
    "(;.*)" # Comment
    {
        if( !($section) )
        {
            $section = "No-Section"
            $ini[$section] = @{}
        }
        $value = $matches[1]
        $CommentCount = $CommentCount + 1
        $name = "Comment" + $CommentCount
        $ini[$section][$name] = $value
    }
    "(.+?)\s*=\s*(.*)" # Key
    {
        if( !($section) )
        {
            $section = "No-Section"
            $ini[$section] = @{}
        }
        $name,$value = $matches[1..2]
        $ini[$section][$name] = $value
    }
}

return $ini
}

```

La documentation de la fonction ci-dessus peut être affichée en exécutant `Get-Help -Name Get-IniContent -Full :`

```
PS C:\Scripts> Get-Help -Name Get-IniContent -Full

NAME
    Get-IniContent

SYNOPSIS
    Gets the content of an INI file.

SYNTAX
    Get-IniContent [-FilePath] <String> [<CommonParameters>]

DESCRIPTION
    Gets the content of an INI file and returns it as a hashtable.

PARAMETERS
    -FilePath <String>
        Specifies the path to the input INI file.

        Required?                true
        Position?                 1
        Default value
        Accept pipeline input?    true (ByValue)
        Accept wildcard characters? false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).

INPUTS
    System.String

OUTPUTS
    System.Collections.Hashtable

----- EXAMPLE 1 -----

C:\PS>$IniContent = Get-IniContent -FilePath file.ini

C:\PS>$IniContent['Section1'].Key1
Gets the content of file.ini and access Key1 from Section1.

RELATED LINKS
    Out-IniFile

PS C:\Scripts>
```

Notez que les mots clés basés sur des commentaires commençant par a . correspondent aux sections de résultats `Get-Help` .

Aide basée sur les commentaires de script

```
<#
.SYNOPSIS
    Reads a CSV file and filters it.
```

```
.DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.

.PARAMETER Path
    Specifies the path of the CSV input file.

.INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.

.OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.

.EXAMPLE
    C:\PS> .\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

#>
Param
(
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $Path,
    [Parameter(Mandatory=$true,ValueFromPipeline=$false)]
    [System.String]
    $UserName
)

Import-Csv -Path $Path | Where-Object -FilterScript {$_.UserName -eq $UserName}
```

La documentation de script ci-dessus peut être affichée en exécutant `Get-Help -Name ReadUsersCsv.ps1 -Full` :

```

PS C:\Scripts> Get-Help -Name .\ReadUsersCsv.ps1 -Full
NAME
    C:\Scripts\ReadUsersCsv.ps1
SYNOPSIS
    Reads a CSV file and filters it.
SYNTAX
    C:\Scripts\ReadUsersCsv.ps1 [-Path] <String> [-UserName] <String> [<CommonParameters>]
DESCRIPTION
    The ReadUsersCsv.ps1 script reads a CSV file and filters it on the 'UserName' column.
PARAMETERS
    -Path <String>
        Specifies the path of the CSV input file.

        Required?                true
        Position?                 1
        Default value
        Accept pipeline input?    false
        Accept wildcard characters? false

    -UserName <String>
        Specifies the user name that will be used to filter the CSV file.

        Required?                true
        Position?                 2
        Default value
        Accept pipeline input?    false
        Accept wildcard characters? false

    <CommonParameters>
        This cmdlet supports the common parameters: Verbose, Debug,
        ErrorAction, ErrorVariable, WarningAction, WarningVariable,
        OutBuffer, PipelineVariable, and OutVariable. For more information, see
        about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216).
INPUTS
    None. You cannot pipe objects to ReadUsersCsv.ps1.
OUTPUTS
    None. ReadUsersCsv.ps1 does not generate any output.

----- EXAMPLE 1 -----
C:\PS>.\ReadUsersCsv.ps1 -Path C:\Temp\Users.csv -UserName j.doe

RELATED LINKS

PS C:\Scripts>

```

Lire Aide basée sur les commentaires en ligne:

<https://riptutorial.com/fr/powershell/topic/9530/aide-basee-sur-les-commentaires>

Chapitre 3: Alias

Remarques

Le système de nommage de Powershell a des règles assez strictes pour nommer les applets de commande (modèle Verb-Noun; voir [sujet non encore créé] pour plus d'informations). Mais il n'est pas très pratique d'écrire `Get-ChildItems` chaque fois que vous souhaitez répertorier les fichiers dans un répertoire de manière interactive.

Par conséquent, Powershell permet d'utiliser des raccourcis (alias) au lieu de noms d'applet de commande.

Vous pouvez écrire `ls`, `dir` ou `gci` au lieu de `Get-ChildItem` et obtenir le même résultat. Alias est équivalent à sa cmdlet.

Certains des alias courants sont:

alias	applet de commande
%, pour chaque	For-EachObject
?, où	Où-objet
cat, gc, type	Obtenir du contenu
cd, chdir, sl	Set-Location
cls, clair	Clear-Host
cp, copie, cpi	Élément de copie
dir / ls / gci	Get-ChildItem
écho, écris	Sortie d'écriture
fl	Format-List
ft	Format-Table
fw	Format à l'échelle
gc, pwd	Get-Location
gm	Get-Member
iex	Invocation-expression
ii	Invoke-Item

alias	applet de commande
mv, bouge	Objet de déménagement
rm, rmdir, del, erase, rd, ri	Retirer l'objet
dormir	Start-Sleep
commence, sape	Processus de démarrage

Dans le tableau ci-dessus, vous pouvez voir comment les alias ont permis de simuler des commandes connues d'autres environnements (cmd, bash), d'où une plus grande facilité de découverte.

Exemples

Get-Alias

Pour lister tous les alias et leurs fonctions:

```
Get-Alias
```

Pour obtenir tous les alias pour une applet de commande spécifique:

```
PS C:\> get-alias -Definition Get-ChildItem
```

CommandType	Name	Version	Source
-----	----	-----	-----
Alias	dir -> Get-ChildItem		
Alias	gci -> Get-ChildItem		
Alias	ls -> Get-ChildItem		

Pour rechercher des alias en faisant correspondre:

```
PS C:\> get-alias -Name p*
```

CommandType	Name	Version	Source
-----	----	-----	-----
Alias	popd -> Pop-Location		
Alias	proc -> Get-Process		
Alias	ps -> Get-Process		
Alias	pushd -> Push-Location		
Alias	pwd -> Get-Location		

Alias Set

Cette applet de commande vous permet de créer de nouveaux noms alternatifs pour les applets de commande existantes.

```
PS C:\> Set-Alias -Name proc -Value Get-Process
```

```
PS C:\> proc
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	-----	--	--	-----
292	17	13052	20444	...19	7.94	620	1	ApplicationFrameHost
....								

Gardez à l'esprit que tous les alias que vous créez ne seront conservés que dans la session en cours. Lorsque vous démarrez une nouvelle session, vous devez à nouveau créer vos alias. Les profils Powershell (voir [sujet non encore créé]) sont parfaits pour ces fins.

Lire Alias en ligne: <https://riptutorial.com/fr/powershell/topic/5287/alias>

Chapitre 4: Analyse CSV

Exemples

Utilisation de base de Import-Csv

Étant donné le fichier CSV suivant

```
String,DateTime,Integer
First,2016-12-01T12:00:00,30
Second,2015-12-01T12:00:00,20
Third,2015-12-01T12:00:00,20
```

On peut importer les lignes CSV dans les objets PowerShell à l'aide de la commande `Import-Csv`

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows

String DateTime          Integer
-----
First  2016-12-01T12:00:00 30
Second 2015-11-03T13:00:00 20
Third  2015-12-05T14:00:00 20

> Write-Host $row[0].String1
Third
```

Importer à partir de CSV et convertir les propriétés pour corriger le type

Par défaut, `Import-Csv` importe toutes les valeurs sous forme de chaînes, donc pour obtenir les objets `DateTime`- et `integer`, nous devons les convertir ou les analyser.

Utiliser `Foreach-Object` :

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | ForEach-Object {
    #Cast properties
    $_.DateTime = [datetime]$_.DateTime
    $_.Integer = [int]$_.Integer

    #Output object
    $_
}
```

Utilisation des propriétés calculées:

```
> $listOfRows = Import-Csv .\example.csv
> $listOfRows | Select-Object String,
    @{name="DateTime";expression={ [datetime]$_.DateTime }},
    @{name="Integer";expression={ [int]$_.Integer }}
```

Sortie:

```
String DateTime      Integer
-----
First 01.12.2016 12:00:00    30
Second 03.11.2015 13:00:00    20
Third 05.12.2015 14:00:00    20
```

Lire Analyse CSV en ligne: <https://riptutorial.com/fr/powershell/topic/5691/analyse-csv>

Chapitre 5: Anonymiser les adresses IP (v4 et v6) dans un fichier texte avec Powershell

Introduction

Manipulation de Regex pour IPv4 et IPv6 et remplacement par une fausse adresse IP dans un fichier journal lu

Exemples

Anonymiser l'adresse IP dans un fichier texte

```
# Read a text file and replace the IPv4 and IPv6 by fake IP Address

# Describe all variables
$SourceFile = "C:\sourcefile.txt"
$IPv4File = "C:\IPV4.txt"
$DestFile = "C:\ANONYM.txt"
$Regex_v4 = "(\\d{1,3}\\.|\\d{1,3}\\.|\\d{1,3}\\.|\\d{1,3})"
$Anonym_v4 = "XXX.XXX.XXX.XXX"
$Regex_v6 = "(((\\[0-9A-Fa-f\\]{1,4}:){7}[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){6}:[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){5}:([0-9A-Fa-f]{1,4})?|[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){4}:([0-9A-Fa-f]{1,4}){0,2}|[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){3}:([0-9A-Fa-f]{1,4}){0,3}|[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){2}:([0-9A-Fa-f]{1,4}){0,4}|[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){6}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(((\\[0-9A-Fa-f\\]{1,4}:){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|(::[0-9A-Fa-f]{1,4}){0,5}((b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b).){3}(b((25[0-5])|(1d{2})|(2[0-4]d)|(d{1,2}))b))|([0-9A-Fa-f]{1,4}::([0-9A-Fa-f]{1,4}){0,5}[0-9A-Fa-f]{1,4})|(::[0-9A-Fa-f]{1,4}){0,6}[0-9A-Fa-f]{1,4})|(((\\[0-9A-Fa-f\\]{1,4}:){1,7}:)))"
$Anonym_v6 = "YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY:YYYY"
$SuffixName = "-ANONYM."
$AnonymFile = ($Parts[0] + $SuffixName + $Parts[1])

# Replace matching IPv4 from sourcefile and creating a temp file IPV4.txt
Get-Content $SourceFile | Foreach-Object {$ _ -replace $Regex_v4, $Anonym_v4} | Set-Content $IPv4File

# Replace matching IPv6 from IPV4.txt and creating a temp file ANONYM.txt
Get-Content $IPv4File | Foreach-Object {$ _ -replace $Regex_v6, $Anonym_v6} | Set-Content $DestFile

# Delete temp IPV4.txt file
Remove-Item $IPv4File

# Rename ANONYM.txt in sourcefile-ANONYM.txt
$Parts = $SourceFile.Split(".")
If (Test-Path $AnonymFile)
{
    Remove-Item $AnonymFile
    Rename-Item $DestFile -NewName $AnonymFile
}
```

```
Else
{
  Rename-Item $DestFile -NewName $AnonymFile
}
```

Lire Anonymiser les adresses IP (v4 et v6) dans un fichier texte avec Powershell en ligne:
<https://riptutorial.com/fr/powershell/topic/9171/anonymiser-les-adresses-ip--v4-et-v6--dans-un-fichier-texte-avec-powershell>

Chapitre 6: Application des prérequis du script

Syntaxe

- #Requires -Version <N> [. <N>]
- #Requires -PSSnapin <Nom PSSnapin> [-Version <N> [. <N>]]
- #Requires -Modules {<Nom-module> | <Hashtable>}
- #Requires -ShellId <ShellId>
- #Requires -RunAsAdministrator

Remarques

#requires instruction #requires peut être placée sur n'importe quelle ligne du script (il ne doit pas nécessairement s'agir de la première ligne), mais il doit s'agir de la première instruction sur cette ligne.

Plusieurs instructions #requires peuvent être utilisées dans un script.

Pour plus de référence, veuillez vous référer à la documentation officielle sur Technet - [about_about_Requires](#) .

Exemples

Appliquer une version minimale de l'hôte powershell

```
#requires -version 4
```

Après avoir essayé d'exécuter ce script dans la version inférieure, vous verrez ce message d'erreur

```
. \script.ps1: Le script 'script.ps1' ne peut pas être exécuté car il contient une instruction "#requires" à la ligne 1 pour Windows PowerShell version 5.0. La version requise par le script ne correspond pas à la version en cours d'exécution de Windows PowerShell version 2.0.
```

Exécuter le script en tant qu'administrateur

4.0

```
#requires -RunAsAdministrator
```

Après avoir essayé d'exécuter ce script sans privilèges d'administrateur, vous verrez ce message

d'erreur

. \ script.ps1: Le script 'script.ps1' ne peut pas être exécuté car il contient une instruction "#requires" pour s'exécuter en tant qu'administrateur. La session Windows PowerShell en cours ne s'exécute pas en tant qu'administrateur. Démarrez Windows PowerShell en utilisant l'option Exécuter en tant qu'administrateur, puis essayez à nouveau d'exécuter le script.

Lire [Application des prérequis du script en ligne](https://riptutorial.com/fr/powershell/topic/5637/application-des-prerequis-du-script):

<https://riptutorial.com/fr/powershell/topic/5637/application-des-prerequis-du-script>

Chapitre 7: Automatisation de l'infrastructure

Introduction

L'automatisation des services de gestion d'infrastructure permet de réduire les ETP et d'obtenir un meilleur retour sur investissement cumulatif à l'aide de plusieurs outils, orchestrateurs, moteur d'orchestration, scripts et interface utilisateur facile

Exemples

Script simple pour le test d'intégration des boîtes noires des applications de la console

Ceci est un exemple simple sur la façon dont vous pouvez automatiser les tests pour une application console qui interagit avec une entrée standard et une sortie standard.

L'application testée lit et additionne chaque nouvelle ligne et fournit le résultat après qu'une seule ligne blanche soit fournie. Le script Power Shell écrit "pass" lorsque la sortie correspond.

```
$process = New-Object System.Diagnostics.Process
$process.StartInfo.FileName = ".\ConsoleApp1.exe"
$process.StartInfo.UseShellExecute = $false
$process.StartInfo.RedirectStandardOutput = $true
$process.StartInfo.RedirectStandardInput = $true
if ( $process.Start() ) {
    # input
    $process.StandardInput.WriteLine("1");
    $process.StandardInput.WriteLine("2");
    $process.StandardInput.WriteLine("3");
    $process.StandardInput.WriteLine();
    $process.StandardInput.WriteLine();
    # output check
    $output = $process.StandardOutput.ReadToEnd()
    if ( $output ) {
        if ( $output.Contains("sum 6") ) {
            Write "pass"
        }
        else {
            Write-Error $output
        }
    }
    $process.WaitForExit()
}
```

Lire Automatisation de l'infrastructure en ligne:

<https://riptutorial.com/fr/powershell/topic/10909/automatisation-de-l-infrastructure>

Chapitre 8: Boucles

Introduction

Une boucle est une séquence d'instructions répétée continuellement jusqu'à ce qu'une certaine condition soit atteinte. Pouvoir faire exécuter à plusieurs reprises un bloc de code par votre programme est l'une des tâches les plus élémentaires mais utiles de la programmation. Une boucle vous permet d'écrire une instruction très simple pour produire un résultat nettement supérieur simplement par répétition. Si la condition a été atteinte, l'instruction suivante "passe" à l'instruction séquentielle suivante ou se branche en dehors de la boucle.

Syntaxe

- pour (<Initialisation>; <Condition>; <Répétition>) {<Script_Block>}
- <Collection> | Foreach-Object {<Script_Block_with _ \$ __ as_current_item>}
- foreach (<Item> dans <Collection>) {<Script_Block>}
- while (<Condition>) {<Script_Block>}
- faire {<Script_Block>} while (<Condition>)
- faire {<Script_Block>} jusqu'à (<Condition>)
- <Collection> .foreach ({<Script_Block_with _ \$ __ as_current_item>})

Remarques

Pour chaque

Il existe plusieurs manières d'exécuter une boucle foreach dans PowerShell et elles apportent toutes leurs avantages et inconvénients:

Solution	Avantages	Désavantages
Déclaration de foreach	Le plus rapide. Fonctionne mieux avec les collections statiques (stockées dans une variable).	Aucune entrée ou sortie de pipeline
Méthode ForEach ()	Même syntaxe de <code>Foreach-Object</code> de <code>Foreach-Object</code> que <code>Foreach-Object</code> , mais plus rapide. Fonctionne mieux avec les collections statiques (stockées dans une variable). Prend en charge la sortie du pipeline.	Pas de support pour les entrées de pipeline. Nécessite PowerShell 4.0 ou supérieur

Solution	Avantages	Désavantages
Foreach-Object (cmdlet)	Prend en charge les entrées et sorties de pipeline. Prise en charge des blocs de début et de fin de script pour l'initialisation et la fermeture des connexions, etc. Solution la plus flexible.	Le plus lent

Performance

```
$foreach = Measure-Command { foreach ($i in (1..1000000)) { $i * $i } }
$foreachmethod = Measure-Command { (1..1000000).ForEach{ $_ * $_ } }
$foreachobject = Measure-Command { (1..1000000) | ForEach-Object { $_ * $_ } }

"Foreach: $($foreach.TotalSeconds) "
"Foreach method: $($foreachmethod.TotalSeconds) "
"ForEach-Object: $($foreachobject.TotalSeconds) "
```

Example output:

```
Foreach: 1.9039875
Foreach method: 4.7559563
ForEach-Object: 10.7543821
```

Alors que `Foreach-Object` est le plus lent, son support par pipeline peut être utile car il vous permet de traiter les éléments à mesure qu'ils arrivent (lors de la lecture d'un fichier, de la réception de données, etc.). Cela peut être très utile lorsque vous travaillez avec des données volumineuses et de la mémoire faible, car vous n'avez pas besoin de charger toutes les données en mémoire avant le traitement.

Exemples

Pour

```
for($i = 0; $i -le 5; $i++){
    "$i"
}
```

Une utilisation typique de la boucle `for` consiste à opérer sur un sous-ensemble des valeurs d'un tableau. Dans la plupart des cas, si vous souhaitez parcourir toutes les valeurs d'un tableau, envisagez d'utiliser une instruction `foreach`.

Pour chaque

`ForEach` a deux significations différentes dans PowerShell. L'un est un [mot - clé](#) et l'autre est un [alias](#) pour l' [applet de commande ForEach-Object](#) . Le premier est décrit ici.

Cet exemple montre comment imprimer tous les éléments d'un tableau sur l'hôte de la console:

```
$Names = @('Amy', 'Bob', 'Celine', 'David')

ForEach ($Name in $Names)
{
    Write-Host "Hi, my name is $Name!"
}
```

Cet exemple montre comment capturer la sortie d'une boucle ForEach:

```
$Numbers = ForEach ($Number in 1..20) {
    $Number # Alternatively, Write-Output $Number
}
```

Comme dans le dernier exemple, cet exemple montre plutôt la création d'un tableau avant le stockage de la boucle:

```
$Numbers = @()
ForEach ($Number in 1..20)
{
    $Numbers += $Number
}
```

Tandis que

Une boucle while évalue une condition et si true exécute une action. Tant que la condition est vraie, l'action continuera d'être effectuée.

```
while(condition){
    code_block
}
```

L'exemple suivant crée une boucle qui comptera de 10 à 0

```
$i = 10
while($i -ge 0){
    $i
    $i--
}
```

Contrairement à la boucle `do -While`, la condition est évaluée avant la première exécution de l'action. L'action ne sera pas effectuée si la condition initiale est fausse.

Remarque: Lors de l'évaluation de la condition, PowerShell considère que l'existence d'un objet de retour est vraie. Cela peut être utilisé de plusieurs manières, mais ci-dessous est un exemple pour surveiller un processus. Cet exemple va engendrer un processus de bloc-notes et ensuite dormir le shell en cours tant que ce processus est en cours d'exécution. Lorsque vous fermez manuellement l'instance du bloc-notes, la condition while va échouer et la boucle sera interrompue.

```
Start-Process notepad.exe
while(Get-Process notepad -ErrorAction SilentlyContinue){
```

```
Start-Sleep -Milliseconds 500
}
```

ForEach-Object

L' `ForEach-Object` fonctionne de manière similaire à l'instruction `foreach` , mais tire son entrée du pipeline.

Utilisation de base

```
$object | ForEach-Object {
    code_block
}
```

Exemple:

```
$names = @("Any", "Bob", "Celine", "David")
$names | ForEach-Object {
    "Hi, my name is $_!"
}
```

`ForEach-Object` a deux alias par défaut, `foreach` et `%` (syntaxe abrégée). Le plus courant est `%` car `foreach` peut être confondu avec l' [instruction foreach](#) . Exemples:

```
$names | % {
    "Hi, my name is $_!"
}

$names | foreach {
    "Hi, my name is $_!"
}
```

Utilisation avancée

`ForEach-Object` se distingue des solutions `foreach` alternatives car il s'agit d'une applet de commande qui signifie qu'il est conçu pour utiliser le pipeline. De ce fait, il prend en charge trois blocs de script, tout comme une applet de commande ou une fonction avancée:

- **Begin** : Exécuté une fois avant de parcourir les éléments qui arrivent du pipeline. Généralement utilisé pour créer des fonctions à utiliser dans la boucle, créer des variables, ouvrir des connexions (base de données, web +), etc.
- **Processus** : exécuté une fois par article arrivé du pipeline. "Normal" foreach codeblock. C'est la valeur par défaut utilisée dans les exemples ci-dessus lorsque le paramètre n'est pas spécifié.
- **End** : Exécuté une fois après avoir traité tous les éléments. Généralement utilisé pour fermer des connexions, générer un rapport, etc.

Exemple:

```

", "Bob", "Celine", "David" | ForEach-Object -Begin {
    $results = @()
} -Process {
    #Create and store message
    $results += "Hi, my name is $_!"
} -End {
    #Count messages and output
    Write-Host "Total messages: $($results.Count)"
    $results
}

```

Faire

Les Do-Loops sont utiles lorsque vous voulez toujours exécuter au moins une fois un bloc de code. Une Do-loop évaluera la condition après l'exécution du codeblock, contrairement à une boucle while qui le fait avant d'exécuter le codeblock.

Vous pouvez utiliser les boucles de deux manières:

- Boucle tant *que* la condition est vraie:

```

Do {
    code_block
} while (condition)

```

- Boucle *jusqu'à ce que* la condition soit vraie, en d'autres termes, boucle pendant que la condition est fausse:

```

Do {
    code_block
} until (condition)

```

Exemples réels:

```

$i = 0

Do {
    $i++
    "Number $i"
} while ($i -ne 3)

Do {
    $i++
    "Number $i"
} until ($i -eq 3)

```

Do-While et Do-Until sont des boucles antonymes. Si le code à l'intérieur du même, la condition sera inversée. L'exemple ci-dessus illustre ce comportement.

Méthode ForEach ()

4.0

Au lieu de l' `ForEach-Object` , il existe également la possibilité d'utiliser une méthode `ForEach` directement sur les tableaux d'objets comme

```
(1..10).ForEach({$_ * $_})
```

ou - si vous le souhaitez - les parenthèses autour du bloc de script peuvent être omises

```
(1..10).ForEach{$_ * $_}
```

Les deux résulteront en la sortie ci-dessous

```
1
4
9
16
25
36
49
64
81
100
```

Continuer

L'opérateur `Continue` fonctionne dans les boucles `For` , `ForEach` , `While` et `Do` Il ignore l'itération actuelle de la boucle, en sautant au sommet de la boucle la plus interne.

```
$i =0
while ($i -lt 20) {
    $i++
    if ($i -eq 7) { continue }
    Write-Host $i
}
```

Le résultat ci-dessus produira 1 à 20 sur la console mais manquera le numéro 7.

Remarque : Lorsque vous utilisez une boucle de pipeline, vous devez utiliser `return` au lieu de `Continue` .

Pause

L'opérateur de `break` sortira immédiatement d'une boucle de programme. Il peut être utilisé dans les boucles `For` , `ForEach` , `While` et `Do` ou dans une instruction `Switch` .

```
$i = 0
while ($i -lt 15) {
    $i++
    if ($i -eq 7) {break}
    Write-Host $i
}
```

Ce qui précède comptera jusqu'à 15 mais s'arrêtera dès que 7 sera atteint.

Remarque : Lorsque vous utilisez une boucle de pipeline, la `break` se comportera comme si elle se continue . Pour simuler des `break` dans la boucle de pipeline, vous devez intégrer une logique, une cmdlet, etc. supplémentaires. Il est plus facile de rester en contact avec des boucles sans pipeline si vous avez besoin d'utiliser `break` .

Étiquettes de rupture

Break peut également appeler une étiquette placée devant l'instanciation d'une boucle:

```
$i = 0
:mainLoop While ($i -lt 15) {
    Write-Host $i -ForegroundColor 'Cyan'
    $j = 0
    While ($j -lt 15) {
        Write-Host $j -ForegroundColor 'Magenta'
        $k = $i*$j
        Write-Host $k -ForegroundColor 'Green'
        if ($k -gt 100) {
            break mainLoop
        }
        $j++
    }
    $i++
}
```

Remarque: Ce code incrémente `$i` à 8 et `$j` à 13 ce qui fait que `$k` égal à 104 . Comme `$k` dépasse 100 , le code va alors sortir des deux boucles.

Lire Boucles en ligne: <https://riptutorial.com/fr/powershell/topic/1067/boucles>

Chapitre 9: Classes PowerShell

Introduction

Une classe est un modèle de code de programme extensible permettant de créer des objets, fournissant des valeurs initiales pour l'état (variables membres) et des implémentations de comportement (fonctions ou méthodes membres). Une classe est un modèle pour un objet. Il est utilisé comme modèle pour définir la structure des objets. Un objet contient des données auxquelles nous accédons via des propriétés et que nous pouvons utiliser en utilisant des méthodes. PowerShell 5.0 a ajouté la possibilité de créer vos propres classes.

Exemples

Méthodes et propriétés

```
class Person {
    [string] $FirstName
    [string] $LastName
    [string] Greeting() {
        return "Greetings, {0} {1}!" -f $this.FirstName, $this.LastName
    }
}

$x = [Person]::new()
$x.FirstName = "Jane"
$x.LastName = "Doe"
$greeting = $x.Greeting() # "Greetings, Jane Doe!"
```

Liste des constructeurs disponibles pour une classe

5.0

Dans PowerShell 5.0+, vous pouvez répertorier les constructeurs disponibles en appelant la `new` méthode statique sans parenthèses.

```
PS> [DateTime]::new

OverloadDefinitions
-----
datetime new(long ticks)
datetime new(long ticks, System.DateTimeKind kind)
datetime new(int year, int month, int day)
datetime new(int year, int month, int day, System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
```

```

System.DateTimeKind kind)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar)
datetime new(int year, int month, int day, int hour, int minute, int second, int millisecond,
System.Globalization.Calendar calendar, System.DateTimeKind kind)

```

C'est la même technique que vous pouvez utiliser pour répertorier les définitions de surcharge pour n'importe quelle méthode.

```

> 'abc'.CompareTo

OverloadDefinitions
-----
int CompareTo(System.Object value)
int CompareTo(string strB)
int IComparable.CompareTo(System.Object obj)
int IComparable[string].CompareTo(string other)

```

Pour les versions antérieures, vous pouvez créer votre propre fonction pour répertorier les constructeurs disponibles:

```

function Get-Constructor {
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true)]
        [type]$type
    )

    Process {
        $type.GetConstructors() |
        Format-Table -Wrap @{
            n="$($type.Name) Constructors"
            e={ ($_.GetParameters() | % { $_.ToString() }) -Join ", " }
        }
    }
}

```

Usage:

```

Get-Constructor System.DateTime
#Or [datetime] | Get-Constructor

DateTime Constructors
-----
Int64 ticks
Int64 ticks, System.DateTimeKind kind
Int32 year, Int32 month, Int32 day
Int32 year, Int32 month, Int32 day, System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second,
System.Globalization.Calendar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.DateTimeKind kind
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,

```

```
System.Globalization.Cal
endar calendar
Int32 year, Int32 month, Int32 day, Int32 hour, Int32 minute, Int32 second, Int32 millisecond,
System.Globalization.Cal
endar calendar, System.DateTimeKind kind
```

Surcharge du constructeur

```
class Person {
    [string] $Name
    [int] $Age

    Person([string] $Name) {
        $this.Name = $Name
    }

    Person([string] $Name, [int]$Age) {
        $this.Name = $Name
        $this.Age = $Age
    }
}
```

Obtenir tous les membres d'une instance

```
PS > Get-Member -InputObject $anObjectInstance
```

Cela retournera tous les membres de l'instance de type. Voici une partie d'un exemple de sortie pour l'instance String

```
TypeName: System.String

Name                MemberType          Definition
----                -
Clone               Method              System.Object Clone(), System.Object ICloneable.Clone()
CompareTo           Method              int CompareTo(System.Object value), int
CompareTo(string strB), i...
Contains            Method              bool Contains(string value)
CopyTo              Method              void CopyTo(int sourceIndex, char[] destination, int
destinationI...
EndsWith            Method              bool EndsWith(string value), bool EndsWith(string
value, System.S...
Equals              Method              bool Equals(System.Object obj), bool Equals(string
value), bool E...
GetEnumerator        Method              System.CharEnumerator GetEnumerator(),
System.Collections.Generic...
GetHashCode         Method              int GetHashCode()
GetType             Method              type GetType()
...
```

Modèle de classe de base

```
# Define a class
class TypeName
{
```

```

# Property with validate set
[ValidateSet("val1", "Val2")]
[string] $P1

# Static property
static [hashtable] $P2

# Hidden property does not show as result of Get-Member
hidden [int] $P3

# Constructor
TypeName ([string] $s)
{
    $this.P1 = $s
}

# Static method
static [void] MemberMethod1([hashtable] $h)
{
    [TypeName]::P2 = $h
}

# Instance method
[int] MemberMethod2([int] $i)
{
    $this.P3 = $i
    return $this.P3
}
}

```

Héritage de la classe parent à la classe enfant

```

class ParentClass
{
    [string] $Message = "Its under the Parent Class"

    [string] GetMessage()
    {
        return ("Message: {0}" -f $this.Message)
    }
}

# Bar extends Foo and inherits its members
class ChildClass : ParentClass
{
}

$Inherit = [ChildClass]::new()

```

SO, **\$ Inherit.Message** vous donnera le

"Son sous la classe parente"

Lire Classes PowerShell en ligne: <https://riptutorial.com/fr/powershell/topic/1146/classes-powershell>

Chapitre 10: Cmdlet Naming

Introduction

CmdLets devrait être nommé en utilisant un schéma de dénomination `<verb>-<noun>` afin d'améliorer la découverte.

Exemples

Verbes

Les verbes utilisés pour nommer CmdLets doivent être nommés à partir des verbes de la liste fournie par `Get-Verb`

Vous trouverez plus de détails sur l'utilisation des verbes dans les [verbes approuvés pour Windows PowerShell](#).

Des noms

Les noms doivent toujours être singuliers.

Soyez cohérent avec les noms. Par exemple, `Find-Package` besoin d'un fournisseur. Le nom est `PackageProvider` **non** `ProviderPackage` .

Lire Cmdlet Naming en ligne: <https://riptutorial.com/fr/powershell/topic/8703/cmdlet-naming>

Chapitre 11: Comment télécharger le dernier artefact d'Artifactory en utilisant le script Powershell (v2.0 ou inférieur)?

Introduction

Cette documentation explique et fournit des étapes pour télécharger le dernier artefact d'un référentiel JFrog Artifactory en utilisant Powershell Script (version 2.0 ou inférieure).

Exemples

Script Powershell pour télécharger le dernier artefact

```
$username = 'user'
$password = 'password'
$DESTINATION = "D:\test\latest.tar.gz"
$client = New-Object System.Net.WebClient
$client.Credentials = new-object System.Net.NetworkCredential($username, $password)
$lastModifiedResponse =
$client.DownloadString('https://domain.org.com/artifactory/api/storage/FOLDER/repo/?lastModified')

[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestModifiedResponse = $serializer.DeserializeObject($lastModifiedResponse)
$downloadUriResponse = $getLatestModifiedResponse.uri
Write-Host $json.uri
$latestArtifactUrlResponse=$client.DownloadString($downloadUriResponse)
[System.Reflection.Assembly]::LoadWithPartialName("System.Web.Extensions")
$serializer = New-Object System.Web.Script.Serialization.JavaScriptSerializer
$getLatestArtifact = $serializer.DeserializeObject($latestArtifactUrlResponse)
Write-Host $getLatestArtifact.downloadUri
$SOURCE=$getLatestArtifact.downloadUri
$client.DownloadFile($SOURCE,$DESTINATION)
```

Lire Comment télécharger le dernier artefact d'Artifactory en utilisant le script Powershell (v2.0 ou inférieur)? en ligne: <https://riptutorial.com/fr/powershell/topic/8883/comment-telecharger-le-dernier-artefact-d-artifactory-en-utilisant-le-script-powershell--v2-0-ou-inferieur-->

Chapitre 12: Communication avec les API RESTful

Introduction

REST signifie Representational State Transfer (parfois orthographié «ReST»). Il s'appuie sur un protocole de communication sans serveur, client-serveur, apte à la mise en cache et le protocole HTTP est principalement utilisé. Il est principalement utilisé pour créer des services Web légers, maintenables et évolutifs. Un service basé sur REST est appelé service RESTful et les API utilisées sont des API RESTful. Dans PowerShell, `Invoke-RestMethod` est utilisé pour les gérer.

Exemples

Utilisez les Webhooks entrants Slack.com

Définissez votre charge utile pour envoyer d'éventuelles données plus complexes

```
$Payload = @{ text="test string"; username="testuser" }
```

Utilisez l'applet de `ConvertTo-Json` et `Invoke-RestMethod` pour exécuter l'appel

```
Invoke-RestMethod -Uri "https://hooks.slack.com/services/yourwebhookstring" -Method Post -Body  
(ConvertTo-Json $Payload)
```

Poster un message à hipChat

```
$params = @{  
  Uri = "https://your.hipchat.com/v2/room/934419/notification?auth_token=???"  
  Method = "POST"  
  Body = @{  
    color = 'yellow'  
    message = "This is a test message!"  
    notify = $false  
    message_format = "text"  
  } | ConvertTo-Json  
  ContentType = 'application/json'  
}  
  
Invoke-RestMethod @params
```

Utilisation de REST avec des objets PowerShell pour obtenir et mettre des données individuelles

OBTENEZ vos données REST et stockez-les dans un objet PowerShell:

```
$Post = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Modifiez vos données:

```
$Post.title = "New Title"
```

REMETTRE les données REST

```
$Json = $Post | ConvertTo-Json  
Invoke-RestMethod -Method Put -Uri "http://jsonplaceholder.typicode.com/posts/1" -Body $Json -  
ContentType 'application/json'
```

Utilisation de REST avec des objets PowerShell pour GET et POST de nombreux éléments

OBTENEZ vos données REST et stockez-les dans un objet PowerShell:

```
$Users = Invoke-RestMethod -Uri "http://jsonplaceholder.typicode.com/users"
```

Modifier plusieurs éléments dans vos données:

```
$Users[0].name = "John Smith"  
$Users[0].email = "John.Smith@example.com"  
$Users[1].name = "Jane Smith"  
$Users[1].email = "Jane.Smith@example.com"
```

POST toutes les données REST en arrière:

```
$Json = $Users | ConvertTo-Json  
Invoke-RestMethod -Method Post -Uri "http://jsonplaceholder.typicode.com/users" -Body $Json -  
ContentType 'application/json'
```

Utilisation de REST avec PowerShell pour supprimer des éléments

Identifiez l'élément à supprimer et supprimez-le:

```
Invoke-RestMethod -Method Delete -Uri "http://jsonplaceholder.typicode.com/posts/1"
```

Lire Communication avec les API RESTful en ligne:

<https://riptutorial.com/fr/powershell/topic/3869/communication-avec-les-api-restful>

Chapitre 13: Communication TCP avec PowerShell

Exemples

TCP listener

```
Function Receive-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [int] $Port
    )
    Process {
        Try {
            # Set up endpoint and start listening
            $endpoint = new-object System.Net.IPEndPoint([ipaddress]::any, $port)
            $listener = new-object System.Net.Sockets.TcpListener $EndPoint
            $listener.start()

            # Wait for an incoming connection
            $data = $listener.AcceptTcpClient()

            # Stream setup
            $stream = $data.GetStream()
            $bytes = New-Object System.Byte[] 1024

            # Read data from stream and write it to host
            while (($i = $stream.Read($bytes,0,$bytes.Length)) -ne 0){
                $EncodedText = New-Object System.Text.ASCIIEncoding
                $data = $EncodedText.GetString($bytes,0, $i)
                Write-Output $data
            }

            # Close TCP connection and stop listening
            $stream.close()
            $listener.stop()
        }
        Catch {
            "Receive Message failed with: `n" + $Error[0]
        }
    }
}
```

Commencez à écouter avec les éléments suivants et capturez tout message dans la variable `$msg` :

```
$msg = Receive-TCPMessage -Port 29800
```

TCP expéditeur

```

Function Send-TCPMessage {
    Param (
        [Parameter(Mandatory=$true, Position=0)]
        [ValidateNotNullOrEmpty()]
        [string]
        $EndPoint

        ,

        [Parameter(Mandatory=$true, Position=1)]
        [int]
        $Port

        ,

        [Parameter(Mandatory=$true, Position=2)]
        [string]
        $Message
    )
    Process {
        # Setup connection
        $IP = [System.Net.Dns]::GetHostAddresses($EndPoint)
        $Address = [System.Net.IPAddress]::Parse($IP)
        $Socket = New-Object System.Net.Sockets.TCPClient($Address,$Port)

        # Setup stream writer
        $Stream = $Socket.GetStream()
        $Writer = New-Object System.IO.StreamWriter($Stream)

        # Write message to stream
        $Message | % {
            $Writer.WriteLine($_)
            $Writer.Flush()
        }

        # Close connection and stream
        $Stream.Close()
        $Socket.Close()
    }
}

```

Envoyer un message avec:

```
Send-TCPMessage -Port 29800 -Endpoint 192.168.0.1 -message "My first TCP message !"
```

Remarque : les messages TCP peuvent être bloqués par votre pare-feu logiciel ou par tout autre pare-feu externe auquel vous tentez de répondre. Assurez-vous que le port TCP défini dans la commande ci-dessus est ouvert et que vous avez configuré l'écouteur sur le même port.

Lire Communication TCP avec PowerShell en ligne:

<https://riptutorial.com/fr/powershell/topic/5125/communication-tcp-avec-powershell>

Chapitre 14: Comportement de retour dans PowerShell

Introduction

Il peut être utilisé pour quitter la portée actuelle, qui peut être une fonction, un script ou un bloc de script. Dans PowerShell, le résultat de chaque instruction est renvoyé en tant que sortie, même sans mot-clé Return explicite ou pour indiquer que la fin de la portée a été atteinte.

Remarques

Vous pouvez en savoir plus sur la sémantique de retour sur la page [about_Return](#) de TechNet ou en appelant `get-help return` partir d'une invite PowerShell.

Question (s) Q & A notable (s) avec plus d'exemples / d'explications:

- [Valeur de retour de fonction dans PowerShell](#)
- [PowerShell: la fonction n'a pas de valeur de retour correcte](#)

[about_return](#) sur MSDN l'explique succinctement:

Le mot-clé Return permet de quitter une fonction, un script ou un bloc de script. Il peut être utilisé pour quitter une étendue à un point spécifique, pour renvoyer une valeur ou pour indiquer que la fin de la portée a été atteinte.

Les utilisateurs familiarisés avec des langages tels que C ou C # peuvent vouloir utiliser le mot-clé Return pour rendre la logique de laisser une portée explicite.

Dans Windows PowerShell, les résultats de chaque instruction sont renvoyés en tant que sortie, même sans instruction contenant le mot clé Return. Les langages comme C ou C # renvoient uniquement la ou les valeurs spécifiées par le mot-clé Return.

Exemples

Sortie anticipée

```
function earlyexit {  
    "Hello"  
    return  
    "World"  
}
```

"Bonjour" sera placé dans le pipeline de sortie, "Monde" ne sera pas

Je t'ai eu! Retour dans le pipeline

```
get-childitem | foreach-object { if ($_.IsReadOnly) { return } }
```

Les applets de commande de pipeline (par exemple, `ForEach-Object`, `Where-Object`, etc.) fonctionnent sur les fermetures. Le retour ici ne déplacera que le prochain élément du pipeline, et ne quittera pas le traitement. Vous pouvez utiliser **break** au lieu de **return** si vous souhaitez quitter le traitement.

```
get-childitem | foreach-object { if ($_.IsReadOnly) { break } }
```

Je t'ai eu! Ignorer les sorties indésirables

Inspiré par

- [PowerShell: la fonction n'a pas de valeur de retour correcte](#)

```
function bar {  
    [System.Collections.ArrayList]$MyVariable = @()  
    $MyVariable.Add("a") | Out-Null  
    $MyVariable.Add("b") | Out-Null  
    $MyVariable  
}
```

Le `Out-Null` est nécessaire car la méthode `.NET ArrayList.Add` renvoie le nombre d'éléments dans la collection après l'ajout. Si omis, le pipeline aurait contenu `1, 2, "a", "b"`

Il existe plusieurs façons d'omettre les sorties indésirables:

```
function bar  
{  
    # New-Item cmdlet returns information about newly created file/folder  
    New-Item "test1.txt" | out-null  
    New-Item "test2.txt" > $null  
    [void](New-Item "test3.txt")  
    $tmp = New-Item "test4.txt"  
}
```

Remarque: pour en savoir plus sur les raisons de préférer `> $null`, voir [sujet non encore créé].

Retour avec une valeur

(paraphrasé de [about_return](#))

Les méthodes suivantes auront les mêmes valeurs sur le pipeline

```
function foo {  
    $a = "Hello"  
    return $a  
}
```

```
function bar {
    $a = "Hello"
    $a
    return
}

function quux {
    $a = "Hello"
    $a
}
```

Comment travailler avec les fonctions retourne

Une fonction renvoie tout ce qui n'est pas capturé par autre chose.

Si vous utilisez le mot-clé **return**, chaque instruction après la ligne de retour ne sera pas exécutée!

Comme ça:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    "Start"
    if($ExceptionalReturn){Return "Damn, it didn't work!"}
    New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
    Return "Yes, it worked!"
}
```

Fonction de test

Reviendra:

- Début
- La clé de registre nouvellement créée (ceci est dû au fait que certaines instructions créent une sortie inattendue)
- Oui, ça a marché!

Test-Function -ExceptionalReturn Reviendra:

- Début
- Bon sang, ça n'a pas marché!

Si vous le faites comme ceci:

```
Function Test-Function
{
    Param
    (
        [switch]$ExceptionalReturn
    )
    . {
```

```
"Start"
if($ExceptionalReturn)
{
    $Return = "Damn, it didn't work!"
    Return
}
New-ItemProperty -Path "HKCU:\" -Name "test" -Value "TestValue" -Type "String"
$Return = "Yes, it worked!"
Return
} | Out-Null
Return $Return
}
```

Fonction de test

Reviendra:

- Oui, ça a marché!

Test-Fonction -ExceptionalReturn Reviendra:

- Bon sang, ça n'a pas marché!

Avec cette astuce, vous pouvez contrôler la sortie renvoyée, même si vous n'êtes pas sûr de la nature de chaque instruction.

Ça marche comme ça

```
.{<Statements>} | Out-Null
```

la . rend le scriptblock suivant inclus dans le code

le {} marque le bloc de script

le | Out-Null pousse toute sortie inattendue vers Out-Null (donc c'est parti!)

Parce que le scriptblock est inclus, il obtient la même étendue que le reste de la fonction.

Vous pouvez donc accéder aux variables qui ont été créées dans le scriptblock.

Lire Comportement de retour dans PowerShell en ligne:

<https://riptutorial.com/fr/powershell/topic/4781/comportement-de-retour-dans-powershell>

Chapitre 15: Configuration d'état souhaitée

Exemples

Exemple simple - Activation de WindowsFeature

```
configuration EnableIISFeature
{
    node localhost
    {
        WindowsFeature IIS
        {
            Ensure = "Present"
            Name = "Web-Server"
        }
    }
}
```

Si vous exécutez cette configuration dans Powershell (EnableIISFeature), il produira un fichier localhost.mof. C'est la configuration "compilée" que vous pouvez exécuter sur une machine.

Pour tester la configuration DSC sur votre hôte local, vous pouvez simplement appeler les éléments suivants:

```
Start-DscConfiguration -ComputerName localhost -Wait
```

Démarrage de DSC (mof) sur une machine distante

Démarrer un DSC sur une machine distante est presque aussi simple. En supposant que vous avez déjà configuré la communication à distance Powershell (ou WSMAN activé).

```
$remoteComputer = "myserver.somedomain.com"
$cred = (Get-Credential)
Start-DSCConfiguration -ServerName $remoteComputer -Credential $cred -Verbose
```

Nb: En supposant que vous avez compilé une configuration pour votre noeud sur votre machine locale (et que le fichier myserver.somedomain.com.mof est présent avant de démarrer la configuration)

Importation de psd1 (fichier de données) dans une variable locale

Parfois, il peut être utile de tester vos fichiers de données Powershell et de parcourir les nœuds et les serveurs.

Powershell 5 (WMF5) a ajouté cette petite fonctionnalité intéressante appelée Import-PowerShellDataFile.

Exemple:

```
$data = Import-PowerShellDataFile -path .\MydataFile.psd1
$data.AllNodes
```

Liste des ressources disponibles de DSC

Pour répertorier les ressources DSC disponibles sur votre noeud de création:

```
Get-DscResource
```

Cela répertoriera toutes les ressources pour tous les modules installés (qui se trouvent dans votre PSModulePath) sur votre noeud de création.

Pour répertorier toutes les ressources DSC disponibles dans les sources en ligne (PSGallery ++) sur WMF 5:

```
Find-DSCResource
```

Importation de ressources à utiliser dans DSC

Avant de pouvoir utiliser une ressource dans une configuration, vous devez l'importer explicitement. Le fait de l'avoir installé sur votre ordinateur ne vous permettra pas d'utiliser la ressource implicitement.

Importez une ressource en utilisant `Import-DscResource`.

Exemple montrant comment importer la ressource `PSDesiredStateConfiguration` et la ressource `File`.

```
Configuration InstallPreReqs
{
    param(); # params to DSC goes here.

    Import-DscResource PSDesiredStateConfiguration

    File CheckForTmpFolder {
        Type = 'Directory'
        DestinationPath = 'C:\Tmp'
        Ensure = "Present"
    }
}
```

Remarque : Pour que les ressources DSC fonctionnent, les modules doivent être installés sur les ordinateurs cibles lors de l'exécution de la configuration. Si vous ne les avez pas installés, la configuration échouera.

Lire Configuration d'état souhaitée en ligne:

<https://riptutorial.com/fr/powershell/topic/5662/configuration-d-etat-souhaitee>

Chapitre 16: Conventions de nommage

Exemples

Les fonctions

```
Get-User ()
```

- Utilisez le motif *Verb-Noun* en nommant une fonction.
- Verb implique une action, par exemple `Get` , `Set` , `New` , `Read` , `Write` et bien d'autres. Voir [les verbes approuvés](#) .
- Le nom devrait être singulier même s'il agit sur plusieurs éléments. `Get-User ()` peut renvoyer un ou plusieurs utilisateurs.
- Utilisez le cas Pascal pour le verbe et le nom. Par exemple, `Get-UserLogin ()`

Lire Conventions de nommage en ligne:

<https://riptutorial.com/fr/powershell/topic/9714/conventions-de-nommage>

Chapitre 17: Cordes

Syntaxe

- "(Double-quoted) String"
- 'Chaîne littérale'
- @ "
Chaîne de caractères
"@
- @ '
Littéral ici-chaîne
'@

Remarques

Les chaînes sont des objets représentant du texte.

Exemples

Créer une chaîne de base

Chaîne

Les chaînes sont créées en encapsulant le texte avec des guillemets doubles. Les chaînes entre guillemets peuvent évaluer les variables et les caractères spéciaux.

```
$myString = "Some basic text"  
$mySecondString = "String with a $variable"
```

Pour utiliser un guillemet double à l'intérieur d'une chaîne, il doit être échappé en utilisant le caractère d'échappement backtick (`). Les guillemets simples peuvent être utilisés dans une chaîne entre guillemets.

```
$myString = "A `double quoted` string which also has 'single quotes'."
```

Chaîne littérale

Les chaînes littérales sont des chaînes qui n'évaluent pas les variables et les caractères spéciaux. Il est créé à l'aide de guillemets simples.

```
$myLiteralString = 'Simple text including special characters (`n) and a $variable-reference'
```

Pour utiliser des guillemets simples dans une chaîne littérale, utilisez des guillemets simples ou une chaîne littérale ici. Les quotes doubles peuvent être utilisés en toute sécurité dans une chaîne littérale

```
$myLiteralString = 'Simple string with ''single quotes'' and "double quotes".'
```

Format chaîne

```
$hash = @{ city = 'Berlin' }  
  
$result = 'You should really visit {0}' -f $hash.city  
Write-Host $result #prints "You should really visit Berlin"
```

Les chaînes de format peuvent être utilisées avec l'opérateur `-f` ou la méthode static `[String]::Format(string format, args)` .NET.

Chaîne multiligne

Il existe plusieurs façons de créer une chaîne multiligne dans PowerShell:

- Vous pouvez utiliser les caractères spéciaux pour un retour chariot et / ou une nouvelle ligne manuellement ou utiliser la variable `NewLine` pour insérer la valeur "newline" du système.

```
"Hello`r`nWorld"  
"Hello{0}World" -f [environment]::NewLine
```

- Créer un saut de ligne en définissant une chaîne (avant de fermer le devis)

```
"Hello  
World"
```

- Utiliser une chaîne ici. *C'est la technique la plus courante.*

```
@"  
Hello  
World  
"@
```

Chaîne de caractères

Les chaînes ici sont très utiles lors de la création de chaînes multilignes. Un des principaux avantages par rapport aux autres chaînes multilignes est que vous pouvez utiliser des guillemets sans avoir à les échapper avec un backtick.

Chaîne de caractères

Ici, les chaînes commencent par `@` et un saut de ligne et se terminent par `"@` sur sa propre ligne (`"@` doit être les premiers caractères de la ligne, même pas les espaces / tab).

```
@  
Simple  
    Multiline string  
with "quotes"  
"@
```

Littéral ici-chaîne

Vous pouvez également créer une chaîne de caractères littérale ici en utilisant des guillemets simples, lorsque vous ne souhaitez pas que des expressions soient développées de la même manière qu'une chaîne littérale normale.

```
@'  
The following line won't be expanded  
$(Get-Date)  
because this is a literal here-string  
'@
```

Chaînes concaténantes

Utilisation de variables dans une chaîne

Vous pouvez concaténer des chaînes en utilisant des variables dans une chaîne entre guillemets. Cela ne fonctionne pas avec les propriétés.

```
$string1 = "Power"  
$string2 = "Shell"  
"Greetings from $string1$string2"
```

Utiliser l'opérateur +

Vous pouvez également joindre des chaînes à l'aide de l'opérateur + .

```
$string1 = "Greetings from"  
$string2 = "PowerShell"  
$string1 + " " + $string2
```

Cela fonctionne également avec les propriétés des objets.

```
"The title of this console is '" + $host.Name + '"
```

Utiliser des sous-expressions

Le résultat / résultat d'une sous-expression `$()` peut être utilisé dans une chaîne. Ceci est utile pour accéder aux propriétés d'un objet ou effectuer une expression complexe. Les sous-expressions peuvent contenir plusieurs instructions séparées par un point-virgule ;

```
"Tomorrow is $((Get-Date).AddDays(1).DayOfWeek) "
```

Caractères spéciaux

Utilisé dans une chaîne entre guillemets, le caractère d'échappement (backtick ```) représente un caractère spécial.

```
`0      #Null
`a      #Alert/Beep
`b      #Backspace
`f      #Form feed (used for printer output)
`n      #New line
`r      #Carriage return
`t      #Horizontal tab
`v      #Vertical tab (used for printer output)
```

Exemple:

```
> "This`tuses`ttab`r`nThis is on a second line"
This      uses      tab
This is on a second line
```

Vous pouvez également échapper à des caractères spéciaux ayant des significations spéciales:

```
`#      #Comment-operator
`$      #Variable operator
``      #Escape character
`'      #Single quote
`"      #Double quote
```

Lire Cordes en ligne: <https://riptutorial.com/fr/powershell/topic/5124/cordes>

Chapitre 18: Création de ressources DSC basées sur les classes

Introduction

À partir de PowerShell version 5.0, vous pouvez utiliser les définitions de classe PowerShell pour créer des ressources DSC (Desired State Configuration).

Pour vous aider à créer une ressource DSC, il existe un `[DscResource()]` appliqué à la définition de classe et une `[DscProperty()]` pour désigner les propriétés configurables par l'utilisateur de ressource DSC.

Remarques

Une ressource DSC basée sur une classe doit:

- Être décoré avec l' `[DscResource()]`
- Définir une méthode `Test()` qui retourne `[bool]`
- Définissez une méthode `Get()` qui renvoie son propre type d'objet (par exemple, `[Ticket]`)
- Définir une méthode `Set()` qui renvoie `[void]`
- Au moins une propriété DSC `Key`

Après avoir créé une ressource DSC PowerShell basée sur une classe, celle-ci doit être "exportée" d'un module à l'aide d'un fichier de manifeste de module (.psd1). Dans le manifeste de module, la `DscResourcesToExport` hachage `DscResourcesToExport` est utilisée pour déclarer un tableau de ressources DSC (noms de classe) à "exporter" à partir du module. Cela permet aux consommateurs du module DSC de "voir" les ressources basées sur les classes à l'intérieur du module.

Exemples

Créer une classe de squelette de ressource DSC

```
[DscResource()]  
class File {  
}
```

Cet exemple montre comment créer la section externe d'une classe PowerShell, qui déclare une ressource DSC. Vous devez toujours remplir le contenu de la définition de classe.

Squelette de ressource DSC avec propriété de clé

```
[DscResource()]  
class Ticket {
```

```
[DscProperty(Key)]
[string] $TicketId
}
```

Une ressource DSC doit déclarer au moins une propriété clé. La propriété clé est ce qui identifie de manière unique la ressource des autres ressources. Par exemple, supposons que vous construisiez une ressource DSC qui représente un ticket dans un système de tickets. Chaque ticket serait représenté de manière unique avec un identifiant de ticket.

Chaque propriété qui sera exposée à l' *utilisateur* de la ressource DSC doit être décorée avec l' `[DscProperty()]` . Cet attribut accepte un paramètre `key` pour indiquer que la propriété est un attribut clé de la ressource DSC.

Ressource DSC avec propriété obligatoire

```
[DscResource()]
class Ticket {
    [DscProperty(Key)]
    [string] $TicketId

    [DscProperty(Mandatory)]
    [string] $Subject
}
```

Lors de la création d'une ressource DSC, vous constaterez souvent que chaque propriété ne doit pas nécessairement être obligatoire. Toutefois, vous devez vous assurer que certaines propriétés principales sont configurées par l'utilisateur de la ressource DSC. Vous utilisez le paramètre `Mandatory` de l' `[DscResource()]` pour déclarer une propriété comme requis par l'utilisateur de la ressource DSC.

Dans l'exemple ci-dessus, nous avons ajouté une propriété `Subject` à une ressource `Ticket` , qui représente un ticket unique dans un système de tickets et l'a désignée comme une propriété `Mandatory` .

Ressource DSC avec les méthodes requises

```
[DscResource()]
class Ticket {
    [DscProperty(Key)]
    [string] $TicketId

    # The subject line of the ticket
    [DscProperty(Mandatory)]
    [string] $Subject

    # Get / Set if ticket should be open or closed
    [DscProperty(Mandatory)]
    [string] $TicketState

    [void] Set() {
        # Create or update the resource
    }
}
```

```
[Ticket] Get() {  
    # Return the resource's current state as an object  
    $TicketState = [Ticket]::new()  
    return $TicketState  
}  
  
[bool] Test() {  
    # Return $true if desired state is met  
    # Return $false if desired state is not met  
    return $false  
}  
}
```

Il s'agit d'une ressource DSC complète qui illustre toutes les exigences de base pour créer une ressource valide. Les implémentations de méthode ne sont pas complètes, mais sont fournies dans le but de montrer la structure de base.

Lire [Création de ressources DSC basées sur les classes en ligne](https://riptutorial.com/fr/powershell/topic/8733/creation-de-ressources-dsc-basees-sur-les-classes):

<https://riptutorial.com/fr/powershell/topic/8733/creation-de-ressources-dsc-basees-sur-les-classes>

Chapitre 19: Déclaration de changement

Introduction

Une instruction `switch` permet de tester l'égalité d'une variable par rapport à une liste de valeurs. Chaque valeur est appelée un *cas* et la variable *activée* est vérifiée pour chaque cas de commutation. Cela vous permet d'écrire un script qui peut choisir parmi une série d'options, mais sans vous obliger à écrire une longue série d'instructions `if`.

Remarques

Cette rubrique documente l' *instruction `switch`* utilisée pour brancher le flux du script. Ne le confondez pas avec les *paramètres de commutateur* utilisés dans les fonctions comme indicateurs booléens.

Exemples

Commutateur simple

Les instructions de commutateur comparent une seule valeur de test à plusieurs conditions et effectuent toutes les actions associées pour des comparaisons réussies. Cela peut entraîner plusieurs correspondances / actions.

Compte tenu de l'interrupteur suivant ...

```
switch($myValue)
{
    'First Condition'    { 'First Action' }
    'Second Condition'  { 'Second Action' }
}
```

'First Action' sera `$myValue` si `$myValue` est défini comme 'First Condition' .

'Section Action' sera `$myValue` si `$myValue` est défini comme 'Second Condition' .

Rien ne sera `$myValue` si `$myValue` ne correspond à aucune des deux conditions.

Instruction de changement avec le paramètre `Regex`

Le paramètre `-Regex` permet aux instructions `switch` d'effectuer une correspondance d'expressions régulières par rapport aux conditions.

Exemple:

```
switch -Regex ('Condition')
{
```

```
'Con\D+ion'    {'One or more non-digits'}
'Conditio*$'  {'Zero or more "o"'}
'C.ndition'   {'Any single char.'}
'^C\w+ition$' {'Anchors and one or more word chars.'}
'Test'        {'No match'}
}
```

Sortie:

```
One or more non-digits
Any single char.
Anchors and one or more word chars.
```

Commutation simple avec pause

Le mot-clé `break` peut être utilisé dans les instructions `switch` pour quitter l'instruction avant d'évaluer toutes les conditions.

Exemple:

```
switch('Condition')
{
  'Condition'
  {
    'First Action'
  }
  'Condition'
  {
    'Second Action'
    break
  }
  'Condition'
  {
    'Third Action'
  }
}
```

Sortie:

```
First Action
Second Action
```

En raison du mot-clé `break` dans la seconde action, la troisième condition n'est pas évaluée.

Instruction de changement avec un paramètre générique

Le paramètre `-Wildcard` permet aux instructions de commutateur d'effectuer une correspondance avec les caractères génériques.

Exemple:

```
switch -Wildcard ('Condition')
```

```

{
  'Condition'           {'Normal match'}
  'Condit*'            {'Zero or more wildcard chars.'}
  'C[aoc]ndit[f-l]on'  {'Range and set of chars.'}
  'C?ndition'          {'Single char. wildcard'}
  'Test*'              {'No match'}
}

```

Sortie:

```

Normal match
Zero or more wildcard chars.
Range and set of chars.
Single char. wildcard

```

Instruction de changement avec paramètre exact

Le paramètre `-Exact` applique les instructions de commutateur pour effectuer une correspondance exacte, insensible à la casse, par rapport aux conditions de chaîne.

Exemple:

```

switch -Exact ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'  {'Third Action'}
  '^*ondition$' {'Fourth Action'}
  'Conditio*'  {'Fifth Action'}
}

```

Sortie:

```

First Action
Second Action
Third Action

```

Les première à troisième actions sont exécutées car leurs conditions associées correspondent à l'entrée. Les chaînes regex et wildcard dans les quatrième et cinquième conditions échouent à la correspondance.

Notez que la quatrième condition correspondrait également à la chaîne d'entrée si la correspondance des expressions régulières était en cours, mais a été ignorée dans ce cas, car ce n'est pas le cas.

Instruction de changement avec le paramètre CaseSensitive

Le paramètre `-CaseSensitive` applique les instructions de commutateur pour effectuer une correspondance exacte et sensible à la casse par rapport aux conditions.

Exemple:

```
switch -CaseSensitive ('Condition')
{
  'condition'   {'First Action'}
  'Condition'   {'Second Action'}
  'conditioN'  {'Third Action'}
}
```

Sortie:

```
Second Action
```

La seconde action est la seule action exécutée car c'est la seule condition qui correspond exactement à la chaîne 'Condition' lors de la prise en compte de la sensibilité à la casse.

Instruction de changement avec paramètre de fichier

Le paramètre `-file` permet à l'instruction `switch` de recevoir les entrées d'un fichier. Chaque ligne du fichier est évaluée par l'instruction `switch`.

Exemple de fichier `input.txt` :

```
condition
test
```

Exemple de déclaration de commutation:

```
switch -file input.txt
{
  'condition' {'First Action'}
  'test'      {'Second Action'}
  'fail'      {'Third Action'}
}
```

Sortie:

```
First Action
Second Action
```

Commutateur simple avec condition par défaut

Le mot-clé `Default` est utilisé pour exécuter une action lorsqu'aucune autre condition ne correspond à la valeur d'entrée.

Exemple:

```
switch('Condition')
{
  'Skip Condition'
  {
    'First Action'
  }
}
```

```
'Skip This Condition Too'
{
  'Second Action'
}
Default
{
  'Default Action'
}
}
```

Sortie:

```
Default Action
```

Changer de déclaration avec des expressions

Les conditions peuvent aussi être des expressions:

```
$myInput = 0

switch($myInput) {
  # because the result of the expression, 4,
  # does not equal our input this block should not be run.
  (2+2) { 'True. 2 +2 = 4' }

  # because the result of the expression, 0,
  # does equal our input this block should be run.
  (2-2) { 'True. 2-2 = 0' }

  # because our input is greater than -1 and is less than 1
  # the expression evaluates to true and the block should be run.
  { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }
}

#Output
True. 2-2 = 0
True. Value is 0
```

Lire Déclaration de changement en ligne:

<https://riptutorial.com/fr/powershell/topic/1174/declaration-de-changement>

Chapitre 20: Envoi d'email

Introduction

Une technique utile pour les administrateurs Exchange Server consiste à pouvoir envoyer des messages électroniques via SMTP à partir de PowerShell. Selon la version de PowerShell installée sur votre ordinateur ou votre serveur, il existe plusieurs façons d'envoyer des e-mails via powershell. Il existe une option d'applet de commande native simple et facile à utiliser. Il utilise l'applet de commande **Send-MailMessage**.

Paramètres

Paramètre	Détails
Pièces jointes <String []>	Chemin et noms de fichiers des fichiers à joindre au message. Les chemins et les noms de fichiers peuvent être acheminés vers Send-MailMessage.
Cci <String []>	Les adresses e-mail qui reçoivent une copie d'un e-mail mais n'apparaissent pas en tant que destinataire dans le message. Entrez des noms (facultatif) et l'adresse e-mail (obligatoire), par exemple Nom de personne@exemple.com ou quelqu'un@exemple.com.
Corps <String_>	Contenu du message électronique.
BodyAsHtml	Il indique que le contenu est au format HTML.
Cc <String []>	Les adresses électroniques qui reçoivent une copie d'un message électronique. Entrez des noms (facultatif) et l'adresse e-mail (obligatoire), par exemple Nom de personne@exemple.com ou quelqu'un@exemple.com.
Credential	Spécifie un compte d'utilisateur autorisé à envoyer un message à partir d'une adresse électronique spécifiée. La valeur par défaut est l'utilisateur actuel. Entrez un nom tel que User ou Domain \ User, ou entrez un objet PSCredential.
LivraisonNotificationOption	Spécifie les options de notification de remise pour le message électronique. Plusieurs valeurs peuvent être spécifiées. Les notifications de livraison sont envoyées dans le message à l'adresse spécifiée dans le paramètre À. Valeurs acceptables: Aucune, OnSuccess, OnFailure, Delay, Never.
Codage	Encodage pour le corps et le sujet. Valeurs acceptables: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Default,

Paramètre	Détails
	OEM.
De	Les adresses e-mail à partir desquelles le courrier est envoyé. Entrez des noms (facultatif) et l'adresse e-mail (requis), tels que Nom de personne@exemple.com ou quelqu'un@exemple.com.
Port	Port alternatif sur le serveur SMTP. La valeur par défaut est 25. Disponible à partir de Windows PowerShell 3.0.
Priorité	Priorité du message électronique. Valeurs acceptables: Normal, Haut, Bas.
SmtpServer	Nom du serveur SMTP qui envoie le message électronique. La valeur par défaut est la valeur de la variable \$ PSEmailServer.
Assujettir	Objet du message électronique.
À	Les adresses e-mail auxquelles le courrier est envoyé. Entrez des noms (facultatif) et l'adresse e-mail (obligatoire), par exemple, nom de quelqu'un@exemple.com ou quelqu'un@exemple.com
UseSsl	Utilise le protocole SSL (Secure Sockets Layer) pour établir une connexion avec l'ordinateur distant pour envoyer du courrier

Exemples

Message d'envoi simple

```
Send-MailMessage -From sender@bar.com -Subject "Email Subject" -To receiver@bar.com -
SmtpServer smtp.com
```

Send-MailMessage avec des paramètres prédéfinis

```
$parameters = @{
    From = 'from@bar.com'
    To = 'to@bar.com'
    Subject = 'Email Subject'
    Attachments = @( 'C:\files\samplefile1.txt', 'C:\files\samplefile2.txt' )
    BCC = 'bcc@bar.com'
    Body = 'Email body'
    BodyAsHTML = $False
    CC = 'cc@bar.com'
    Credential = Get-Credential
    DeliveryNotificationOption = 'onSuccess'
    Encoding = 'UTF8'
    Port = '25'
    Priority = 'High'
}
```

```
SmtptServer = 'smtp.com'
UseSSL = $True
}

# Notice: Splatting requires @ instead of $ in front of variable name
Send-MailMessage @parameters
```

SMTPClient - Mail avec un fichier .txt dans le message du corps

```
# Define the txt which will be in the email body
$txt_File = "c:\file.txt"

function Send_mail {
    #Define Email settings
    $EmailFrom = "source@domain.com"
    $EmailTo = "destination@domain.com"
    $Txt_Body = Get-Content $Txt_File -RAW
    $Body = $Body_Custom + $Txt_Body
    $Subject = "Email Subject"
    $SMTPServer = "smtpserver.domain.com"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtptServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)
}

$Body_Custom = "This is what contain file.txt : "

Send_mail
```

Lire Envoi d'email en ligne: <https://riptutorial.com/fr/powershell/topic/2040/envoi-d-email>

Chapitre 21: Executables en cours d'exécution

Exemples

Applications de console

```
PS> console_app.exe
PS> & console_app.exe
PS> Start-Process console_app.exe
```

Applications GUI

```
PS> gui_app.exe (1)
PS> & gui_app.exe (2)
PS> & gui_app.exe | Out-Null (3)
PS> Start-Process gui_app.exe (4)
PS> Start-Process gui_app.exe -Wait (5)
```

Les applications d'interface graphique se lancent dans un processus différent et renverront immédiatement le contrôle à l'hôte PowerShell. Parfois, vous avez besoin de l'application pour terminer le traitement avant que la prochaine instruction PowerShell ne soit exécutée. Ceci peut être réalisé en redirigeant la sortie de l'application vers \$ null (3) ou en utilisant le processus de démarrage avec le commutateur -Wait (5).

Flux de console

```
PS> $ErrorActionPreference = "Continue" (1)
PS> & console_app.exe *>&1 | % { $_ } (2)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.ErrorRecord] } (3)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.WarningRecord] } (4)
PS> & console_app.exe *>&1 | ? { $_ -is [System.Management.Automation.VerboseRecord] } (5)
PS> & console_app.exe *>&1 (6)
PS> & console_app.exe 2>&1 (7)
```

Le flux 2 contient des objets System.Management.Automation.ErrorRecord. Notez que certaines applications comme git.exe utilisent le "flux d'erreur" à des fins d'information, qui ne sont pas nécessairement des erreurs. Dans ce cas, il est préférable de regarder le code de sortie pour déterminer si le flux d'erreur doit être interprété comme une erreur.

PowerShell comprend ces flux: Output, Error, Warning, Verbose, Debug, Progress. Les applications natives utilisent généralement uniquement ces flux: Output, Error, Warning.

Dans PowerShell 5, tous les flux peuvent être redirigés vers le flux de sortie / succès standard (6).

Dans les versions antérieures de PowerShell, seuls des flux spécifiques pouvaient être redirigés

vers le flux de sortie / succès standard (7). Dans cet exemple, le "flux d'erreur" sera redirigé vers le flux de sortie.

Codes de sortie

```
PS> $LastExitCode  
PS> $?  
PS> $Error[0]
```

Ce sont des variables PowerShell intégrées qui fournissent des informations supplémentaires sur l'erreur la plus récente. `$LastExitCode` est le code de sortie final de la dernière application native exécutée. `$?` et `$Error[0]` est le dernier enregistrement d'erreur généré par PowerShell.

Lire Executables en cours d'exécution en ligne:

<https://riptutorial.com/fr/powershell/topic/7707/executables-en-cours-d-execution>

Chapitre 22: Expressions régulières

Syntaxe

- 'texte' -match 'RegexPattern'
- 'text' -replace 'RegexPattern', 'newvalue'
- [regex] :: Match ("text", "pattern") # Match unique
- [regex] :: Matches ("text", "pattern") # Correspondances multiples
- [regex] :: Remplacer ("text", "pattern", "newvalue")
- [regex] :: Remplacer ("text", "pattern", {param (\$ m)}) #MatchEvaluator
- [regex] :: Escape ("input") #Escape des caractères spéciaux

Exemples

Match unique

Vous pouvez rapidement déterminer si un texte inclut un motif spécifique à l'aide de Regex. Il existe plusieurs façons de travailler avec Regex dans PowerShell.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Utilisation de l'opérateur -Match

Pour déterminer si une chaîne correspond à un modèle à l'aide de l'opérateur `-matches`, utilisez la syntaxe `'input' -match 'pattern'`. Cela retournera `true` ou `false` selon le résultat de la recherche. S'il y avait une correspondance, vous pouvez afficher la correspondance et les groupes (si définis dans le modèle) en accédant à la variable `$Matches`.

```
> $text -match $pattern
True

> $Matches

Name Value
----
0     (a)
```

Vous pouvez également utiliser `-match` pour filtrer un tableau de chaînes et renvoyer uniquement

les chaînes contenant une correspondance.

```
> $textarray = @"
This is (a) sample
text, this is
a (sample text)
"@ -split "`n"

> $textarray -match $pattern
This is (a) sample
a (sample text)
```

2.0

Utilisation de Select-String

PowerShell 2.0 a introduit une nouvelle applet de commande pour parcourir le texte en utilisant regex. Il renvoie un objet `MatchInfo` par entrée de texte contenant une correspondance. Vous pouvez accéder à ses propriétés pour trouver des groupes correspondants, etc.

```
> $m = Select-String -InputObject $text -Pattern $pattern

> $m

This is (a) sample
text, this is
a (sample text)

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
            text, this is
            a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*\?)
Context    :
Matches    : {(a)}
```

Comme `-match`, `Select-String` peut également être utilisé pour filtrer un tableau de chaînes en y insérant un tableau. Il crée un `MatchInfo` `MatchInfo` par chaîne comprenant une correspondance.

```
> $textarray | Select-String -Pattern $pattern

This is (a) sample
a (sample text)

#You can also access the matches, groups etc.
> $textarray | Select-String -Pattern $pattern | fl *

IgnoreCase : True
LineNumber : 1
```

```
Line      : This is (a) sample
Filename  : InputStream
Path      : InputStream
Pattern   : \(.*?\)
Context   :
Matches   : {(a)}
```

```
IgnoreCase : True
LineNumber  : 3
Line       : a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(sample text)}
```

`Select-String` peut également rechercher en utilisant un modèle de texte normal (pas de regex) en ajoutant le commutateur `-SimpleMatch`.

Utilisation de [Regex] :: Match ()

Vous pouvez également utiliser la méthode statique `Match()` disponible dans la classe `.NET [Regex]`.

```
> [regex]::Match($text,$pattern)

Groups    : {(a)}
Success   : True
Captures : {(a)}
Index     : 8
Length    : 3
Value     : (a)

> [regex]::Match($text,$pattern) | Select-Object -ExpandProperty Value
(a)
```

Remplacer

Une tâche courante pour regex consiste à remplacer le texte qui correspond à un modèle par une nouvelle valeur.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Text wrapped in ()
$pattern = \(.*?\)

#Replace matches with:
$newvalue = 'test'
```

Utilisation de l'opérateur -Replace

L'opérateur `-replace` dans PowerShell peut être utilisé pour remplacer le texte correspondant à un motif par une nouvelle valeur en utilisant la syntaxe `'input' -replace 'pattern', 'newvalue'` .

```
> $text -replace $pattern, $newvalue
This is test sample
text, this is
a test
```

En utilisant la méthode [Regex] :: Replace ()

Le remplacement de correspondances peut également être effectué à l'aide de la méthode `Replace()` dans la classe `[Regex]` .NET.

```
[regex]::Replace($text, $pattern, 'test')
This is test sample
text, this is
a test
```

Remplacer le texte par une valeur dynamique par un MatchEvaluator

Parfois, vous devez remplacer une valeur correspondant à un modèle par une nouvelle valeur basée sur cette correspondance spécifique, ce qui rend impossible la prédiction de la nouvelle valeur. Pour ces types de scénarios, un `MatchEvaluator` peut être très utile.

Dans PowerShell, un `MatchEvaluator` est aussi simple qu'un bloc de script avec un seul paramètre contenant un objet `Match` pour la correspondance en cours. La sortie de l'action sera la nouvelle valeur pour cette correspondance spécifique. `MatchEvaluator` peut être utilisé avec la méthode statique `[Regex]::Replace()` .

Exemple : Remplacement du texte à l'intérieur `()` par sa longueur

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '(?<=\()\.*(?=\))'

$MatchEvaluator = {
    param($match)

    #Replace content with length of content
    $match.Value.Length
}
```

```
}
```

Sortie:

```
> [regex]::Replace($text, $pattern, $MatchEvaluator)

This is 1 sample
text, this is
a 11
```

Exemple: Créer un `sample` majuscule

```
#Sample pattern: "Sample"
$pattern = 'sample'

$MatchEvaluator = {
    param($match)

    #Return match in upper-case
    $match.Value.ToUpper()
}
```

Sortie:

```
> [regex]::Replace($text, $pattern, $MatchEvaluator)

This is (a) SAMPLE
text, this is
a (SAMPLE text)
```

Échapper des caractères spéciaux

Un motif regex utilise de nombreux caractères spéciaux pour décrire un motif. Ex., `.` signifie "tout caractère", `+` est "un ou plusieurs" etc.

Pour utiliser ces caractères, sous forme de `.`, `+` Etc., dans un modèle, vous devez les échapper pour enlever leur signification particulière. Ceci est fait en utilisant le caractère d'échappement qui est une barre oblique inverse `\` dans l'expression régulière. Exemple: Pour rechercher `+`, vous utiliseriez le pattern `\+`.

Il peut être difficile de se souvenir de tous les caractères spéciaux dans regex. Pour échapper à tous les caractères spéciaux d'une chaîne que vous souhaitez rechercher, vous pouvez utiliser la méthode `[Regex]::Escape("input")`.

```
> [regex]::Escape("(foo)")
\ (foo\ )

> [regex]::Escape("1+1.2=2.2")
1\+1\.2=2\.2
```

Plusieurs correspondances

Il existe plusieurs façons de rechercher toutes les correspondances pour un motif dans un texte.

```
#Sample text
$text = @"
This is (a) sample
text, this is
a (sample text)
"@

#Sample pattern: Content wrapped in ()
$pattern = '\(.*?\)'
```

Utilisation de Select-String

Vous pouvez trouver toutes les correspondances (correspondance globale) en ajoutant le commutateur `-AllMatches` à `Select-String`.

```
> $m = Select-String -InputObject $text -Pattern $pattern -AllMatches

> $m | Format-List *

IgnoreCase : True
LineNumber : 1
Line       : This is (a) sample
           : text, this is
           : a (sample text)
Filename   : InputStream
Path       : InputStream
Pattern    : \(.*?\)
Context    :
Matches    : {(a), (sample text)}

#List all matches
> $m.Matches

Groups     : {(a)}
Success    : True
Captures  : {(a)}
Index      : 8
Length     : 3
Value      : (a)

Groups     : {(sample text)}
Success    : True
Captures  : {(sample text)}
Index      : 37
Length     : 13
Value      : (sample text)

#Get matched text
> $m.Matches | Select-Object -ExpandProperty Value
(a)
(sample text)
```

Utiliser [Regex] :: Matches ()

La méthode `Matches()` de la classe .NET `[regex]` peut également être utilisée pour effectuer une recherche globale de plusieurs correspondances.

```
> [regex]::Matches($text,$pattern)

Groups      : {(a)}
Success     : True
Captures   : {(a)}
Index       : 8
Length      : 3
Value       : (a)

Groups      : {(sample text)}
Success     : True
Captures   : {(sample text)}
Index       : 37
Length      : 13
Value       : (sample text)

> [regex]::Matches($text,$pattern) | Select-Object -ExpandProperty Value

(a)
(sample text)
```

Lire Expressions régulières en ligne: <https://riptutorial.com/fr/powershell/topic/6674/expressions-regulieres>

Chapitre 23: Flux de travail PowerShell

Introduction

PowerShell Workflow est une fonctionnalité qui a été introduite à partir de PowerShell version 3.0. Les définitions de flux de travail ressemblent beaucoup aux définitions de fonction PowerShell, mais elles s'exécutent dans l'environnement Windows Workflow Foundation, et non directement dans le moteur PowerShell.

Le moteur de workflow comprend plusieurs fonctionnalités uniques, telles que la persistance des tâches.

Remarques

La fonctionnalité de workflow PowerShell est exclusivement prise en charge sur la plate-forme Microsoft Windows, sous PowerShell Desktop Edition. PowerShell Core Edition, qui est pris en charge sous Linux, Mac et Windows, ne prend pas en charge la fonctionnalité PowerShell Workflow.

Lors de la création d'un workflow PowerShell, gardez à l'esprit que les workflows appellent des activités, pas des applets de commande. Vous pouvez toujours appeler des applets de commande à partir d'un workflow PowerShell, mais le moteur de workflow `InlineScript` implicitement l'appel de la cmdlet dans une activité `InlineScript`. Vous pouvez également explicitement envelopper le code à l'intérieur de l'activité `InlineScript`, qui exécute le code PowerShell; Par défaut, l'activité `InlineScript` s'exécute dans un processus distinct et renvoie le résultat au workflow appelant.

Exemples

Exemple de workflow simple

```
workflow DoSomeWork {
    Get-Process -Name notepad | Stop-Process
}
```

Ceci est un exemple de base d'une définition de workflow PowerShell.

Workflow avec des paramètres d'entrée

Tout comme les fonctions PowerShell, les flux de travail peuvent accepter des paramètres d'entrée. Les paramètres d'entrée peuvent éventuellement être liés à un type de données spécifique, tel qu'une chaîne, un entier, etc. Utilisez le mot-clé `param` standard pour définir un bloc de paramètres d'entrée, directement après la déclaration de workflow.

```
workflow DoSomeWork {
    param (
```

```
[string[]] $ComputerName
)
Get-Process -ComputerName $ComputerName
}

DoSomeWork -ComputerName server01, server02, server03
```

Exécuter le workflow en tant que job d'arrière-plan

Les workflows PowerShell sont intrinsèquement équipés de la possibilité de s'exécuter en tâche de fond. Pour appeler un workflow en tant que job d'arrière-plan PowerShell, utilisez le paramètre `-AsJob` lors de l'appel du workflow.

```
workflow DoSomeWork {
    Get-Process -ComputerName server01
    Get-Process -ComputerName server02
    Get-Process -ComputerName server03
}

DoSomeWork -AsJob
```

Ajouter un bloc parallèle à un workflow

```
workflow DoSomeWork {
    parallel {
        Get-Process -ComputerName server01
        Get-Process -ComputerName server02
        Get-Process -ComputerName server03
    }
}
```

L'une des fonctionnalités uniques de PowerShell Workflow est la possibilité de définir un bloc d'activités en parallèle. Pour utiliser cette fonctionnalité, utilisez le mot-clé `parallel` dans votre workflow.

L'appel des activités de workflow en parallèle peut aider à améliorer les performances de votre flux de travail.

Lire Flux de travail PowerShell en ligne: <https://riptutorial.com/fr/powershell/topic/8745/flux-de-travail-powershell>

Chapitre 24: Fonctions PowerShell

Introduction

Une fonction est essentiellement un bloc de code nommé. Lorsque vous appelez le nom de la fonction, le bloc de script de cette fonction s'exécute. C'est une liste d'instructions PowerShell qui porte un nom que vous attribuez. Lorsque vous exécutez une fonction, vous tapez le nom de la fonction. C'est une méthode permettant de gagner du temps lors de tâches répétitives.

PowerShell formate en trois parties: le mot-clé 'Fonction', suivi d'un nom, enfin, la charge utile contenant le bloc de script, entre crochets / parenthèses.

Exemples

Fonction simple sans paramètre

Ceci est un exemple de fonction qui renvoie une chaîne. Dans l'exemple, la fonction est appelée dans une instruction affectant une valeur à une variable. La valeur dans ce cas est la valeur de retour de la fonction.

```
function Get-Greeting{
    "Hello World"
}

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` déclare que le code suivant est une fonction.

`Get-Greeting` est le nom de la fonction. Chaque fois que cette fonction doit être utilisée dans le script, la fonction peut être appelée en l'appelant par son nom.

`{ ... }` est le bloc de script exécuté par la fonction.

Si le code ci-dessus est exécuté dans l'ISE, les résultats seraient les suivants:

```
Hello World
Hello World
```

Paramètres de base

Une fonction peut être définie avec des paramètres en utilisant le bloc `param`:

```
function Write-Greeting {
```

```
param(
    [Parameter(Mandatory,Position=0)]
    [String]$name,
    [Parameter(Mandatory,Position=1)]
    [Int]$age
)
"Hello $name, you are $age years old."
}
```

Ou en utilisant la syntaxe de fonction simple:

```
function Write-Greeting ($name, $age) {
    "Hello $name, you are $age years old."
}
```

Remarque: les paramètres de moulage ne sont requis dans aucun type de définition de paramètre.

La syntaxe de fonction simple (SFS) a des capacités très limitées par rapport au bloc param. Bien que vous puissiez définir des paramètres à exposer dans la fonction, vous ne pouvez pas spécifier d' [attributs de paramètre](#) , utiliser la [validation de paramètre](#) , inclure `[CmdletBinding()]` , avec SFS (et cette liste n'est pas exhaustive).

Les fonctions peuvent être appelées avec des paramètres ordonnés ou nommés.

L'ordre des paramètres sur l'invocation correspond à l'ordre de la déclaration dans l'en-tête de fonction (par défaut) ou peut être spécifié à l'aide de l'attribut de paramètre de `Position` (comme illustré dans l'exemple de fonction avancée ci-dessus).

```
$greeting = Write-Greeting "Jim" 82
```

Alternativement, cette fonction peut être appelée avec des paramètres nommés

```
$greeting = Write-Greeting -name "Bob" -age 82
```

Paramètres obligatoires

Les paramètres d'une fonction peuvent être marqués comme obligatoires

```
function Get-Greeting{
    param
    (
        [Parameter(Mandatory=$true)]$name
    )
    "Hello World $name"
}
```

Si la fonction est appelée sans valeur, la ligne de commande demandera la valeur:

```
$greeting = Get-Greeting
```

```
cmdlet Get-Greeting at command pipeline position 1
Supply values for the following parameters:
name:
```

Fonction avancée

Ceci est une copie de l'extrait de fonctions avancées de Powershell ISE. Fondamentalement, il s'agit d'un modèle pour un grand nombre de choses que vous pouvez utiliser avec les fonctions avancées de Powershell. Points clés à noter:

- Intégration de get-help - le début de la fonction contient un bloc de commentaires configuré pour être lu par l'applet de commande get-help. Le bloc fonction peut être situé à la fin, si vous le souhaitez.
- cmdletbinding - la fonction se comportera comme une applet de commande
- paramètres
- jeux de paramètres

```
<#
.Synopsis
    Short description
.DESRIPTION
    Long description
.EXAMPLE
    Example of how to use this cmdlet
.EXAMPLE
    Another example of how to use this cmdlet
.INPUTS
    Inputs to this cmdlet (if any)
.OUTPUTS
    Output from this cmdlet (if any)
.NOTES
    General notes
.COMPONENT
    The component this cmdlet belongs to
.ROLE
    The role this cmdlet belongs to
.FUNCTIONALITY
    The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
        SupportsShouldProcess=$true,
        PositionalBinding=$false,
        HelpUri = 'http://www.microsoft.com/',
        ConfirmImpact='Medium')]

    [Alias()]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true,
            ValueFromRemainingArguments=$false,
            Position=0,
```

```

        ParameterSetName='Parameter Set 1'])
[ValidateNotNull()]
[ValidateNotNullOrEmpty()]
[ValidateCount(0,5)]
[ValidateSet("sun", "moon", "earth")]
[Alias("p1")]
$Param1,

# Param2 help description
[Parameter(ParameterSetName='Parameter Set 1')]
[AllowNull()]
[AllowEmptyCollection()]
[AllowEmptyString()]
[ValidateScript({$true})]
[ValidateRange(0,5)]
[int]
$Param2,

# Param3 help description
[Parameter(ParameterSetName='Another Parameter Set')]
[ValidatePattern("[a-z]*")]
[ValidateLength(0,15)]
[String]
$Param3
)

Begin
{
}
Process
{
    if ($pscmdlet.ShouldProcess("Target", "Operation"))
    {
    }
}
End
{
}
}

```

Validation des paramètres

Il existe plusieurs façons de valider la saisie de paramètres dans PowerShell.

Au lieu d'écrire du code dans les fonctions ou les scripts pour valider les valeurs des paramètres, ces paramètres lanceront si des valeurs non valides sont transmises.

ValidateSet

Parfois, nous devons limiter les valeurs possibles qu'un paramètre peut accepter. Disons que nous voulons autoriser uniquement le rouge, le vert et le bleu pour le paramètre `$Color` dans un script ou une fonction.

Nous pouvons utiliser l'attribut de paramètre `ValidateSet` pour limiter cela. Il présente l'avantage supplémentaire de permettre la complétion de tabulation lors de la définition de cet argument (dans certains environnements).

```
param(  
    [ValidateSet('red','green','blue',IgnoreCase)]  
    [string]$Color  
)
```

Vous pouvez également spécifier `IgnoreCase` pour désactiver la sensibilité à la casse.

ValiderRange

Cette méthode de validation des paramètres prend une valeur min et max `Int32` et nécessite que le paramètre soit compris dans cette plage.

```
param(  
    [ValidateRange(0,120)]  
    [Int]$Age  
)
```

ValidatePattern

Cette méthode de validation des paramètres accepte les paramètres correspondant au modèle de regex spécifié.

```
param(  
    [ValidatePattern("\w{4-6}\d{2}")]  
    [string]$UserName  
)
```

ValidateLength

Cette méthode de validation des paramètres teste la longueur de la chaîne transmise.

```
param(  
    [ValidateLength(0,15)]  
    [String]$PhoneNumber  
)
```

ValidateCount

Cette méthode de validation des paramètres teste la quantité d'arguments transmis, par exemple, un tableau de chaînes.

```
param(  
    [ValidateCount(1,5)]  
    [String[]]$ComputerName  
)
```

ValidateScript

Enfin, la méthode `ValidateScript` est extraordinairement flexible, prenant un scriptblock et l'évaluant en utilisant `$ _` pour représenter l'argument passé. Il passe ensuite l'argument si le résultat est `$ true` (y compris toute sortie valide).

Cela peut être utilisé pour tester qu'un fichier existe:

```
param(
    [ValidateScript({Test-Path $_})]
    [IO.FileInfo]$Path
)
```

Pour vérifier qu'un utilisateur existe dans AD:

```
param(
    [ValidateScript({Get-ADUser $_})]
    [String]$UserName
)
```

Et à peu près tout ce que vous pouvez écrire (car cela ne se limite pas aux oneliners):

```
param(
    [ValidateScript({
        $AnHourAgo = (Get-Date).AddHours(-1)
        if ($_ -lt $AnHourAgo.AddMinutes(5) -and $_ -gt $AnHourAgo.AddMinutes(-5)) {
            $true
        } else {
            throw "That's not within five minutes. Try again."
        }
    })]
    [String]$TimeAboutAnHourAgo
)
```

Lire Fonctions PowerShell en ligne: <https://riptutorial.com/fr/powershell/topic/1673/fonctions-powershell>

Chapitre 25: Gestion des paquets

Introduction

PowerShell Package Management vous permet de rechercher, d'installer, de mettre à jour et de désinstaller des modules PowerShell et d'autres packages.

PowerShellGallery.com est la source par défaut pour les modules PowerShell. Vous pouvez également parcourir le site pour rechercher les packages disponibles, commander et prévisualiser le code.

Exemples

Recherchez un module PowerShell à l'aide d'un motif

Trouver un module qui se termine par `DSC`

```
Find-Module -Name *DSC
```

Créez la tolérance de module PowerShell par défaut

Si pour une raison quelconque, le dépôt du module PowerShell par défaut `PSGallery` se retire. Vous devrez le créer. Ceci est la commande.

```
Register-PSRepository -Default
```

Trouver un module par nom

```
Find-Module -Name <Name>
```

Installer un module par nom

```
Install-Module -Name <name>
```

Désinstaller un module mon nom et sa version

```
Uninstall-Module -Name <Name> -RequiredVersion <Version>
```

Mettre à jour un module par nom

```
Update-Module -Name <Name>
```

Lire Gestion des paquets en ligne: <https://riptutorial.com/fr/powershell/topic/8698/gestion-des-paquets>

Chapitre 26: Gestion des secrets et des informations d'identification

Introduction

Dans Powershell, pour éviter de stocker le mot de passe en *texte clair*, nous utilisons différentes méthodes de chiffrement et le stockons en tant que chaîne sécurisée. Lorsque vous ne spécifiez pas de clé ou de clé sécurisée, cela ne fonctionnera que si le même utilisateur sur le même ordinateur pourra déchiffrer la chaîne chiffrée si vous n'utilisez pas Keys / SecureKeys. Tout processus exécuté sous ce même compte d'utilisateur pourra déchiffrer cette chaîne chiffrée sur le même ordinateur.

Exemples

Demander des informations d'identification

Pour demander des informations d'identification, vous devez presque toujours utiliser l'applet de commande `Get-Credential` :

```
$credential = Get-Credential
```

Nom d'utilisateur pré-rempli:

```
$credential = Get-Credential -UserName 'myUser'
```

Ajoutez un message d'invite personnalisé:

```
$credential = Get-Credential -Message 'Please enter your company email address and password.'
```

Accéder au mot de passe en texte clair

Le mot de passe dans un objet de référence est un `[SecureString]` . Le moyen le plus simple est d'obtenir un `[NetworkCredential]` qui ne stocke pas le mot de passe crypté:

```
$credential = Get-Credential  
$plainPass = $credential.GetNetworkCredential().Password
```

La méthode d'assistance (`.GetNetworkCredential()`) n'existe que sur les objets `[PSCredential]` . Pour gérer directement un `[SecureString]` , utilisez les méthodes .NET:

```
$bstr = [System.Runtime.InteropServices.Marshal]::SecureStringToBSTR($secStr)  
$plainPass = [System.Runtime.InteropServices.Marshal]::PtrToStringAuto($bstr)
```

Travailler avec des informations d'identification stockées

Pour stocker et récupérer facilement les informations d'identification chiffrées, utilisez la sérialisation XML intégrée de PowerShell (Clixml):

```
$credential = Get-Credential  
  
$credential | Export-CliXml -Path 'C:\My\Path\cred.xml'
```

Pour réimporter:

```
$credential = Import-CliXml -Path 'C:\My\Path\cred.xml'
```

La chose importante à retenir est que, par défaut, il utilise l'API de protection des données Windows et que la clé utilisée pour chiffrer le mot de passe est spécifique à l' *utilisateur et à la machine* sur laquelle le code est exécuté.

Par conséquent, les informations d'identification cryptées ne peuvent pas être importées par un utilisateur différent ni par le même utilisateur sur un autre ordinateur.

En cryptant plusieurs versions du même identifiant avec différents utilisateurs en cours d'exécution et sur différents ordinateurs, vous pouvez disposer du même secret pour plusieurs utilisateurs.

En plaçant le nom de l'utilisateur et de l'ordinateur dans le nom du fichier, vous pouvez stocker tous les secrets cryptés de manière à ce que le même code puisse les utiliser sans coder en dur:

Encrypter

```
# run as each user, and on each computer  
  
$credential = Get-Credential  
  
$credential | Export-CliXml -Path  
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

Le code qui utilise les informations d'identification stockées:

```
$credential = Import-CliXml -Path  
"C:\My\Secrets\myCred_${env:USERNAME}_${env:COMPUTERNAME}.xml"
```

La version correcte du fichier pour l'utilisateur en cours d'exécution sera chargée automatiquement (ou échouera car le fichier n'existe pas).

Stockage des informations d'identification sous forme chiffrée et transmission en tant que paramètre lorsque requis

```
$username = "user1@domain.com"
$pwdTxt = Get-Content "C:\temp\Stored_Password.txt"
$securePwd = $pwdTxt | ConvertTo-SecureString
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd
# Now, $credObject is having the credentials stored and you can pass it wherever you want.

## Import Password with AES

$username = "user1@domain.com"
$AESKey = Get-Content $AESKeyFilePath
$pwdTxt = Get-Content $SecurePwdFilePath
$securePwd = $pwdTxt | ConvertTo-SecureString -Key $AESKey
$credObject = New-Object System.Management.Automation.PSCredential -ArgumentList $username,
$securePwd

# Now, $credObject is having the credentials stored with AES Key and you can pass it wherever
you want.
```

Lire [Gestion des secrets et des informations d'identification en ligne](https://riptutorial.com/fr/powershell/topic/2917/gestion-des-secrets-et-des-informations-d-identification):

<https://riptutorial.com/fr/powershell/topic/2917/gestion-des-secrets-et-des-informations-d-identification>

Chapitre 27: GUI dans Powershell

Exemples

Interface utilisateur WPF pour applet de commande Get-Service

```
Add-Type -AssemblyName PresentationFramework

[xml]$XAMLWindow = '
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="Auto"
  SizeToContent="WidthAndHeight"
  Title="Get-Service">
  <ScrollViewer Padding="10,10,10,0" ScrollViewer.VerticalScrollBarVisibility="Disabled">
    <StackPanel>
      <StackPanel Orientation="Horizontal">
        <Label Margin="10,10,0,10">ComputerName:</Label>
        <TextBox Name="Input" Margin="10" Width="250px"></TextBox>
      </StackPanel>
      <DockPanel>
        <Button Name="ButtonGetService" Content="Get-Service" Margin="10"
Width="150px" IsEnabled="false"/>
        <Button Name="ButtonClose" Content="Close" HorizontalAlignment="Right"
Margin="10" Width="50px"/>
      </DockPanel>
    </StackPanel>
  </ScrollViewer >
</Window>
'
```

```
# Create the Window Object
$Reader=(New-Object System.Xml.XmlNodeReader $XAMLWindow)
$Window=[Windows.Markup.XamlReader]::Load( $Reader )

# TextChanged Event Handler for Input
$TextboxInput = $Window.FindName("Input")
$TextboxInput.add_TextChanged.Invoke({
  $ComputerName = $TextboxInput.Text
  $ButtonGetService.IsEnabled = $ComputerName -ne ''
})

# Click Event Handler for ButtonClose
$ButtonClose = $Window.FindName("ButtonClose")
$ButtonClose.add_Click.Invoke({
  $Window.Close();
})

# Click Event Handler for ButtonGetService
$ButtonGetService = $Window.FindName("ButtonGetService")
$ButtonGetService.add_Click.Invoke({
  $ComputerName = $TextboxInput.text.Trim()
  try{
    Get-Service -ComputerName $computerName | Out-GridView -Title "Get-Service on
$ComputerName"
  }catch{
```

```
[System.Windows.MessageBox]::Show($_.exception.message, "Error", [System.Windows.MessageBoxButton]::OK, [S  
    }  
})  
  
# Open the Window  
$Window.ShowDialog() | Out-Null
```

Cela crée une fenêtre de dialogue qui permet à l'utilisateur de sélectionner un nom d'ordinateur, puis affiche une table de services et leurs états sur cet ordinateur. Cet exemple utilise WPF plutôt que Windows Forms.

Lire GUI dans Powershell en ligne: <https://riptutorial.com/fr/powershell/topic/7141/gui-dans-powershell>

Chapitre 28: HashTables

Introduction

Une table de hachage est une structure qui mappe les clés aux valeurs. Voir la [table de hachage](#) pour plus de détails.

Remarques

Un concept important qui repose sur les tables de hachage est la [fragmentation](#) . C'est très utile pour faire un grand nombre d'appels avec des paramètres répétitifs.

Exemples

Créer une table de hachage

Exemple de création d'un HashTable vide:

```
$hashTable = @{ }
```

Exemple de création d'une table de hachage avec des données:

```
$hashTable = @{  
    Name1 = 'Value'  
    Name2 = 'Value'  
    Name3 = 'Value3'  
}
```

Accédez à une valeur de table de hachage par clé.

Un exemple de définition d'une table de hachage et d'accès à une valeur par la clé

```
$hashTable = @{  
    Key1 = 'Value1'  
    Key2 = 'Value2'  
}  
$hashTable.Key1  
#output  
Value1
```

Exemple d'accès à une clé comportant des caractères non valides pour un nom de propriété:

```
$hashTable = @{  
    'Key 1' = 'Value3'  
    Key2 = 'Value4'  
}  
$hashTable.'Key 1'
```

```
#Output
Value3
```

En boucle sur une table de hachage

```
$hashTable = @{
    Key1 = 'Value1'
    Key2 = 'Value2'
}

foreach($key in $hashTable.Keys)
{
    $value = $hashTable.$key
    Write-Output "$key : $value"
}

#Output
Key1 : Value1
Key2 : Value2
```

Ajouter une paire de valeurs de clé à une table de hachage existante

Par exemple, pour ajouter une clé "Key2" avec une valeur "Value2" à la table de hachage, en utilisant l'opérateur d'addition:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable += @{Key2 = 'Value2'}
$hashTable

#Output

Name                Value
----                -
Key1                Value1
Key2                Value2
```

Un exemple, pour ajouter une clé "Key2" avec une valeur de "Value2" à la table de hachage à l'aide de la méthode Add:

```
$hashTable = @{
    Key1 = 'Value1'
}
$hashTable.Add("Key2", "Value2")
$hashTable

#Output

Name                Value
----                -
Key1                Value1
Key2                Value2
```

Enumérer à travers les clés et les paires valeur-clé

Enumérer à travers les clés

```
foreach ($key in $var1.Keys) {  
    $value = $var1[$key]  
    # or  
    $value = $var1.$key  
}
```

Enumérer à travers des paires valeur-clé

```
foreach ($keyvaluepair in $var1.GetEnumerator()) {  
    $key1 = $_.Key1  
    $val1 = $_.Val1  
}
```

Supprimer une paire de valeurs de clé d'une table de hachage existante

Par exemple, pour supprimer une clé "Key2" avec la valeur "Value2" de la table de hachage, à l'aide de l'opérateur de suppression:

```
$hashTable = @{  
    Key1 = 'Value1'  
    Key2 = 'Value2'  
}  
$hashTable.Remove("Key2", "Value2")  
$hashTable  
  
#Output  
  
Name                Value  
----                -  
Key1                Value1
```

Lire HashTables en ligne: <https://riptutorial.com/fr/powershell/topic/8083/hashtables>

Chapitre 29: Incorporation de code géré (C # | VB)

Introduction

Cette rubrique décrit brièvement comment le code géré C # ou VB .NET peut être scripté et utilisé dans un script PowerShell. Cette rubrique n'explore pas toutes les facettes de l'applet de commande Add-Type.

Pour plus d'informations sur l'applet de commande Add-Type, consultez la documentation MSDN (pour 5.1) ici: <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-type>

Paramètres

Paramètre	Détails
-TypeDefinition <String_>	Accepte le code sous forme de chaîne
-Langue <String_>	Spécifie la langue du code managé. Valeurs acceptées: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript

Remarques

Suppression de types ajoutés

Dans les versions ultérieures de PowerShell, Remove-TypeData a été ajouté aux bibliothèques d'applets de commande PowerShell, ce qui peut permettre la suppression d'un type au sein d'une session. Pour plus d'informations sur cette applet de commande, cliquez ici:

<https://msdn.microsoft.com/en-us/powershell/reference/4.0/microsoft.powershell.utility/remove-typedata>

Syntaxe CSharp et .NET

Pour ceux qui expérimentent avec .NET, il va sans dire que les différentes versions de C # peuvent être radicalement différentes dans leur niveau de prise en charge de certaines syntaxes.

Si vous utilisez Powershell 1.0 et / ou -Language CSharp, le code géré utilisera .NET 2.0, qui manque de nombreuses fonctionnalités que les développeurs C # utilisent généralement sans y penser, telles que Generics, Linq et Lambda. En plus de cela, le polymorphisme formel est traité avec les paramètres par défaut dans les versions ultérieures de C # / .NET.

Examples

C # Exemple

Cet exemple montre comment intégrer certains éléments C # de base dans un script PowerShell, les ajouter à l'espace d'exécution / session et utiliser le code dans la syntaxe PowerShell.

```
$code = "
using System;

namespace MyNameSpace
{
    public class Responder
    {
        public static void StaticRespond()
        {
            Console.WriteLine("Static Response");
        }

        public void Respond()
        {
            Console.WriteLine("Instance Respond");
        }
    }
}
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName] 'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language CSharp;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Exemple VB.NET

Cet exemple montre comment intégrer certains éléments C # de base dans un script PowerShell, les ajouter à l'espace d'exécution / session et utiliser le code dans la syntaxe PowerShell.

```
$code = @"
Imports System

Namespace MyNameSpace
    Public Class Responder
        Public Shared Sub StaticRespond()
            Console.WriteLine("Static Response")
        End Sub

        Public Sub Respond()
            Console.WriteLine("Instance Respond")
        End Sub
    End Class
End Namespace
"@
```

```
End Class
End Namespace
"@

# Check the type has not been previously added within the session, otherwise an exception is
raised
if (-not ([System.Management.Automation.PSTypeName] 'MyNameSpace.Responder').Type)
{
    Add-Type -TypeDefinition $code -Language VisualBasic;
}

[MyNameSpace.Responder]::StaticRespond();

$instance = New-Object MyNameSpace.Responder;
$instance.Respond();
```

Lire Incorporation de code géré (C # | VB) en ligne:

<https://riptutorial.com/fr/powershell/topic/9823/incorporation-de-code-gere--c-sharp---vb->

Chapitre 30: Introduction à Pester

Remarques

Pester est un framework de test pour PowerShell qui vous permet d'exécuter des scénarios de test pour votre code PowerShell. Il peut être utilisé pour exécuter ex. tests unitaires pour vous aider à vérifier que vos modules, scripts, etc. fonctionnent comme prévu.

[Qu'est-ce que Pester et pourquoi devrais-je m'en soucier?](#)

Exemples

Premiers pas avec Pester

Pour commencer à tester le code PowerShell à l'aide du module Pester, vous devez vous familiariser avec trois mots-clés / commandes:

- **Description** : Définit un groupe de tests. Tous les fichiers de test Pester nécessitent au moins un bloc de description.
- **It** : Définit un test individuel. Vous pouvez avoir plusieurs blocs It dans un bloc Describe.
- **Devrait** : La commande verify / test. Il est utilisé pour définir le résultat qui doit être considéré comme un test réussi.

Échantillon:

```
Import-Module Pester

#Sample function to run tests against
function Add-Numbers{
    param($a, $b)
    return [int]$a + [int]$b
}

#Group of tests
Describe "Validate Add-Numbers" {

    #Individual test cases
    It "Should add 2 + 2 to equal 4" {
        Add-Numbers 2 2 | Should Be 4
    }

    It "Should handle strings" {
        Add-Numbers "2" "2" | Should Be 4
    }

    It "Should return an integer"{
        Add-Numbers 2.3 2 | Should BeOfType Int32
    }
}
```

Sortie:

```
Describing Validate Add-Numbers  
[+] Should add 2 + 2 to equal 4 33ms  
[+] Should handle strings 19ms  
[+] Should return an integer 23ms
```

Lire Introduction à Pester en ligne: <https://riptutorial.com/fr/powershell/topic/5753/introduction-a-pester>

Chapitre 31: Introduction à Psake

Syntaxe

- Tâche - fonction principale pour exécuter une étape de votre script de construction
- Depends - propriété qui spécifie ce que dépend l'étape actuelle
- default - il doit toujours y avoir une tâche par défaut qui sera exécutée si aucune tâche initiale n'est spécifiée
- FormatTaskName - spécifie comment chaque étape est affichée dans la fenêtre de résultat.

Remarques

[psake](#) est un outil d'automatisation de build écrit en PowerShell, inspiré par Rake (make Ruby) et Bake (Boo make). Il est utilisé pour créer des builds en utilisant un modèle de dépendance. Documentation disponible [ici](#)

Exemples

Contour de base

```
Task Rebuild -Depends Clean, Build {
    "Rebuild"
}

Task Build {
    "Build"
}

Task Clean {
    "Clean"
}

Task default -Depends Build
```

Exemple FormatTaskName

```
# Will display task as:
# ----- Rebuild -----
# ----- Build -----
FormatTaskName "----- {0} -----"

# will display tasks in yellow colour:
# Running Rebuild
FormatTaskName {
    param($taskName)
    "Running $taskName" - foregroundcolor yellow
}

Task Rebuild -Depends Clean, Build {
```

```

    "Rebuild"
  }

Task Build {
    "Build"
}

Task Clean {
    "Clean"
}

Task default -Depends Build

```

Exécuter la tâche sous condition

```

properties {
    $isOk = $false
}

# By default the Build task won't run, unless there is a param $true
Task Build -precondition { return $isOk } {
    "Build"
}

Task Clean {
    "Clean"
}

Task default -Depends Build

```

ContinueOnError

```

Task Build -depends Clean {
    "Build"
}

Task Clean -ContinueOnError {
    "Clean"
    throw "throw on purpose, but the task will continue to run"
}

Task default -Depends Build

```

Lire Introduction à Psake en ligne: <https://riptutorial.com/fr/powershell/topic/5019/introduction-a-psake>

Chapitre 32: Jeux de paramètres

Introduction

Les jeux de paramètres servent à limiter la combinaison possible de paramètres ou à appliquer des paramètres lorsque 1 ou plusieurs paramètres sont sélectionnés.

Les exemples expliqueront l'utilisation et la raison d'un jeu de paramètres.

Exemples

Jeux de paramètres simples

```
function myFunction
{
    param(
        # If parameter 'a' is used, then 'c' is mandatory
        # If parameter 'b' is used, then 'c' is optional, but allowed
        # You can use parameter 'c' in combination with either 'a' or 'b'
        # 'a' and 'b' cannot be used together

        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [switch]$a,
        [parameter(ParameterSetName="BandC", mandatory=$true)]
        [switch]$b,
        [parameter(ParameterSetName="AandC", mandatory=$true)]
        [parameter(ParameterSetName="BandC", mandatory=$false)]
        [switch]$c
    )
    # $PSCmdlet.ParameterSetName can be used to check which parameter set was used
    Write-Host $PSCmdlet.ParameterSetName
}

# Valid syntaxes
myFunction -a -c
# => "Parameter set : AandC"
myFunction -b -c
# => "Parameter set : BandC"
myFunction -b
# => "Parameter set : BandC"

# Invalid syntaxes
myFunction -a -b
# => "Parameter set cannot be resolved using the specified named parameters."
myFunction -a
# => "Supply values for the following parameters:"
#     c:"
```

Parameterset pour imposer l'utilisation d'un paramètre lorsqu'un autre est sélectionné.

Lorsque vous voulez par exemple, utilisez le paramètre Password si le paramètre User est fourni.

(et vice versa)

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomethingToDo,
        [Parameter(ParameterSetName="Credentials", mandatory=$false)]
        [String]$Computername = "LocalHost",
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [String]$User,
        [Parameter(ParameterSetName="Credentials", mandatory=$true)]
        [SecureString]$Password
    )

    #Do something
}

# This will not work he will ask for user and password
Do-Something -SomethingToDo 'get-help about_Functions_Advanced' -ComputerName

# This will not work he will ask for password
Do-Something -SomethingToDo 'get-help about_Functions_Advanced' -User
```

Paramètre défini pour limiter la combinaison des paramètres

```
Function Do-Something
{
    Param
    (
        [Parameter(Mandatory=$true)]
        [String]$SomethingToDo,
        [Parameter(ParameterSetName="Silently", mandatory=$false)]
        [Switch]$Silently,
        [Parameter(ParameterSetName="Loudly", mandatory=$false)]
        [Switch]$Loudly
    )

    #Do something
}

# This will not work because you can not use the combination Silently and Loudly
Do-Something -SomethingToDo 'get-help about_Functions_Advanced' -Silently -Loudly
```

Lire Jeux de paramètres en ligne: <https://riptutorial.com/fr/powershell/topic/6598/jeux-de-parametres>

Chapitre 33: La gestion des erreurs

Introduction

Cette rubrique traite des types d'erreur et de la gestion des erreurs dans PowerShell.

Exemples

Types d'erreur

Une erreur est une erreur, on pourrait se demander comment il pourrait y avoir des types dedans. Eh bien, avec Powershell, l'erreur tombe en gros sur deux critères:

- Erreur de terminaison
- Erreur non terminale

Comme son nom l'indique, les erreurs de terminaison mettront fin à l'exécution et une erreur de non-terminaison laissera l'exécution continuer à l'instruction suivante.

Cela est vrai en supposant que la valeur **\$ ErrorActionPreference** est par défaut (Continue). `$ ErrorActionPreference` est une [variable Preference](#) qui indique à Powershell ce qu'il faut faire en cas d'erreur "non terminante".

Erreur de terminaison

Une erreur de terminaison peut être traitée avec une prise d'essai typique, comme ci-dessous

```
Try
{
    Write-Host "Attempting Divide By Zero"
    1/0
}
Catch
{
    Write-Host "A Terminating Error: Divide by Zero Caught!"
}
```

L'extrait ci-dessus s'exécutera et l'erreur sera interceptée par le bloc catch.

Erreur non terminale

Une erreur de non-terminaison dans l'autre main ne sera pas interceptée par défaut dans le bloc catch. La raison en est une erreur de non-terminaison n'est pas considérée comme une erreur critique.

```
Try
{
    Stop-Process -Id 123456
```

```
}  
Catch  
{  
    Write-Host "Non-Terminating Error: Invalid Process ID"  
}
```

Si vous exécutez la ligne ci-dessus, vous n'obtiendrez pas la sortie du bloc catch puisque l'erreur n'est pas considérée comme critique et que l'exécution continuera simplement à partir de la commande suivante. Cependant, l'erreur sera affichée dans la console. Pour gérer une erreur de non-terminaison, vous devez simplement modifier la préférence d'erreur.

```
Try  
{  
    Stop-Process -Id 123456 -ErrorAction Stop  
}  
Catch  
{  
    "Non-Terminating Error: Invalid Process ID"  
}
```

Maintenant, avec la préférence d'erreur mise à jour, cette erreur sera considérée comme une erreur de terminaison et sera interceptée dans le bloc catch.

invoquer des erreurs de terminaison et de non-terminaison:

La cmdlet **Write-Error** écrit simplement l'erreur dans le programme hôte invoquant. Cela n'arrête pas l'exécution. Où comme **throw** vous donnera une erreur de fin et arrêter l'exécution

```
Write-host "Going to try a non terminating Error "  
Write-Error "Non terminating"  
Write-host "Going to try a terminating Error "  
throw "Terminating Error "  
Write-host "This Line wont be displayed"
```

Lire [La gestion des erreurs en ligne](https://riptutorial.com/fr/powershell/topic/8075/la-gestion-des-erreurs): <https://riptutorial.com/fr/powershell/topic/8075/la-gestion-des-erreurs>

Chapitre 34: Les opérateurs

Introduction

Un opérateur est un caractère qui représente une action. Il indique au compilateur / interprète d'effectuer des opérations mathématiques, relationnelles ou logiques spécifiques et de produire un résultat final. PowerShell interprète de manière spécifique et classe en conséquence, comme les opérateurs arithmétiques effectuent des opérations principalement sur les nombres, mais ils affectent également les chaînes et autres types de données. Avec les opérateurs de base, PowerShell dispose d'un certain nombre d'opérateurs qui permettent de gagner du temps et de réduire les efforts de codage (ex: -comme, -match, -remplace, etc.).

Exemples

Opérateurs arithmétiques

```
1 + 2      # Addition
1 - 2      # Subtraction
-1         # Set negative value
1 * 2      # Multiplication
1 / 2      # Division
1 % 2      # Modulus
100 -shl 2 # Bitwise Shift-left
100 -shr 1 # Bitwise Shift-right
```

Opérateurs logiques

```
-and # Logical and
-or  # Logical or
-xor # Logical exclusive or
-not # Logical not
!    # Logical not
```

Opérateurs d'affectation

Arithmétique simple:

```
$var = 1      # Assignment. Sets the value of a variable to the specified value
$var += 2     # Addition. Increases the value of a variable by the specified value
$var -= 1     # Subtraction. Decreases the value of a variable by the specified value
$var *= 2     # Multiplication. Multiplies the value of a variable by the specified value
$var /= 2     # Division. Divides the value of a variable by the specified value
$var %= 2     # Modulus. Divides the value of a variable by the specified value and then
              # assigns the remainder (modulus) to the variable
```

Incrémenter et décrémenter:

```
$var++ # Increases the value of a variable, assignable property, or array element by 1
$var-- # Decreases the value of a variable, assignable property, or array element by 1
```

Opérateurs de comparaison

Les opérateurs de comparaison PowerShell se composent d'un tiret (-) suivi d'un nom (`eq` pour `equal` , `gt` pour `greater than` , etc.).

Les noms peuvent être précédés de caractères spéciaux pour modifier le comportement de l'opérateur:

```
i # Case-Insensitive Explicit (-ieq)
c # Case-Sensitive Explicit (-ceq)
```

Case-Insensitive est la valeur par défaut si elle n'est pas spécifiée ("`a`" -`eq` "`A`") identique à ("`a`" -`ieq` "`A`").

Opérateurs de comparaison simples:

```
2 -eq 2 # Equal to (==)
2 -ne 4 # Not equal to (!=)
5 -gt 2 # Greater-than (>)
5 -ge 5 # Greater-than or equal to (>=)
5 -lt 10 # Less-than (<)
5 -le 5 # Less-than or equal to (<=)
```

Opérateurs de comparaison de chaînes:

```
"MyString" -like "*String" # Match using the wildcard character (*)
"MyString" -notlike "Other*" # Does not match using the wildcard character (*)
"MyString" -match "$String^" # Matches a string using regular expressions
"MyString" -notmatch "$Other^" # Does not match a string using regular expressions
```

Opérateurs de comparaison de collections:

```
"abc", "def" -contains "def" # Returns true when the value (right) is present
# in the array (left)
"abc", "def" -notcontains "123" # Returns true when the value (right) is not present
# in the array (left)
"def" -in "abc", "def" # Returns true when the value (left) is present
# in the array (right)
"123" -notin "abc", "def" # Returns true when the value (left) is not present
# in the array (right)
```

Opérateurs de redirection

Flux de sortie réussi:

```
cmdlet > file # Send success output to file, overwriting existing content
cmdlet >> file # Send success output to file, appending to existing content
cmdlet 1>&2 # Send success and error output to error stream
```

Erreur de flux de sortie:

```
cmdlet 2> file      # Send error output to file, overwriting existing content
cmdlet 2>> file    # Send error output to file, appending to existing content
cmdlet 2>&1         # Send success and error output to success output stream
```

Flux de sortie d'avertissement: (PowerShell 3.0+)

```
cmdlet 3> file      # Send warning output to file, overwriting existing content
cmdlet 3>> file    # Send warning output to file, appending to existing content
cmdlet 3>&1         # Send success and warning output to success output stream
```

Flux de sortie détaillé: (PowerShell 3.0+)

```
cmdlet 4> file      # Send verbose output to file, overwriting existing content
cmdlet 4>> file    # Send verbose output to file, appending to existing content
cmdlet 4>&1         # Send success and verbose output to success output stream
```

Flux de sortie de débogage: (PowerShell 3.0+)

```
cmdlet 5> file      # Send debug output to file, overwriting existing content
cmdlet 5>> file    # Send debug output to file, appending to existing content
cmdlet 5>&1         # Send success and debug output to success output stream
```

Flux de sortie d'information: (PowerShell 5.0+)

```
cmdlet 6> file      # Send information output to file, overwriting existing content
cmdlet 6>> file    # Send information output to file, appending to existing content
cmdlet 6>&1         # Send success and information output to success output stream
```

Tous les flux de sortie:

```
cmdlet *> file      # Send all output streams to file, overwriting existing content
cmdlet *>> file    # Send all output streams to file, appending to existing content
cmdlet *>&1         # Send all output streams to success output stream
```

Différences par rapport à l'opérateur de tuyauterie (|)

Les opérateurs de redirection redirigent uniquement les flux vers des fichiers ou des flux vers des flux. L'opérateur de canalisation achemine un objet dans le pipeline vers une applet de commande ou la sortie. Le fonctionnement du pipeline diffère en général du fonctionnement de la redirection et peut être lu lors [de l'utilisation du pipeline PowerShell](#)

Mélanger les types d'opérandes: le type de l'opérande gauche détermine le comportement.

Pour ajout

```
"4" + 2           # Gives "42"
4 + "2"           # Gives 6
```

```
1,2,3 + "Hello" # Gives 1,2,3,"Hello"  
"Hello" + 1,2,3 # Gives "Hello1 2 3"
```

Pour la multiplication

```
"3" * 2 # Gives "33"  
2 * "3" # Gives 6  
1,2,3 * 2 # Gives 1,2,3,1,2,3  
2 * 1,2,3 # Gives an error op_Multiply is missing
```

L'impact peut avoir des conséquences cachées sur les opérateurs de comparaison:

```
$a = Read-Host "Enter a number"  
Enter a number : 33  
$a -gt 5  
False
```

Opérateurs de manipulation de chaînes

Remplacez l'opérateur:

L'opérateur `-replace` remplace un motif dans une valeur d'entrée à l'aide d'une expression régulière. Cet opérateur utilise deux arguments (séparés par une virgule): un modèle d'expression régulière et sa valeur de remplacement (qui est facultative et une chaîne vide par défaut).

```
"The rain in Seattle" -replace 'rain','hail' #Returns: The hail in Seattle  
"kenmyer@contoso.com" -replace '^[\\w]+@(\\.)', '$1' #Returns: contoso.com
```

Opérateurs Split et Join:

L'opérateur `-split` divise une chaîne en un tableau de sous-chaînes.

```
"A B C" -split " " #Returns an array string collection object containing A,B and C.
```

L'opérateur `-join` joint un tableau de chaînes en une seule chaîne.

```
"E","F","G" -join ":" #Returns a single string: E:F:G
```

Lire Les opérateurs en ligne: <https://riptutorial.com/fr/powershell/topic/1071/les-operateurs>

Chapitre 35: Ligne de commande PowerShell.exe

Paramètres

Paramètre	La description
-Aide -? /?	Affiche l'aide
-Fichier <FilePath> [<Args>]	Chemin d'accès au fichier script qui doit être exécuté et aux arguments (facultatif)
-Command {- <script-block> [-args <arg-array>] <string> [<CommandParameters>]}	Commandes à exécuter suivies d'arguments
-EncodedCommand <Base64EncodedCommand>	Commandes encodées en base64
-ExecutionPolicy <ExecutionPolicy>	Définit la stratégie d'exécution pour ce processus uniquement
-InputFormat {Texte XML}	Définit le format d'entrée des données envoyées au processus. Texte (chaînes) ou XML (CLIXML sérialisé)
-Mta	PowerShell 3.0+: exécute PowerShell dans un appartement multithread (STA par défaut)
-Sta	PowerShell 2.0: Exécute PowerShell dans un appartement mono-thread (MTA par défaut)
-Sans issue	Laisse la console PowerShell en cours d'exécution après l'exécution du script / commande
-Pas de logo	Masque la bannière de copyright au lancement
-NonInteractive	Cache la console de l'utilisateur
-Aucun profil	Évitez de charger des profils PowerShell pour une machine ou un utilisateur
-OutputFormat {Texte XML}	Définit le format de sortie pour les données renvoyées par PowerShell. Texte (chaînes)

Paramètre	La description
	ou XML (CLIXML sérialisé)
-PSConsoleFile <FilePath>	Charge un fichier de console pré-créé qui configure l'environnement (créé à l'aide de <code>Export-Console</code>)
-Version <version de Windows PowerShell>	Spécifiez une version de PowerShell à exécuter. Principalement utilisé avec 2.0
-WindowStyle <style>	Indique si le processus PowerShell doit être démarré en tant que fenêtre <code>normal</code> , <code>hidden</code> , <code>minimized</code> OU <code>maximized</code> .

Exemples

Exécuter une commande

Le paramètre `-Command` permet de spécifier les commandes à exécuter au lancement. Il prend en charge plusieurs entrées de données.

-Command <string>

Vous pouvez spécifier des commandes à exécuter lors du lancement sous forme de chaîne. Point-virgule multiple ; des instructions séparées peuvent être exécutées.

```
>PowerShell.exe -Command "(Get-Date).ToShortDateString()"
10.09.2016

>PowerShell.exe -Command "(Get-Date).ToShortDateString(); 'PowerShell is fun!'"
10.09.2016
PowerShell is fun!
```

-Command {scriptblock}

Le paramètre `-Command` prend également en charge une entrée de scriptblock (une ou plusieurs instructions `{ #code }` accolades `{ #code }`). Cela ne fonctionne que lorsque vous appelez `PowerShell.exe` depuis une autre session Windows PowerShell).

```
PS > powershell.exe -Command {
"This can be useful, sometimes..."
(Get-Date).ToShortDateString()
}
This can be useful, sometimes...
10.09.2016
```

-Command - (entrée standard)

Vous pouvez passer des commandes à partir de l'entrée standard en utilisant `-Command -`. L'entrée standard peut provenir de l' `echo`, de la lecture d'un fichier, d'une application de console héritée, etc.

```
>echo "Hello World";"Greetings from PowerShell" | PowerShell.exe -NoProfile -Command -  
Hello World  
Greetings from PowerShell
```

Exécuter un fichier script

Vous pouvez spécifier un fichier à un `ps1` pour exécuter son contenu lors du lancement à l'aide du paramètre `-File`.

Script de base

MyScript.ps1

```
(Get-Date).ToShortDateString()  
"Hello World"
```

Sortie:

```
>PowerShell.exe -File Desktop\MyScript.ps1  
10.09.2016  
Hello World
```

Utiliser des paramètres et des arguments

Vous pouvez ajouter des paramètres et / ou des arguments après filepath pour les utiliser dans le script. Les arguments seront utilisés comme valeurs pour les paramètres de script non définis / disponibles, le reste sera disponible dans le tableau `$args`

MyScript.ps1

```
param($Name)  
  
"Hello $Name! Today's date it $((Get-Date).ToShortDateString())"  
"First arg: $($args[0])"
```

Sortie:

```
>PowerShell.exe -File Desktop\MyScript.ps1 -Name StackOverflow foo  
Hello StackOverflow! Today's date it 10.09.2016
```

```
First arg: foo
```

Lire Ligne de commande PowerShell.exe en ligne:

<https://riptutorial.com/fr/powershell/topic/5839/ligne-de-commande-powershell-exe>

Chapitre 36: Logique conditionnelle

Syntaxe

- if (expression) {}
- if (expression) {} else {}
- if (expression) {} elseif (expression) {}
- if (expression) {} elseif (expression) {} else {}

Remarques

Voir aussi [Opérateurs de comparaison](#) , utilisables dans les expressions conditionnelles.

Exemples

si, sinon et sinon si

Powershell prend en charge les opérateurs de logique conditionnelle standard, comme beaucoup de langages de programmation. Celles-ci permettent d'exécuter certaines fonctions ou commandes dans des circonstances particulières.

Avec un `if` les commandes à l'intérieur des parenthèses (`{}`) ne sont exécutées que si les conditions à l'intérieur du `if` (`()`) sont remplies

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if condition met"
}
```

Vous pouvez aussi faire `else` . Ici, les commandes `else` sont exécutées si les conditions `if` ne sont **pas** remplies:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
else{
    Write-Host "if condition not met"
}
```

ou un `elseif` . Un autre s'exécute les commandes si le `if` les conditions ne sont pas remplies et les `elseif` conditions sont remplies:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
```

```
elseif ($test -eq "test"){
    Write-Host "ifelse condition met"
}
```

Notez que l'utilisation ci-dessus `-eq` (égalité) `CmdLet` et non `=` ou `==` comme beaucoup d'autres langues le font pour l'équité.

Négation

Vous pouvez vouloir annuler une valeur booléenne, c'est-à-dire entrer une instruction `if` lorsqu'une condition est fautive plutôt que vraie. Cela peut être fait en utilisant le `-Not` `CmdLet`

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if condition not met"
}
```

Vous pouvez aussi utiliser `!` :

```
$test = "test"
if (!$test -eq "test2"){
    Write-Host "if condition not met"
}
```

il y a aussi l' `-ne` (pas égal):

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test is not equal to 'test2'"
}
```

Si sténographie conditionnelle

Si vous souhaitez utiliser la sténographie, vous pouvez utiliser une logique conditionnelle avec le raccourci suivant. Seule la chaîne 'false' sera évaluée à true (2.0).

```
#Done in Powershell 2.0
$boolean = $false;
$string = "false";
$emptyString = "";

If($boolean){
    # this does not run because $boolean is false
    Write-Host "Shorthand If conditions can be nice, just make sure they are always boolean."
}

If($string){
    # This does run because the string is non-zero length
    Write-Host "If the variable is not strictly null or Boolean false, it will evaluate to true as it is an object or string with length greater than 0."
}

If($emptyString){
```

```
# This does not run because the string is zero-length
Write-Host "Checking empty strings can be useful as well."
}

If($null){
  # This does not run because the condition is null
  Write-Host "Checking Nulls will not print this statement."
}
```

Lire Logique conditionnelle en ligne: <https://riptutorial.com/fr/powershell/topic/7208/logique-conditionnelle>

Chapitre 37: Module ActiveDirectory

Introduction

Cette rubrique présente certaines des applets de commande de base utilisées dans le module Active Directory pour PowerShell pour manipuler des utilisateurs, des groupes, des ordinateurs et des objets.

Remarques

N'oubliez pas que le système d'aide de PowerShell est l'une des meilleures ressources que vous puissiez éventuellement utiliser.

```
Get-Help Get-ADUser -Full
Get-Help Get-ADGroup -Full
Get-Help Get-ADComputer -Full
Get-Help Get-ADObject -Full
```

Toute la documentation d'aide fournira des exemples, de la syntaxe et des paramètres d'aide.

Exemples

Module

```
#Add the ActiveDirectory Module to current PowerShell Session
Import-Module ActiveDirectory
```

Utilisateurs

Récupérer l'utilisateur Active Directory

```
Get-ADUser -Identity JohnSmith
```

Récupérer toutes les propriétés associées à l'utilisateur

```
Get-ADUser -Identity JohnSmith -Properties *
```

Récupérer les propriétés sélectionnées pour l'utilisateur

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name, Mail
```

Nouvel utilisateur AD

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path "CN=Users,DC=Domain,DC=Local"
```

Groupes

Récupérer le groupe Active Directory

```
Get-ADGroup -Identity "My-First-Group" #Ensure if group name has space quotes are used
```

Récupérer toutes les propriétés associées à un groupe

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Récupérer tous les membres d'un groupe

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName  
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Ajouter un utilisateur AD à un groupe AD

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

Nouveau groupe AD

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

Des ordinateurs

Récupérer un ordinateur AD

```
Get-ADComputer -Identity "JohnLaptop"
```

Récupérer toutes les propriétés associées à l'ordinateur

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Récupérer les propriétés de sélection de l'ordinateur

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Objets

Récupérer un objet Active Directory

```
#Identity can be ObjectGUID, Distinguished Name or many more  
Get-ADObject -Identity "ObjectGUID07898"
```

Déplacer un objet Active Directory

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath  
"OU=SuperUser,DC=Domain,DC=Local"
```

Modifier un objet Active Directory

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is  
My First Object Modification"
```

Lire Module ActiveDirectory en ligne: <https://riptutorial.com/fr/powershell/topic/8213/module-activedirectory>

Chapitre 38: Module d'archive

Introduction

Le module d'archivage `Microsoft.PowerShell.Archive` fournit des fonctions pour stocker des fichiers dans des archives ZIP (`Compress-Archive`) et les extraire (`Expand-Archive`). Ce module est disponible dans PowerShell 5.0 et supérieur.

Dans les versions antérieures de PowerShell, les [extensions de communauté](#) ou `.NET System.IO.Compression.FileSystem` pouvaient être utilisées.

Syntaxe

- **Expand-Archive / Compress-Archive**
- **-Chemin**
 - le chemin du ou des fichiers à compresser (Compress-Archive) ou le chemin de l'archive pour extraire le ou les fichiers (Expand-Archive)
 - il y a plusieurs autres options liées au chemin, veuillez voir ci-dessous.
- **-DestinationPath** (optionnel)
 - Si vous ne fournissez pas ce chemin, l'archive sera créée dans le répertoire de travail actuel (Compress-Archive) ou le contenu de l'archive sera extrait dans le répertoire de travail actuel (Expand-Archive).

Paramètres

Paramètre	Détails
CompressionLevel	(<i>Compression-Archive uniquement</i>) Définissez le niveau de compression sur <code>Fastest</code> , <code>Optimal</code> OU <code>NoCompression</code>
Confirmer	Invite à confirmer avant de lancer
Obliger	Force la commande à s'exécuter sans confirmation
LiteralPath	Chemin qui est utilisé littéralement, les <i>caractères génériques</i> sont pris en charge, l'utilisation <code>*,*</code> pour spécifier des chemins multiples
Chemin	Chemin pouvant contenir des caractères génériques, utiliser <code>*,*</code> pour spécifier plusieurs chemins
Mettre à jour	(<i>Compression-Archive uniquement</i>) Mettre à jour les archives existantes
Et qu'est-ce qui se passerait si	Simuler la commande

Remarques

Voir [MSDN Microsoft.PowerShell.Archive \(5.1\)](#) pour plus d'informations.

Exemples

Compresser les archives avec un joker

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath  
C:\Archives\Documents.zip
```

Cette commande:

- Comprime tous les fichiers dans `C:\Documents`
- Utilise la compression `Optimal`
- Enregistrez l'archive résultante dans `C:\Archives\Documents.zip`
 - `-DestinationPath` ajoutera `.zip` s'il n'est pas présent.
 - `-LiteralPath` peut être utilisé si vous avez besoin de le nommer sans `.zip`.

Mettre à jour le ZIP existant avec Compress-Archive

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- cela va ajouter ou remplacer tous les fichiers `Documents.zip` avec les nouveaux de `C:\Documents`

Extraire un zip avec Expand-Archive

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- Cela va extraire tous les fichiers de `Documents.zip` dans le dossier `C:\Documents`

Lire Module d'archive en ligne: <https://riptutorial.com/fr/powershell/topic/9896/module-d-archive>

Chapitre 39: Module de tâches planifiées

Introduction

Exemples d'utilisation du module Tâches programmées disponibles dans Windows 8 / Server 2012 et les versions ultérieures.

Exemples

Exécuter un script PowerShell dans une tâche planifiée

Crée une tâche planifiée qui s'exécute immédiatement, puis au démarrage pour exécuter

C:\myscript.ps1 tant que SYSTEM

```
$ScheduledTaskPrincipal = New-ScheduledTaskPrincipal -UserId "SYSTEM" -LogonType
ServiceAccount
$ScheduledTaskTrigger1 = New-ScheduledTaskTrigger -AtStartup
$ScheduledTaskTrigger2 = New-ScheduledTaskTrigger -Once -At $(Get-Date) -RepetitionInterval
"00:01:00" -RepetitionDuration $([timeSpan] "24855.03:14:07")
$ScheduledTaskActionParams = @{
    Execute = "PowerShell.exe"
    Argument = '-executionpolicy Bypass -NonInteractive -c C:\myscript.ps1 -verbose >>
C:\output.log 2>&1"'
}
$ScheduledTaskAction = New-ScheduledTaskAction @ScheduledTaskActionParams
Register-ScheduledTask -Principal $ScheduledTaskPrincipal -Trigger
@($ScheduledTaskTrigger1,$ScheduledTaskTrigger2) -TaskName "Example Task" -Action
$ScheduledTaskAction
```

Lire Module de tâches planifiées en ligne: <https://riptutorial.com/fr/powershell/topic/10940/module-de-taches-planifiees>

Chapitre 40: Module ISE

Introduction

L'environnement de script intégré Windows PowerShell (ISE) est une application hôte qui vous permet d'écrire, d'exécuter et de tester des scripts et des modules dans un environnement graphique et intuitif. Les fonctionnalités clés de Windows PowerShell ISE incluent la coloration de la syntaxe, l'achèvement des tabulations, Intellisense, le débogage visuel, la conformité Unicode et l'aide contextuelle, et fournissent une expérience de script riche.

Exemples

Scripts de test

L'utilisation simple mais puissante de l'ISE consiste par exemple à écrire du code dans la section supérieure (avec une coloration syntaxique intuitive) et à exécuter le code simplement en le marquant et en appuyant sur la touche F8.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum
}

1..10 | Get-Sum

#output
55
```

Lire Module ISE en ligne: <https://riptutorial.com/fr/powershell/topic/10954/module-ise>

Chapitre 41: Module SharePoint

Exemples

Chargement du composant logiciel enfichable SharePoint

Le chargement de SharePoint Snapin peut être effectué en utilisant les éléments suivants:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

Cela ne fonctionne que dans la version 64 bits de PowerShell. Si la fenêtre indique "Windows PowerShell (x86)" dans le titre, vous utilisez la version incorrecte.

Si le composant logiciel enfichable est déjà chargé, le code ci-dessus provoquera une erreur. L'utilisation de ce qui suit ne chargera que si nécessaire, ce qui peut être utilisé dans les applets de commande / fonctions:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

Sinon, si vous démarrez SharePoint Management Shell, il inclura automatiquement le composant logiciel enfichable.

Pour obtenir une liste de toutes les cmdlets SharePoint disponibles, exécutez les opérations suivantes:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Itérer toutes les listes d'une collection de sites

Imprimez tous les noms de liste et le nombre d'articles.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Prints list title and item count
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Obtenez toutes les fonctionnalités installées sur une collection de sites

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature peut également être exécuté sur une étendue Web (`-Web <WebUrl>`), une étendue

de batterie (`-Farm`) et une étendue d'application Web (`-WebApplication <WebAppUrl>`).

Obtenir toutes les fonctionnalités orphelines sur une collection de sites

Une autre utilisation de `Get-SPFeature` peut être de rechercher toutes les fonctionnalités sans portée:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null }
```

Lire Module SharePoint en ligne: <https://riptutorial.com/fr/powershell/topic/5147/module-sharepoint>

Chapitre 42: Modules Powershell

Introduction

À partir de PowerShell version 2.0, les développeurs peuvent créer des modules PowerShell. Les modules PowerShell encapsulent un ensemble de fonctionnalités communes. Par exemple, il existe des modules PowerShell spécifiques aux fournisseurs qui gèrent différents services de cloud. Il existe également des modules PowerShell génériques qui interagissent avec les services de médias sociaux et effectuent des tâches de programmation courantes, telles que l'encodage Base64, l'utilisation de canaux nommés, etc.

Les modules peuvent exposer des alias de commandes, des fonctions, des variables, des classes, etc.

Exemples

Créer un manifeste de module

```
@{
  RootModule = 'MyCoolModule.psm1'
  ModuleVersion = '1.0'
  CompatiblePSEditions = @('Core')
  GUID = '6b42c995-67da-4139-be79-597a328056cc'
  Author = 'Bob Schmob'
  CompanyName = 'My Company'
  Copyright = '(c) 2017 Administrator. All rights reserved.'
  Description = 'It does cool stuff.'
  FunctionsToExport = @()
  CmdletsToExport = @()
  VariablesToExport = @()
  AliasesToExport = @()
  DscResourcesToExport = @()
}
```

Chaque bon module PowerShell possède un manifeste de module. Le manifeste du module contient simplement des métadonnées sur un module PowerShell et ne définit pas le contenu réel du module.

Le fichier manifeste est un fichier de script PowerShell, avec une extension de fichier `.psd1`, qui contient une table de hachage. La table de hachage dans le manifeste doit contenir des clés spécifiques, afin que PowerShell puisse l'interpréter correctement en tant que fichier de module PowerShell.

L'exemple ci-dessus fournit une liste des clés HashTable principales qui constituent un manifeste de module, mais il en existe bien d'autres. La commande `New-ModuleManifest` vous aide à créer un nouveau squelette de manifeste de module.

Exemple de module simple

```
function Add {
    [CmdletBinding()]
    param (
        [int] $x
        , [int] $y
    )

    return $x + $y
}

Export-ModuleMember -Function Add
```

Voici un exemple simple de ce à quoi un fichier de module de script PowerShell peut ressembler. Ce fichier s'appellera `MyCoolModule.psm1` et sera référencé à partir du fichier manifeste du module (`.psd1`). Vous remarquerez que la commande `Export-ModuleMember` nous permet de spécifier les fonctions du module que nous voulons "exporter" ou exposer à l'utilisateur du module. Certaines fonctions seront exclusivement internes et ne devraient pas être exposées. `Export-ModuleMember` seraient donc omises lors de l'appel à `Export-ModuleMember`.

Exportation d'une variable à partir d'un module

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

Pour exporter une variable depuis un module, utilisez la commande `Export-ModuleMember`, avec le paramètre `-Variable`. Rappelez-vous cependant que si la variable n'est pas explicitement exportée dans le fichier manifeste du module (`.psd1`), la variable ne sera pas visible pour le consommateur du module. Considérez le module comme un "gatekeeper". Si une fonction ou une variable n'est pas autorisée dans le manifeste du module, elle ne sera pas visible pour le consommateur du module.

Remarque: L'exportation d'une variable est similaire à la création d'un champ dans une classe publique. Ce n'est pas conseillé. Il serait préférable d'exposer une fonction pour obtenir le champ et une fonction pour définir le champ.

Structuration des modules PowerShell

Plutôt que de définir toutes vos fonctions dans un seul fichier de module de script `.psm1` PowerShell, vous souhaitez peut-être séparer votre fonction en fichiers individuels. Vous pouvez ensuite doter en source ces fichiers de votre fichier de module de script, qui les traite essentiellement comme s'ils faisaient partie du fichier `.psm1` lui-même.

Considérez cette structure de répertoire de module:

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

Dans votre fichier `MyCoolModule.psm1` , vous pouvez insérer le code suivant:

```
Get-ChildItem -Path $PSScriptRoot\Functions |
  ForEach-Object -Process { . $PSItem.FullName }
```

Cela source-dot les fichiers de fonction individuels dans le fichier de module `.psm1` .

Emplacement des modules

PowerShell recherche les modules dans les répertoires répertoriés dans `$ Env: PSModulepath`.

Un module appelé `foo` , dans un dossier appelé `foo` sera trouvé avec `Import-Module foo`

Dans ce dossier, PowerShell recherchera un manifeste de module (`foo.psd1`), un fichier de module (`foo.psm1`), une DLL (`foo.dll`).

Visibilité du membre du module

Par défaut, seules les fonctions définies dans un module sont visibles en dehors du module. En d'autres termes, si vous définissez des variables et des alias dans un module, ils ne seront pas disponibles sauf dans le code du module.

Pour remplacer ce comportement, vous pouvez utiliser la cmdlet `Export-ModuleMember` . Il a des paramètres appelés `-Function` , `-Variable` et `-Alias` qui vous permettent de spécifier exactement quels membres sont exportés.

Il est important de noter que si vous utilisez `Export-ModuleMember` , **seuls** les éléments que vous spécifiez seront visibles.

Lire Modules Powershell en ligne: <https://riptutorial.com/fr/powershell/topic/8734/modules-powershell>

Chapitre 43: Modules, scripts et fonctions

Introduction

Les modules PowerShell apportent une *extension* à l'administrateur système, à l'administrateur de base de données et au développeur. Que ce soit simplement comme une méthode pour partager des fonctions et des scripts.

Les fonctions de Powershell évitent les codes de répétition. Voir [Fonctions PS] [1] [1]: [Fonctions PowerShell](#)

Les scripts PowerShell sont utilisés pour automatiser les tâches administratives qui consistent en un shell de ligne de commande et des applets de commande associées construites au-dessus de .NET Framework.

Exemples

Fonction

Une fonction est un bloc de code nommé utilisé pour définir un code réutilisable facile à utiliser. Il est généralement inclus dans un script pour aider à réutiliser le code (pour éviter le code en double) ou distribué dans le cadre d'un module pour le rendre utile pour d'autres dans plusieurs scripts.

Scénarios où une fonction pourrait être utile:

- Calculer la moyenne d'un groupe de nombres
- Générer un rapport pour les processus en cours d'exécution
- Ecrire une fonction testant qu'un ordinateur est "sain" en envoyant `c$` requête ping à l'ordinateur et en accédant au partage `c$`

Les fonctions sont créées à l'aide du mot-clé `function`, suivi d'un nom à mot unique et d'un bloc de script contenant le code à exécuter lorsque le nom de la fonction est appelé.

```
function NameOfFunction {
    Your code
}
```

Démo

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}
```

Usage:

```
> HelloWorld
Greetings from PowerShell!
```

Scénario

Un script est un fichier texte avec l'extension de fichier `.ps1` qui contient des commandes PowerShell qui seront exécutées lors de l'appel du script. Les scripts étant des fichiers enregistrés, ils sont faciles à transférer entre les ordinateurs.

Les scripts sont souvent écrits pour résoudre un problème spécifique, par exemple:

- Exécuter une tâche de maintenance hebdomadaire
- Pour installer et configurer une solution / application sur un ordinateur

Démo

MyFirstScript.ps1:

```
Write-Host "Hello World!"
2+2
```

Vous pouvez exécuter un script en entrant le chemin d'accès au fichier à l'aide de:

- Chemin absolu, ex. `c:\MyFirstScript.ps1`
- Chemin relatif, ex. `.\MyFirstScript.ps1` si le répertoire en cours de votre console PowerShell était `C:\`

Usage:

```
> .\MyFirstScript.ps1
Hello World!
4
```

Un script peut également importer des modules, définir ses propres fonctions, etc.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}

HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Usage:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
```

```
Let's get started!  
4  
Greetings from PowerShell!
```

Module

Un module est un ensemble de fonctions réutilisables (ou applets de commande) pouvant être facilement distribuées à d'autres utilisateurs PowerShell et utilisées dans plusieurs scripts ou directement dans la console. Un module est généralement enregistré dans son propre répertoire et se compose de:

- Un ou plusieurs fichiers de code avec l'extension de fichier `.psm1` contenant des fonctions ou des assemblys binaires (`.dll`) contenant des applets de commande
- Un manifeste de module `.psd1` décrivant le nom du module, la version, l'auteur, la description, les fonctions / applets de commande qu'il fournit, etc.
- Autres exigences pour que cela fonctionne incl. dépendances, scripts etc.

Exemples de modules:

- Un module contenant des fonctions / cmdlets qui effectuent des statistiques sur un dataset
- Un module pour interroger et configurer des bases de données

Pour faciliter la recherche et l'importation d'un module par PowerShell, il est souvent placé dans l'un des emplacements de module PowerShell connus définis dans `$env:PSModulePath`.

Démo

Liste des modules installés sur l'un des emplacements de module connus:

```
Get-Module -ListAvailable
```

Importer un module, ex. Module `Hyper-V` :

```
Import-Module Hyper-V
```

Liste les commandes disponibles dans un module, ex. le module `Microsoft.PowerShell.Archive`

```
> Import-Module Microsoft.PowerShell.Archive  
> Get-Command -Module Microsoft.PowerShell.Archive
```

CommandType	Name	Version	Source
Function	Compress-Archive	1.0.1.0	Microsoft.PowerShell.Archive
Function	Expand-Archive	1.0.1.0	Microsoft.PowerShell.Archive

Fonctions avancées

Les fonctions avancées se comportent de la même manière que les applets de commande. Le

PowerShell ISE inclut deux squelettes de fonctions avancées. Accédez à ceux-ci via le menu, éditer, des extraits de code ou par Ctrl + J. (À partir de PS 3.0, les versions ultérieures peuvent différer)

Les fonctions clés incluent notamment:

- `Get-Help` intégrée et personnalisée pour la fonction, accessible via `Get-Help`
- peut utiliser `[CmdletBinding()]` qui fait agir la fonction comme une applet de commande
- options étendues de paramètres

Version simple:

```
<#
.Synopsis
    Short description
.DESCRIPTION
    Long description
.EXAMPLE
    Example of how to use this cmdlet
.EXAMPLE
    Another example of how to use this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding()]
    [OutputType([int])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
                    ValueFromPipelineByPropertyName=$true,
                    Position=0)]
        $Param1,

        # Param2 help description
        [int]
        $Param2
    )

    Begin
    {
    }
    Process
    {
    }
    End
    {
    }
}
```

Version complète:

```
<#
.Synopsis
    Short description
.DESCRIPTION
    Long description
```

```

.EXAMPLE
    Example of how to use this cmdlet
.EXAMPLE
    Another example of how to use this cmdlet
.INPUTS
    Inputs to this cmdlet (if any)
.OUTPUTS
    Output from this cmdlet (if any)
.NOTES
    General notes
.COMPONENT
    The component this cmdlet belongs to
.ROLE
    The role this cmdlet belongs to
.FUNCTIONALITY
    The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
        SupportsShouldProcess=$true,
        PositionalBinding=$false,
        HelpUri = 'http://www.microsoft.com/',
        ConfirmImpact='Medium')]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true,
            ValueFromRemainingArguments=$false,
            Position=0,
            ParameterSetName='Parameter Set 1')]
        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
        $Param1,

        # Param2 help description
        [Parameter(ParameterSetName='Parameter Set 1')]
        [AllowNull()]
        [AllowEmptyCollection()]
        [AllowEmptyString()]
        [ValidateScript({$true})]
        [ValidateRange(0,5)]
        [int]
        $Param2,

        # Param3 help description
        [Parameter(ParameterSetName='Another Parameter Set')]
        [ValidatePattern("[a-z]*")]
        [ValidateLength(0,15)]
        [String]
        $Param3
    )

    Begin
    {

```

```
}  
Process  
{  
    if ($pscmdlet.ShouldProcess("Target", "Operation"))  
    {  
    }  
}  
End  
{  
}  
}
```

Lire Modules, scripts et fonctions en ligne: <https://riptutorial.com/fr/powershell/topic/5755/modules-scripts-et-fonctions>

Chapitre 44: MongoDB

Remarques

La partie la plus difficile est d'attacher un **sous** -document au document qui n'a pas encore été créé si nous avons besoin que le sous-document ait besoin d'être dans le document attendu `$doc2.add("Key", "Value")` au lieu d'utiliser le tableau actuel `foreach` avec `index`. Cela fera le sous-document en deux lignes, comme vous pouvez le voir dans les "Tags" =

```
[MongoDB.Bson.BsonDocument] $doc2 .
```

Exemples

MongoDB avec le pilote C # 1.7 utilisant PowerShell

Je dois interroger tous les détails de la machine virtuelle et les mettre à jour dans la MongoDB.

```
Which require the output look like this.
{
  "_id" : ObjectId("5800509f23888a12bccf2347"),
  "ResourceGrp" : "XYZZ-MachineGrp",
  "ProcessTime" : ISODate("2016-10-14T03:27:16.586Z"),
  "SubscriptionName" : "GSS",
  "OS" : "Windows",
  "HostName" : "VM1",
  "IPAddress" : "192.168.22.11",
  "Tags" : {
    "costCenter" : "803344",
    "BusinessUNIT" : "WinEng",
    "MachineRole" : "App",
    "OwnerEmail" : "zteffer@somewhere.com",
    "appSupporter" : "Steve",
    "environment" : "Prod",
    "implementationOwner" : "xyzr@somewhere.com",
    "appSoftware" : "WebServer",
    "Code" : "Gx",
    "WholeOwner" : "zzzgg@somewhere.com"
  },
  "SubscriptionID" : "",
  "Status" : "running fine",
  "ResourceGroupName" : "XYZZ-MachineGrp",
  "LocalTime" : "14-10-2016-11:27"
}
```

J'ai 3 séries de tableaux dans Powershell

```
$MachinesList # Array
$ResourceList # Array
$MachineTags # Array

pseudo code
```

```

$mongoDriverPath = 'C:\Program Files (x86)\MongoDB\CSharpDriver 1.7';
Add-Type -Path "$($mongoDriverPath)\MongoDB.Bson.dll";
Add-Type -Path "$($mongoDriverPath)\MongoDB.Driver.dll";

$db = [MongoDB.Driver.MongoDatabase]::Create('mongodb://127.0.0.1:2701/RGrpMachines');
[System.Collections.ArrayList]$TagList = $vm.tags
$A1 = $Taglist.key
$A2 = $Taglist.value
foreach ($Machine in $MachinesList)
{
    foreach($Resource in $ResourceList)
    {
        $doc2 = $null
        [MongoDB.Bson.BsonDocument] $doc2 = @{}; #Create a Document here
        for($i = 0; $i -lt $TagList.count; $i++)
        {
            $A1Key = $A1[$i].ToString()
            $A2Value = $A2[$i].toString()
            $doc2.add("$A1Key", "$A2Value")
        }

        [MongoDB.Bson.BsonDocument] $doc = @{
            "_id"= [MongoDB.Bson.ObjectId]::GenerateNewId();
            "ProcessTime"= [MongoDB.Bson.BsonDateTime] $ProcessTime;
            "LocalTime" = "$LocalTime";
            "Tags" = [MongoDB.Bson.BsonDocument] $doc2;
            "ResourceGrp" = "$RGName";
            "HostName"= "$VMName";
            "Status"= "$VMStatus";
            "IPAddress"= "$IPAddress";
            "ResourceGroupName"= "$RGName";
            "SubscriptionName"= "$CurSubName";
            "SubscriptionID"= "$subid";
            "OS"= "$OSType";
        }; #doc loop close

        $collection.Insert($doc);
    }
}

```

Lire MongoDB en ligne: <https://riptutorial.com/fr/powershell/topic/7438/mongodb>

Chapitre 45: Opérateurs spéciaux

Exemples

Opérateur d'expression de tableau

Renvoie l'expression en tant que tableau.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Renvoiera un tableau avec un élément

```
@(Get-ChildItem $env:windir\System32)
```

Renvoie un tableau contenant tous les éléments du dossier (ce qui ne constitue pas un changement de comportement par rapport à l'expression interne).

Appel opération

```
$command = 'Get-ChildItem'  
& $Command
```

Va exécuter `Get-ChildItem`

Opérateur de sourcing

```
.. \myScript.ps1
```

`.. \myScript.ps1` dans la portée actuelle, rendant toutes les fonctions et variables disponibles dans la portée actuelle.

Lire Opérateurs spéciaux en ligne: <https://riptutorial.com/fr/powershell/topic/8981/operateurs-speciaux>

Chapitre 46: Opérations d'ensemble de base

Introduction

Un ensemble est une collection d'éléments qui peuvent être n'importe quoi. Quel que soit l'opérateur sur lequel nous devons travailler, ces *opérateurs* sont en somme les *opérateurs définis* et l'opération est également appelée *opération définie*. L'opération de base comprend l'union, l'intersection ainsi que l'addition, la soustraction, etc.

Syntaxe

- Objet de groupe
- Group-Object -Property <propertyName>
- Objet de groupe -Property <propertyName>, <propertyName2>
- Group-Object -Property <propertyName> -CaseSensitive
- Group-Object -Property <propertyName> -Culture <culture>
- Objet de groupe -Property <ScriptBlock>
- Objet de tri
- Objet de tri -Property <propertyName>
- Objet de tri -Property <ScriptBlock>
- Objet de tri -Property <propertyName>, <propertyName2>
- Objet de tri -Property <propertyObject> -CaseSensitive
- Objet de tri -Property <propertyObject> -Descending
- Sort-Object -Property <propertyObject> -Unique
- Objet de tri -Property <propertyObject> -Culture <culture>

Exemples

Filtrage: Où-Objet / où /?

Filtrer une énumération en utilisant une expression conditionnelle

Synonymes:

```
Where-Object
where
?
```

Exemple:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

Résultats:

```
Aaron
Albert
Alphonse
```

Commande: Sort-Object / sort

Trier une énumération par ordre croissant ou décroissant

Synonymes:

```
Sort-Object
sort
```

En supposant:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

Le tri croissant est la valeur par défaut:

```
$names | Sort-Object
$names | sort
```

```
Aaron
Aaron
Bernie
Charlie
Danny
```

Pour demander un ordre décroissant:

```
$names | Sort-Object -Descending
$names | sort -Descending
```

```
Danny
Charlie
```

Bernie
Aaron
Aaron

Vous pouvez trier en utilisant une expression.

```
$names | Sort-Object { $_.length }
```

Aaron
Aaron
Danny
Bernie
Charlie

Groupement: groupe-objet / groupe

Vous pouvez regrouper une énumération basée sur une expression.

Synonymes:

```
Group-Object  
group
```

Exemples:

```
$names = @( "Aaron", "Albert", "Alphonse", "Bernie", "Charlie", "Danny", "Ernie", "Frank")  
  
$names | Group-Object -Property Length  
$names | group -Property Length
```

Réponse:

Compter	prénom	Groupe
4	5	{Aaron, Danny, Ernie, Frank}
2	6	{Albert, Bernie}
1	8	{Alphonse}
1	7	{Charlie}

Projection: Select-Object / select

La projection d'une énumération vous permet d'extraire des membres spécifiques de chaque objet, d'extraire tous les détails ou de calculer des valeurs pour chaque objet.

Synonymes:

```
Select-Object  
select
```

Sélection d'un sous-ensemble des propriétés:

```
$dir = dir "C:\MyFolder"  
  
$dir | Select-Object Name, FullName, Attributes  
$dir | select Name, FullName, Attributes
```

prénom	Nom complet	Les attributs
Images	C:\MyFolder\Images	Annuaire
data.txt	C:\MyFolder\data.txt	Archiver
source.c	C:\MyFolder\source.c	Archiver

Sélectionner le premier élément et afficher toutes ses propriétés:

```
$d | select -first 1 *
```

PSPath
PSParentPath
PSChildName
PSDrive
PSProvider
PSIsContainer
BaseName
Mode
prénom
Parent
Existe
Racine
Nom complet
Extension

Temps de creation
CreationTimeUtc
LastAccessTime
LastAccessTimeUtc
LastWriteTime
LastWriteTimeUtc
Les attributs

Lire Opérations d'ensemble de base en ligne:

<https://riptutorial.com/fr/powershell/topic/1557/operations-d-ensemble-de-base>

Chapitre 47: Paramètres communs

Remarques

Les paramètres communs peuvent être utilisés avec n'importe quelle applet de commande (cela signifie que dès que vous marquez votre fonction comme une applet de commande [voir `CmdletBinding()`], vous obtenez tous ces paramètres gratuitement).

Voici la liste de tous les paramètres communs (l'alias est entre parenthèses après le paramètre correspondant):

```
-Debug (db)
-ErrorAction (ea)
-ErrorVariable (ev)
-InformationAction (ia) # introduced in v5
-InformationVariable (iv) # introduced in v5
-OutVariable (ov)
-OutBuffer (ob)
-PipelineVariable (pv)
-Verbose (vb)
-WarningAction (wa)
-WarningVariable (wv)
-WhatIf (wi)
-Confirm (cf)
```

Exemples

Paramètre ErrorAction

Les valeurs possibles sont `Continue` | `Ignore` | `Inquire` | `SilentlyContinue` | `Stop` | `Suspend`

La valeur de ce paramètre déterminera comment l'applet de commande gèrera les erreurs non terminantes (celles générées à partir de `Write-Error`, par exemple; pour en savoir plus sur la gestion des erreurs, voir [*rubrique non encore créée*]).

La valeur par défaut (si ce paramètre est omis) est `Continue` .

-ErrorAction Continue

Cette option produira un message d'erreur et continuera avec l'exécution.

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : test
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
Second command
```

-ErrorAction Ignore

Cette option ne produira aucun message d'erreur et continuera avec l'exécution. De plus, aucune erreur ne sera ajoutée à la variable automatique `$Error`.

Cette option a été introduite dans v3.

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Ignore ; Write-Host "Second command"
Second command
```

-ErrorAction Enquête

Cette option produira un message d'erreur et invitera l'utilisateur à choisir une action à entreprendre.

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
Confirm
test
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is "N")
```

-ErrorAction silencieusementContinuer

Cette option ne produira pas de message d'erreur et continuera avec l'exécution. Toutes les erreurs seront ajoutées à la variable automatique `$Error`.

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction SilentlyContinue ; Write-Host "Second command"
Second command
```

-ErrorAction Stop

Cette option produira un message d'erreur et ne continuera pas avec l'exécution.

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
Write-Error "test" -ErrorAction Stop ; Write-Host "Second command" : test
At line:1 char:1
+ Write-Error "test" -ErrorAction Stop ; Write-Host "Second command"
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
```

-ErrorAction Suspend

Disponible uniquement dans les flux de travail Powershell. Lorsqu'il est utilisé, si la commande rencontre une erreur, le workflow est suspendu. Cela permet de rechercher une telle erreur et offre la possibilité de reprendre le workflow. Pour en savoir plus sur le système de workflow, voir [rubrique non encore créée].

Lire Paramètres communs en ligne: <https://riptutorial.com/fr/powershell/topic/5951/parametres-communs>

Chapitre 48: Paramètres dynamiques PowerShell

Exemples

Paramètre dynamique "simple"

Cet exemple ajoute un nouveau paramètre à MyTestFunction si `$SomeUsefulNumber` est supérieur à 5.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)] [int] $SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
        System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
        System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # If "SomeUsefulNumber" is greater than 5, then add the "MandatoryParam1" parameter
        if($SomeUsefulNumber -gt 5)
        {
            # Create a mandatory string parameter called "MandatoryParam1"
            $dynParam1 = New-Object -Type
            System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
            $attributeCollection)
            # Add the new parameter to the dictionary
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Notice that dynamic parameters need a specific syntax
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }
}
```

Usage:

```
PS > MyTestFunction -SomeUsefulNumber 3
```

```

SomeUsefulNumber = 3
MandatoryParam1 =

PS > MyTestFunction -SomeUsefulNumber 6
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:

PS >MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test

```

Dans le deuxième exemple d'utilisation, vous pouvez voir clairement qu'un paramètre est manquant.

Les paramètres dynamiques sont également pris en compte lors de l'achèvement automatique. Voici ce qui se passe si vous appuyez sur ctrl + espace à la fin de la ligne:

```

PS >MyTestFunction -SomeUsefulNumber 3 -<ctrl+space>
Verbose          WarningAction      WarningVariable    OutBuffer
Debug            InformationAction  InformationVariable PipelineVariable
ErrorAction      ErrorVariable      OutVariable

PS >MyTestFunction -SomeUsefulNumber 6 -<ctrl+space>
MandatoryParam1  ErrorAction        ErrorVariable      OutVariable
Verbose          WarningAction      WarningVariable    OutBuffer
Debug            InformationAction  InformationVariable PipelineVariable

```

Lire Paramètres dynamiques PowerShell en ligne:

<https://riptutorial.com/fr/powershell/topic/6704/parametres-dynamiques-powershell>

Chapitre 49: Postes de travail PowerShell

Introduction

Des travaux ont été introduits dans PowerShell 2.0 et ont permis de résoudre un problème inhérent aux outils de ligne de commande. En bref, si vous lancez une longue tâche, votre invite est indisponible jusqu'à la fin de la tâche. À titre d'exemple d'une longue tâche, pensez à cette simple commande PowerShell:

```
Get-ChildItem -Path c: \ -Recurse
```

Il faudra un certain temps pour récupérer la liste complète des répertoires de votre lecteur C :. Si vous l'exécutez en tant que Job, la console récupérera le contrôle et vous pourrez capturer le résultat ultérieurement.

Remarques

Les jobs PowerShell s'exécutent dans un nouveau processus. Cela a des avantages et des inconvénients qui sont liés.

Avantages:

1. Le travail s'exécute dans un processus propre, y compris l'environnement.
2. Le travail peut être exécuté de manière asynchrone sur votre processus PowerShell principal

Les inconvénients:

1. Les modifications de l'environnement de processus ne seront pas présentes dans le travail.
2. Les paramètres transmis et les résultats renvoyés sont sérialisés.
 - Cela signifie que si vous modifiez un objet de paramètre pendant l'exécution du travail, il ne sera pas répercuté dans le travail.
 - Cela signifie également que si un objet ne peut pas être sérialisé, vous ne pouvez ni le transmettre ni le renvoyer (bien que PowerShell puisse copier des paramètres et transmettre / renvoyer un objet PSObject).

Exemples

Création d'emplois de base

Commencez un bloc de script en tâche de fond:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Lancer un script en tâche de fond:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Démarrez un travail en utilisant `Invoke-Command` sur un ordinateur distant:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName "WinRM" -ThrottleLimit 16 -AsJob
```

Démarrer le travail en tant qu'utilisateur différent (invite le mot de passe):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

Ou

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Démarrer le travail en tant qu'utilisateur différent (pas d'invite):

```
$username = "Domain\Username"  
$password = "password"  
$secPassword = ConvertTo-SecureString -String $password -AsPlainText -Force  
$credentials = New-Object System.Management.Automation.PSCredential -ArgumentList @($username,  
$secPassword)  
Start-Job -ScriptBlock {Get-Process} -Credential $credentials
```

Gestion de base

Obtenez une liste de tous les travaux de la session en cours:

```
Get-Job
```

En attente d'un travail pour terminer avant d'obtenir le résultat:

```
$job | Wait-job | Receive-Job
```

Expire un travail s'il est trop long (10 secondes dans cet exemple)

```
$job | Wait-job -Timeout 10
```

Arrêt d'un travail (termine toutes les tâches en attente dans cette file d'attente avant la fin):

```
$job | Stop-Job
```

Supprimer le travail de la liste des travaux en arrière-plan de la session en cours:

```
$job | Remove-Job
```

Remarque : Les éléments suivants ne fonctionnent que sur les travaux de `Workflow` .

Suspendre un job de Workflow (Pause):

```
$job | Suspend-Job
```

Reprendre un job de Workflow :

```
$job | Resume-Job
```

Lire Postes de travail PowerShell en ligne: <https://riptutorial.com/fr/powershell/topic/3970/postes-de-travail-powershell>

Chapitre 50: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output et Information

Remarques

<https://technet.microsoft.com/en-us/library/hh849921.aspx>

Exemples

Sortie d'écriture

`Write-Output` génère une sortie. Cette sortie peut aller à la commande suivante après le pipeline ou à la console pour être simplement affichée.

La cmdlet envoie des objets dans le pipeline principal, également appelé "flux de sortie" ou "pipeline de réussite". Pour envoyer des objets d'erreur dans le pipeline d'erreurs, utilisez `Write-Error`.

```
# 1.) Output to the next Cmdlet in the pipeline
Write-Output 'My text' | Out-File -FilePath "$env:TEMP\Test.txt"

Write-Output 'Bob' | ForEach-Object {
    "My name is $_"
}

# 2.) Output to the console since Write-Output is the last command in the pipeline
Write-Output 'Hello world'

# 3.) 'Write-Output' CmdLet missing, but the output is still considered to be 'Write-Output'
'Hello world'
```

1. La cmdlet `Write-Output` envoie l'objet spécifié dans le pipeline vers la commande suivante.
2. Si la commande est la dernière commande du pipeline, l'objet est affiché dans la console.
3. L'interpréteur PowerShell considère cela comme une écriture en sortie implicite.

Étant donné que le comportement par défaut de `Write-Output` consiste à afficher les objets à la fin d'un pipeline, il n'est généralement pas nécessaire d'utiliser la cmdlet. Par exemple, `Get-Process | Write-Output` est équivalent à `Get-Process .`

Préférences d'écriture

Les messages peuvent être écrits avec;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
```

```
Write-Debug "Debug Message"
Write-Progress "Progress Message"
Write-Warning "Warning Message"
```

Chacun d'eux a une variable de préférence;

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

La variable de préférence contrôle la gestion du message et de l'exécution ultérieure du script.

```
$InformationPreference = "SilentlyContinue"
Write-Information "This message will not be shown and execution continues"

$InformationPreference = "Continue"
Write-Information "This message is shown and execution continues"

$InformationPreference = "Inquire"
Write-Information "This message is shown and execution will optionally continue"

$InformationPreference = "Stop"
Write-Information "This message is shown and execution terminates"
```

La couleur des messages peut être contrôlée pour `Write-Error` en définissant;

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

Des paramètres similaires sont disponibles pour `Write-Verbose`, `Write-Debug` et `Write-Warning`.

Lire PowerShell "Streams"; Debug, Verbose, Warning, Error, Output et Information en ligne:
<https://riptutorial.com/fr/powershell/topic/3255/powershell--streams---debug--verbose--warning--error--output-et-information>

Chapitre 51: Powershell Remoting

Remarques

- [about_remote](#)
- [about_RemoteFAQ](#)
- [about_RemoteDépannage](#)

Exemples

Activation de la communication à distance PowerShell

La communication à distance PowerShell doit d'abord être activée sur le serveur auquel vous souhaitez vous connecter à distance.

```
Enable-PSRemoting -Force
```

Cette commande effectue les opérations suivantes:

- Exécute la cmdlet Set-WSManQuickConfig, qui effectue les tâches suivantes:
- Démarre le service WinRM.
- Définit le type de démarrage du service WinRM sur Automatique.
- Crée un écouteur pour accepter les demandes sur n'importe quelle adresse IP, s'il n'en existe pas déjà une.
- Active une exception de pare-feu pour les communications WS-Management.
- Enregistre les configurations de session Microsoft.PowerShell et Microsoft.PowerShell.Workflow, si elles ne sont pas déjà enregistrées.
- Enregistre la configuration de session Microsoft.PowerShell32 sur des ordinateurs 64 bits, si elle n'est pas déjà enregistrée.
- Active toutes les configurations de session.
- Modifie le descripteur de sécurité de toutes les configurations de session pour autoriser l'accès à distance.
- Redémarre le service WinRM pour que les modifications précédentes prennent effet.

Uniquement pour les environnements autres que les domaines

Pour les serveurs d'un domaine AD, l'authentification à distance PS s'effectue via Kerberos ('Default') ou NTLM ('Negotiate'). Si vous souhaitez autoriser l'accès distant à un serveur autre qu'un domaine, vous avez deux options.

Configurez la communication WSMAN sur HTTPS (qui nécessite la génération de certificats) ou

activez l'authentification de base qui envoie vos informations d'identification sur le câble encodé en base64 (ce qui est fondamentalement le même que du texte brut).

Dans les deux cas, vous devrez ajouter les systèmes distants à votre liste d'hôtes sécurisés WSMAN.

Activation de l'authentification de base

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

Ensuite, sur l'ordinateur que vous souhaitez connecter à *partir*, vous devez lui dire de faire confiance à l'ordinateur que vous vous connectez.

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

Important : vous devez demander à votre client de faire confiance à l'ordinateur adressé comme vous le souhaitez (par exemple, si vous vous connectez via IP, il doit faire confiance à l'adresse IP et non au nom d'hôte).

Connexion à un serveur distant via PowerShell

Utilisation des informations d'identification de votre ordinateur local:

```
Enter-PSSession 192.168.1.1
```

Demander des informations d'identification sur l'ordinateur distant

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Exécuter des commandes sur un ordinateur distant

Une fois que la communication à distance Powershell est activée (Enable-PSRemoting) Vous pouvez exécuter des commandes sur l'ordinateur distant comme ceci:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {  
    Write host "Remote Computer Name: $ENV:ComputerName"  
}
```

La méthode ci-dessus crée une session temporaire et la ferme juste après la fin de la commande ou du scriptblock.

Pour laisser la session ouverte et exécuter d'autres commandes ultérieurement, vous devez

d'abord créer une session distante:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Ensuite, vous pouvez utiliser cette session chaque fois que vous invoquez des commandes sur l'ordinateur distant:

```
Invoke-Command -Session $Session -ScriptBlock {  
    Write host "Remote Computer Name: $ENV:ComputerName"  
}  
  
Invoke-Command -Session $Session -ScriptBlock {  
    Get-Date  
}
```

Si vous devez utiliser des informations d'identification différentes, vous pouvez les ajouter avec le paramètre `-Credential` :

```
$Cred = Get-Credential  
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Avertissement de sérialisation à distance

Remarque:

Il est important de savoir que la communication à distance sérialise les objets PowerShell sur le système distant et les désérialise à la fin de la session à distance, c'est-à-dire qu'ils sont convertis en XML pendant le transport et perdent toutes leurs méthodes.

```
$output = Invoke-Command -Session $Session -ScriptBlock {  
    Get-WmiObject -Class win32_printer  
}  
  
$output | Get-Member -MemberType Method  
  
    TypeName: Deserialized.System.Management.ManagementObject#root\cimv2\Win32_Printer  
  
Name      MemberType Definition  
----      -  
GetType   Method      type GetType()  
ToString  Method      string ToString(), string ToString(string format, System.IFormatProvi...
```

Considérant que vous avez les méthodes sur l'objet PS standard:

```
Get-WmiObject -Class win32_printer | Get-Member -MemberType Method  
  
    TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer  
  
Name      MemberType Definition
```

```

-----
CancelAllJobs      Method      System.Management.ManagementBaseObject  CancelAllJobs ()
GetSecurityDescriptor Method      System.Management.ManagementBaseObject
GetSecurityDescriptor ()
Pause              Method      System.Management.ManagementBaseObject  Pause ()
PrintTestPage     Method      System.Management.ManagementBaseObject  PrintTestPage ()
RenamePrinter     Method      System.Management.ManagementBaseObject
RenamePrinter (System.String NewPrinterName)
Reset             Method      System.Management.ManagementBaseObject  Reset ()
Resume            Method      System.Management.ManagementBaseObject  Resume ()
SetDefaultPrinter Method      System.Management.ManagementBaseObject  SetDefaultPrinter ()
SetPowerState     Method      System.Management.ManagementBaseObject
SetPowerState (System.UInt16 PowerState, System.String Time)
SetSecurityDescriptor Method      System.Management.ManagementBaseObject
SetSecurityDescriptor (System.Management.ManagementObject#Win32_SecurityDescriptor Descriptor)

```

Utilisation des arguments

Pour utiliser des arguments comme paramètres du bloc de script distant, vous pouvez utiliser le paramètre `ArgumentList` d' `Invoke-Command` ou utiliser la syntaxe `$Using: :`

Utiliser `ArgumentList` avec des paramètres non nommés (c'est-à-dire dans l'ordre dans lequel ils sont passés au scriptblock):

```

$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}

```

Utiliser `ArgumentList` avec les paramètres nommés:

```

$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)

    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}

```

Utiliser la syntaxe `$Using: :`

```

$servicesToShow = "service1"

```

```
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

Une bonne pratique pour nettoyer automatiquement les sessions PSSession

Lorsqu'une session distante est créée via l'applet de `New-PSSession`, la session `PSSession` se poursuit jusqu'à la fin de la session PowerShell en cours. Cela signifie que, par défaut, la `PSSession` et toutes les ressources associées continueront à être utilisées jusqu'à la fin de la session PowerShell en cours.

Des `PSSessions` actives `PSSessions` peuvent devenir une charge pour les ressources, en particulier pour les scripts longs ou liés entre eux qui créent des centaines de `PSSessions` dans une seule session PowerShell.

Il est `PSSession` supprimer explicitement chaque `PSSession` après son utilisation. [1]

Le modèle de code suivant utilise `try-catch-finally` pour obtenir ce qui précède, en combinant la gestion des erreurs avec un moyen sécurisé pour s'assurer que toutes les `PSSessions` créées sont supprimées lorsqu'elles sont utilisées:

```
try
{
    $session = New-PSSession -Computername "RemoteMachineName"
    Invoke-Command -Session $session -ScriptBlock {write-host "This is running on
$ENV:ComputerName"}
}
catch
{
    Write-Output "ERROR: $_"
}
finally
{
    if ($session)
    {
        Remove-PSSession $session
    }
}
```

Références: [1] <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession>

Lire Powershell Remoting en ligne: <https://riptutorial.com/fr/powershell/topic/3087/powershell-remoting>

Chapitre 52: Profils Powershell

Remarques

Le fichier de profil est un script powershell qui s'exécute au démarrage de la console powershell. De cette façon, nous pouvons préparer notre environnement à chaque fois que nous commençons une nouvelle session.

Les choses typiques que nous voulons faire au démarrage de PowerShell sont les suivantes:

- importer des modules que nous utilisons souvent (ActiveDirectory, Exchange, une DLL spécifique)
- enregistrement
- changer l'invite
- diagnostic

Plusieurs fichiers de profil et emplacements ont des utilisations différentes et une hiérarchie de l'ordre de démarrage:

Hôte	Utilisateur	Chemin	Ordre de départ	Variable
Tout	Tout	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ profile.ps1	1	\$ profile.AllUsersAllHosts
Tout	Actuel	% USERPROFILE% \ Documents \ WindowsPowerShell \ profile.ps1	3	\$ profile.CurrentUserAllHosts
Console	Tout	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShell_profile.ps1	2	\$ profile.AllUsersCurrentHost
Console	Actuel	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShell_profile.ps1	4	\$ profile.CurrentUserCurrentHost
ISE	Tout	% WINDIR% \ System32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShellISE_profile.ps1	2	\$ profile.AllUsersCurrentHost
ISE	Actuel	% USERPROFILE% \ Documents \ WindowsPowerShell \ Microsoft.PowerShellISE_profile.ps1	4	\$ profile.CurrentUserCurrentHost

Exemples

Créer un profil de base

Un profil PowerShell est utilisé pour charger automatiquement les variables et les fonctions définies par l'utilisateur.

Les profils PowerShell ne sont pas automatiquement créés pour les utilisateurs.

Pour créer un profil PowerShell `C:>New-Item -ItemType File $profile .`

Si vous êtes dans ISE, vous pouvez utiliser l'éditeur intégré `C:>psEdit $profile`

Un moyen simple de commencer avec votre profil personnel pour l'hôte actuel consiste à enregistrer du texte dans le chemin d'accès stocké dans la variable `$profile` .

```
"#Current host, current user" > $profile
```

Toute modification ultérieure du profil peut être effectuée à l'aide de PowerShell ISE, du bloc-notes, du code Visual Studio ou de tout autre éditeur.

La variable `$profile` renvoie le profil utilisateur actuel par défaut de l'hôte actuel, mais vous pouvez accéder au chemin d'accès à la règle machine (tous les utilisateurs) et / ou au profil de tous les hôtes (console, ISE, tiers) en utilisant c'est des propriétés.

```
PS> $PROFILE | Format-List -Force

AllUsersAllHosts      : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
AllUsersCurrentHost  :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1
CurrentUserAllHosts  : C:\Users\user\Documents\WindowsPowerShell\profile.ps1
CurrentUserCurrentHost :
C:\Users\user\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1
Length                : 75

PS> $PROFILE.AllUsersAllHosts
C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1
```

Lire Profils Powershell en ligne: <https://riptutorial.com/fr/powershell/topic/5636/profils-powershell>

Chapitre 53: Propriétés calculées

Introduction

Les propriétés calculées dans Powershell sont des propriétés dérivées personnalisées (calculées). Il permet à l'utilisateur de formater une propriété donnée de la manière qu'il le souhaite. Le calcul (expression) peut être tout à fait possible.

Exemples

Afficher la taille du fichier en Ko - Propriétés calculées

Considérons l'extrait ci-dessous,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

Il génère simplement le contenu du dossier avec les propriétés sélectionnées. Quelque chose comme,

```
Name                CreationTime        Length
----                -
AnotherFile.txt    1/26/2017  2:45:02 PM  546000
filetomove.txt     1/5/2017   2:36:01 PM    5
```

Que faire si je souhaite afficher la taille du fichier en Ko? C'est là que les propriétés calculées sont utiles.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1Kb}}
```

Qui produit,

```
Name                Size_In_KB
----                -
AnotherFile.txt    533.203125
Secondfile.txt    1066.4111328125
```

L' `Expression` est ce qui retient le calcul pour la propriété calculée. Et oui, ça peut être n'importe quoi!

Lire Propriétés calculées en ligne: <https://riptutorial.com/fr/powershell/topic/8913/proprietes-calculées>

Chapitre 54: PSScriptAnalyzer - Analyseur de script PowerShell

Introduction

PSScriptAnalyzer, <https://github.com/PowerShell/PSScriptAnalyzer>, est un vérificateur de code statique pour les modules et les scripts Windows PowerShell. PSScriptAnalyzer vérifie la qualité du code Windows PowerShell en exécutant un ensemble de règles basées sur les meilleures pratiques PowerShell identifiées par l'équipe PowerShell et la communauté. Il génère des résultats de diagnostic (erreurs et avertissements) pour informer les utilisateurs des défauts de code potentiels et suggère des solutions possibles pour les améliorer.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Syntaxe

1. `Get-ScriptAnalyzerRule [-CustomizedRulePath <string[]>] [-Name <string[]>] [-Severity <string[]>] [<CommonParameters>]`
2. `Invoke-ScriptAnalyzer [-Path] <string> [-CustomizedRulePath <string[]>] [-ExcludeRule <string[]>] [-IncludeRule<string[]>] [-Severity <string[]>] [-Recurse] [-SuppressedOnly] [<CommonParameters>]`

Exemples

Analyse de scripts avec les ensembles de règles prédéfinis intégrés

ScriptAnalyzer est livré avec des ensembles de règles prédéfinies intégrées pouvant être utilisées pour analyser les scripts. Ceux-ci incluent: `PSGallery`, `DSC` et `CodeFormatting`. Ils peuvent être exécutés comme suit:

Règles de la Galerie PowerShell

Pour exécuter les règles de la galerie PowerShell, utilisez la commande suivante:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

Règles DSC

Pour exécuter les règles DSC, utilisez la commande suivante:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Règles de formatage du code

Pour exécuter les règles de formatage du code, utilisez la commande suivante:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Analyse des scripts par rapport à chaque règle intégrée

Pour exécuter l'analyseur de script sur un seul fichier de script, exécutez:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

Cela analysera votre script par rapport à chaque règle intégrée. Si votre script est suffisamment volumineux, cela peut entraîner de nombreux avertissements et / ou erreurs.

Pour exécuter l'analyseur de script sur un répertoire entier, spécifiez le dossier contenant les fichiers script, module et DSC à analyser. Spécifiez le paramètre Recurse si vous souhaitez que les sous-répertoires soient recherchés pour les fichiers à analyser.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Liste toutes les règles intégrées

Pour voir toutes les règles intégrées exécutées:

```
Get-ScriptAnalyzerRule
```

Lire [PSScriptAnalyzer - Analyseur de script PowerShell en ligne](https://riptutorial.com/fr/powershell/topic/9619/psscriptanalyzer---analyseur-de-script-powershell):

<https://riptutorial.com/fr/powershell/topic/9619/psscriptanalyzer---analyseur-de-script-powershell>

Chapitre 55: Reconnaissance Amazon Web Services (AWS)

Introduction

Amazon Rekognition est un service qui permet d'ajouter facilement une analyse d'image à vos applications. Avec Reconnaissance, vous pouvez détecter des objets, des scènes et des visages dans les images. Vous pouvez également rechercher et comparer des visages. L'API de Rekognition vous permet d'ajouter rapidement à vos applications une recherche visuelle et une classification d'images sophistiquées basées sur l'apprentissage en profondeur.

Exemples

Détecter les étiquettes d'image avec AWS Rekognition

```
$BucketName = 'trevorrekognition'  
$FileName = 'kitchen.jpg'  
  
New-S3Bucket -BucketName $BucketName  
Write-S3Object -BucketName $BucketName -File $FileName  
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName  
  
$REKResult.Labels
```

Après avoir exécuté le script ci-dessus, vous devriez avoir les résultats imprimés dans votre hôte PowerShell qui ressemblent à ceci:

```
RESULTS:  
  
Confidence Name  
-----  
86.87605    Indoors  
86.87605    Interior Design  
86.87605    Room  
77.4853     Kitchen  
77.25354    Housing  
77.25354    Loft  
66.77325    Appliance  
66.77325    Oven
```

En utilisant le module AWS PowerShell conjointement avec le service AWS Rekognition, vous pouvez détecter des étiquettes dans une image, telles que l'identification des objets dans une pièce, les attributs des photos prises et le niveau de confiance correspondant à chacun de ces attributs.

La commande `Find-REKLabel` est celle qui vous permet d'appeler une recherche pour ces attributs / étiquettes. Bien que vous puissiez fournir un contenu image sous la forme d'un tableau d'octets lors de l'appel d'API, une meilleure méthode consiste à télécharger vos fichiers image dans un

compartiment AWS S3, puis à diriger le service Reconnaissance sur les objets S3 à analyser. L'exemple ci-dessus montre comment accomplir cela.

Comparer la similarité faciale avec AWS Rekognition

```
$BucketName = 'trevorrekognition'

### Create a new AWS S3 Bucket
New-S3Bucket -BucketName $BucketName

### Upload two different photos of myself to AWS S3 Bucket
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg

### Perform a facial comparison between the two photos with AWS Rekognition
$Comparison = @{
    SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

L'exemple de script fourni ci-dessus devrait vous donner des résultats similaires aux suivants:

```
Face                                     Similarity
----                                     -
Amazon.Rekognition.Model.ComparedFace 90
```

Le service AWS Rekognition vous permet d'effectuer une comparaison faciale entre deux photos. L'utilisation de ce service est assez simple. Il vous suffit de télécharger deux fichiers image que vous souhaitez comparer sur un compartiment AWS S3. Ensuite, `Compare-REKFace` commande `Compare-REKFace`, similaire à l'exemple ci-dessus. Bien sûr, vous devrez fournir vos propres noms et noms de fichiers S3 Bucket, uniques au monde.

Lire Reconnaissance Amazon Web Services (AWS) en ligne:

<https://riptutorial.com/fr/powershell/topic/9581/reconnaissance-amazon-web-services--aws->

Chapitre 56: requêtes sql powershell

Introduction

En parcourant ce document, vous pouvez apprendre à utiliser les requêtes SQL avec powershell

Paramètres

Article	La description
\$ ServerInstance	Nous devons mentionner ici l'instance dans laquelle la base de données est présente
\$ Base de données	Il faut mentionner ici la base de données dans laquelle la table est présente
\$ Query	Nous devons ici à la requête que vous voulez exécuter dans SQ
\$ Username & \$ Password	UserName et Password qui ont accès à la base de données

Remarques

Vous pouvez utiliser la fonction ci-dessous si vous ne parvenez pas à importer le module SQLPS

```
function Import-Xls
{
    [CmdletBinding(SupportsShouldProcess=$true)]

    Param(
        [parameter(
            mandatory=$true,
            position=1,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true)]
        [String[]]
        $Path,

        [parameter(mandatory=$false)]
        $Worksheet = 1,

        [parameter(mandatory=$false)]
        [switch]
        $Force
    )

    Begin
    {
        function GetTempFileName($extension)
    }
```

```

{
    $temp = [io.path]::GetTempFileName();
    $params = @{
        Path = $temp;
        Destination = $temp + $extension;
        Confirm = $false;
        Verbose = $VerbosePreference;
    }
    Move-Item @params;
    $temp += $extension;
    return $temp;
}

# since an extension like .xls can have multiple formats, this
# will need to be changed
#
$xmlFileFormats = @{
    # single worksheet formats
    '.csv' = 6;          # 6, 22, 23, 24
    '.dbf' = 11;         # 7, 8, 11
    '.dif' = 9;          #
    '.prn' = 36;         #
    '.slk' = 2;          # 2, 10
    '.wk1' = 31;         # 5, 30, 31
    '.wk3' = 32;         # 15, 32
    '.wk4' = 38;         #
    '.wks' = 4;          #
    '.xlw' = 35;         #

    # multiple worksheet formats
    '.xls' = -4143;      # -4143, 1, 16, 18, 29, 33, 39, 43
    '.xlsb' = 50;        #
    '.xlsm' = 52;        #
    '.xlsx' = 51;        #
    '.xml' = 46;         #
    '.ods' = 60;         #
}

$xml = New-Object -ComObject Excel.Application;
$xml.DisplayAlerts = $false;
$xml.Visible = $false;
}

Process
{
    $Path | ForEach-Object {

        if ($Force -or $psCmdlet.ShouldProcess($_)) {

            $fileExist = Test-Path $_

            if (-not $fileExist) {
                Write-Error "Error: $_ does not exist" -Category ResourceUnavailable;
            } else {
                # create temporary .csv file from excel file and import .csv
                #
                $_ = (Resolve-Path $_).ToString();
                $wb = $xml.Workbooks.Add($_);
                if ($?) {
                    $csvTemp = GetTempFileName(".csv");
                }
            }
        }
    }
}

```


\$ Password = "Mot de passe"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password $Password
```

Lire requêtes sql powershell en ligne: <https://riptutorial.com/fr/powershell/topic/8217/requetes-sql-powershell>

Chapitre 57: Scripts de signature

Remarques

La signature d'un script fera en sorte que vos scripts soient conformes à toutes les stratégies d'exécution dans PowerShell et garantissent l'intégrité d'un script. Les scripts signés ne seront pas exécutés s'ils ont été modifiés après leur signature.

La signature de scripts nécessite un certificat de signature de code. Recommandations:

- Scripts / tests personnels (non partagés): Certificat d'une autorité de certification certifiée (interne ou tierce partie) **OU** un certificat auto-signé.
- Organisation interne partagée: Certificat de l'autorité de certification certifiée (interne ou tierce)
- Organisation externe partagée: Certificat émanant d'une autorité de certification tierce de confiance

En savoir plus sur [about_Signing @ TechNet](#)

Politiques d'exécution

PowerShell dispose de stratégies d'exécution configurables qui contrôlent les conditions requises pour l'exécution d'un script ou d'une configuration. Une politique d'exécution peut être définie pour plusieurs portées. ordinateur, utilisateur actuel et processus en cours. **Les stratégies d'exécution peuvent facilement être ignorées et ne sont pas conçues pour restreindre les utilisateurs, mais plutôt pour les protéger contre la violation involontaire de stratégies de signature.**

Les politiques disponibles sont les suivantes:

Réglage	La description
Limité	Aucun script autorisé
AllSigned	Tous les scripts doivent être signés
RemoteSigned	Tous les scripts locaux autorisés; uniquement des scripts distants signés
Libre	Aucune exigence Tous les scripts sont autorisés, mais avertiront avant d'exécuter des scripts téléchargés à partir d'Internet
Contourne	Tous les scripts sont autorisés et aucun avertissement n'est affiché
Indéfini	Supprimez la stratégie d'exécution en cours pour l'étendue actuelle. Utilisez la politique parent. Si toutes les règles sont indéfinies, la restriction sera utilisée.

Vous pouvez modifier les stratégies d'exécution en cours à l'aide du `Set-ExecutionPolicy` -cmdlet, Group Policy ou `-ExecutionPolicy` lors du lancement d'un processus `powershell.exe` .

En savoir plus sur [about_Execution_Policies @ TechNet](#)

Exemples

Signer un script

La signature d'un script s'effectue à l'aide de la cmdlet `Set-AuthenticodeSignature` et d'un certificat de signature de code.

```
#Get the first available personal code-signing certificate for the logged on user
$cert = @(Get-ChildItem -Path Cert:\CurrentUser\My -CodeSigningCert)[0]

#Sign script using certificate
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1
```

Vous pouvez également lire un certificat à partir d'un `.pfx` `.pfx` en utilisant:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

Le script sera valide jusqu'à l'expiration du certificat. Si vous utilisez un serveur d'horodatage lors de la signature, le script continuera à être valide après l'expiration du certificat. Il est également utile d'ajouter la chaîne d'approbation du certificat (y compris l'autorité racine) pour aider la plupart des ordinateurs à faire confiance au certificat utilisé pour signer le script.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timestamp.dll"
```

Il est recommandé d'utiliser un serveur d'horodatage auprès d'un fournisseur de certificats de confiance tel que Verisign, Comodo, Thawte, etc.

Modification de la stratégie d'exécution à l'aide de `Set-ExecutionPolicy`

Pour modifier la stratégie d'exécution de la portée par défaut (LocalMachine), utilisez:

```
Set-ExecutionPolicy AllSigned
```

Pour modifier la stratégie pour une portée spécifique, utilisez:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

Vous pouvez supprimer les invites en ajoutant le commutateur `-Force` .

Contournement de la politique d'exécution pour un seul script

Vous devrez souvent exécuter un script non signé qui ne respecte pas la stratégie d'exécution en cours. Un moyen simple de le faire est de contourner la stratégie d'exécution pour ce processus unique. Exemple:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

Ou vous pouvez utiliser la sténographie:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Autres politiques d'exécution:

Politique	La description
AllSigned	Seuls les scripts signés par un éditeur approuvé peuvent être exécutés.
Bypass	Pas de restrictions; tous les scripts Windows PowerShell peuvent être exécutés.
Default	Normalement RemoteSigned , mais est contrôlé via ActiveDirectory
RemoteSigned	Les scripts téléchargés doivent être signés par un éditeur approuvé avant de pouvoir être exécutés.
Restricted	Aucun script ne peut être exécuté. Windows PowerShell ne peut être utilisé qu'en mode interactif.
Undefined	N / A
Unrestricted*	Similaire à bypass

Unrestricted* Avertissement: Si vous exécutez un script non signé qui a été téléchargé depuis Internet, vous êtes invité à en demander l'autorisation avant de l'exécuter.

Plus d'informations disponibles [ici](#) .

Obtenir la politique d'exécution actuelle

Obtenir la politique d'exécution effective pour la session en cours:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Énumérer toutes les politiques d'exécution efficaces pour la session en cours:

```
PS> Get-ExecutionPolicy -List
```

```
Scope ExecutionPolicy
-----
MachinePolicy          Undefined
UserPolicy             Undefined
Process               Undefined
CurrentUser            Undefined
LocalMachine           RemoteSigned
```

Énumérer la politique d'exécution pour une portée spécifique, ex. processus:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Obtenir la signature d'un script signé

Obtenez des informations sur la signature Authenticode à partir d'un script signé à l'aide de la cmdlet `Get-AuthenticodeSignature` :

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Création d'un certificat de signature de code auto-signé pour le test

Lors de la signature de scripts personnels ou lors du test de signature de code, il peut être utile de créer un certificat de signature de code auto-signé.

5.0

À partir de PowerShell 5.0, vous pouvez générer un certificat de signature de code auto-signé à l'aide de la `New-SelfSignedCertificate` `SelfSignedCertificate`:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -
CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

Dans les versions antérieures, vous pouvez créer un certificat auto-signé à l'aide de l'outil `makecert.exe` situé dans le Kit de développement .NET Framework SDK et Windows SDK.

Un certificat auto-signé ne sera approuvé que par les ordinateurs sur lesquels le certificat a été installé. Pour les scripts qui seront partagés, un certificat provenant d'une autorité de certification approuvée (interne ou tiers de confiance) est recommandé.

Lire Scripts de signature en ligne: <https://riptutorial.com/fr/powershell/topic/5670/scripts-de-signature>

Chapitre 58: Sécurité et cryptographie

Exemples

Calcul des codes de hachage d'une chaîne via la cryptographie .Net

Utilisation de l'espace de noms .Net `System.Security.Cryptography.HashAlgorithm` pour générer le code de hachage des messages avec les algorithmes pris en charge.

```
$example="Nobody expects the Spanish Inquisition."

#calculate
$hash=[System.Security.Cryptography.HashAlgorithm]::Create("sha256").ComputeHash(
[System.Text.Encoding]::UTF8.GetBytes($example))

#convert to hex
[System.BitConverter]::ToString($hash)

#2E-DF-DA-DA-56-52-5B-12-90-FF-16-FB-17-44-CF-B4-82-DD-29-14-FF-BC-B6-49-79-0C-0E-58-9E-46-2D-
3D
```

La partie "sha256" était l'algorithme de hachage utilisé.

le - peut être supprimé ou changer en minuscule

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-", "").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbc649790c0e589e462d3d
```

Si le format base64 était préféré, utiliser le convertisseur base64 pour la sortie

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Lire Sécurité et cryptographie en ligne: <https://riptutorial.com/fr/powershell/topic/5683/securite-et-cryptographie>

Chapitre 59: Service de stockage simple Amazon Web Services (AWS) (S3)

Introduction

Cette section de documentation traite du développement par rapport au service de stockage simple (S3) Amazon Web Services (AWS). S3 est vraiment un service simple avec lequel interagir. Vous créez des "buckets" S3 pouvant contenir zéro ou plusieurs "objets". Une fois que vous créez un compartiment, vous pouvez télécharger des fichiers ou des données arbitraires dans le compartiment S3 en tant qu'"objet". Vous faites référence aux objets S3, à l'intérieur d'un compartiment, par la "clé" (nom) de l'objet.

Paramètres

Paramètre	Détails
BucketName	Le nom du compartiment AWS S3 sur lequel vous travaillez.
CannedACLName	Nom de la liste de contrôle d'accès (ACL) intégrée (prédéfinie) qui sera associée au compartiment S3.
Fichier	Nom d'un fichier sur le système de fichiers local qui sera téléchargé sur un compartiment AWS S3.

Exemples

Créer un nouveau seau S3

```
New-S3Bucket -BucketName trevor
```

Le nom du compartiment du service de stockage simple (S3) doit être unique au monde. Cela signifie que si quelqu'un d'autre a déjà utilisé le nom du compartiment que vous souhaitez utiliser, vous devez choisir un nouveau nom.

Télécharger un fichier local dans un compartiment S3

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'  
Write-S3Object -BucketName powershell -File myfile.txt
```

Le téléchargement de fichiers depuis votre système de fichiers local dans AWS S3 est simple, à l'aide de la commande `Write-S3Object`. Dans sa forme la plus élémentaire, il vous suffit de spécifier le paramètre `-BucketName` pour indiquer le `-BucketName` S3 dans lequel vous souhaitez

charger un fichier et le paramètre `-File` , qui indique le chemin relatif ou absolu du fichier local que vous souhaitez télécharger dans le compartiment S3.

Supprimer un seau S3

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force  
Remove-S3Bucket -BucketName powershell -Force
```

Pour supprimer un compartiment S3, vous devez d'abord supprimer tous les objets S3 stockés à l'intérieur du compartiment, à condition que vous y soyez autorisé. Dans l'exemple ci-dessus, nous récupérons une liste de tous les objets à l'intérieur d'un compartiment, puis nous les `Remove-S3Object` commande `Remove-S3Object` pour les supprimer. Une fois que tous les objets ont été supprimés, nous pouvons utiliser la commande `Remove-S3Bucket` pour supprimer le `Remove-S3Bucket` .

Lire [Service de stockage simple Amazon Web Services \(AWS\) \(S3\) en ligne:](https://riptutorial.com/fr/powershell/topic/9579/service-de-stockage-simple-amazon-web-services--aws---s3-)

<https://riptutorial.com/fr/powershell/topic/9579/service-de-stockage-simple-amazon-web-services--aws---s3->

Chapitre 60: Splatting

Introduction

La splatting est une méthode de transmission de plusieurs paramètres à une commande en une seule unité. Cela se fait en stockant les paramètres et leurs valeurs en tant que paires clé-valeur dans une [table de hachage](#) et en les répartissant dans une applet de commande à l'aide de l'opérateur de répartition @ .

La fragmentation peut rendre une commande plus lisible et vous permet de réutiliser des paramètres dans des appels de commande multiples.

Remarques

Remarque: L' [opérateur d'expression Array](#) ou @() a un comportement très différent de celui de l'opérateur Splatting @ .

En savoir plus sur [about_Splatting @ TechNet](#)

Exemples

Paramètres de splatting

La répartition est effectuée en remplaçant le signe dollar \$ par l'opérateur de répartition @ lors de l'utilisation d'une variable contenant une [table de hachage](#) contenant des paramètres et des valeurs dans un appel de commande.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Sans splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

Vous pouvez combiner des paramètres normaux avec des paramètres splatted pour ajouter facilement des paramètres communs à vos appels.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters
```

```
Invoke-Command -ScriptBlock { "Something to excute remotely" } @MyParameters
```

Passage d'un paramètre Switch à l'aide de la division

Pour utiliser Splatting pour appeler `Get-Process` avec le commutateur `-FileVersionInfo` similaire à ceci:

```
Get-Process -FileVersionInfo
```

Ceci est l'appel utilisant le splatting:

```
$MyParameters = @{  
    FileVersionInfo = $true  
}  
  
Get-Process @MyParameters
```

Note: Ceci est utile car vous pouvez créer un jeu de paramètres par défaut et passer plusieurs fois l'appel comme ceci

```
$MyParameters = @{  
    FileVersionInfo = $true  
}  
  
Get-Process @MyParameters -Name WmiPrvSE  
Get-Process @MyParameters -Name explorer
```

Tuyauterie et éclaboussure

La déclaration de splat est utile pour réutiliser des ensembles de paramètres plusieurs fois ou avec de légères variations:

```
$splat = @{  
    Class = "Win32_SystemEnclosure"  
    Property = "Manufacturer"  
    ErrorAction = "Stop"  
}  
  
Get-WmiObject -ComputerName $env:COMPUTERNAME @splat  
Get-WmiObject -ComputerName "Computer2" @splat  
Get-WmiObject -ComputerName "Computer3" @splat
```

Cependant, si le splat n'est pas indenté pour être réutilisé, vous ne souhaitez peut-être pas le déclarer. Il peut être canalisé à la place:

```
@{  
    ComputerName = $env:COMPUTERNAME  
    Class = "Win32_SystemEnclosure"  
    Property = "Manufacturer"  
    ErrorAction = "Stop"  
} | % { Get-WmiObject @_ }
```

Division de la fonction de niveau supérieur en une série de fonctions internes

Sans splatting, il est très difficile d'essayer de transmettre des valeurs à travers la pile d'appels. Mais si vous combinez la répartition avec la puissance de **@PSBoundParameters**, vous pouvez transmettre la collection de paramètres de niveau supérieur aux couches.

```
Function Outer-Method
{
    Param
    (
        [string]
        $First,

        [string]
        $Second
    )

    Write-Host ($First) -NoNewline

    Inner-Method @PSBoundParameters
}

Function Inner-Method
{
    Param
    (
        [string]
        $Second
    )

    Write-Host (" {0}!" -f $Second)
}

$parameters = @{
    First = "Hello"
    Second = "World"
}

Outer-Method @parameters
```

Lire Splatting en ligne: <https://riptutorial.com/fr/powershell/topic/5647/splatting>

Chapitre 61: Travailler avec des fichiers XML

Exemples

Accéder à un fichier XML

```
<!-- file.xml -->
<people>
  <person id="101">
    <name>Jon Lajoie</name>
    <age>22</age>
  </person>
  <person id="102">
    <name>Lord Gaben</name>
    <age>65</age>
  </person>
  <person id="103">
    <name>Gordon Freeman</name>
    <age>29</age>
  </person>
</people>
```

Chargement d'un fichier XML

Pour charger un fichier XML, vous pouvez utiliser l'un de ces éléments:

```
# First Method
$xml = New-Object System.Xml.XmlDocument
$file = Resolve-Path(".\file.xml")
$xml.load($file)

# Second Method
[xml] $xml = Get-Content ".\file.xml"

# Third Method
$xml = [xml] (Get-Content ".\file.xml")
```

Accéder à XML en tant qu'objets

```
PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.people

person
-----
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.people.person

id          name          age
--          ----          ---
```

```

101                Jon Lajoie                22
102                Lord Gaben                65
103                Gordon Freeman            29

PS C:\> $xml.people.person[0].name
Jon Lajoie

PS C:\> $xml.people.person[1].age
65

PS C:\> $xml.people.person[2].id
103

```

Accéder à XML avec XPath

```

PS C:\> $xml = [xml](Get-Content file.xml)
PS C:\> $xml

PS C:\> $xml.SelectNodes("//people")

person
-----
{Jon Lajoie, Lord Gaben, Gordon Freeman}

PS C:\> $xml.SelectNodes("//people//person")

id                name                age
--                ----                ---
101                Jon Lajoie                22
102                Lord Gaben                65
103                Gordon Freeman            29

PS C:\> $xml.SelectSingleNode("people//person[1]//name")
Jon Lajoie

PS C:\> $xml.SelectSingleNode("people//person[2]//age")
65

PS C:\> $xml.SelectSingleNode("people//person[3]//@id")
103

```

Accéder à XML contenant des espaces de noms avec XPath

```

PS C:\> [xml]$xml = @"
<ns:people xmlns:ns="http://schemas.xmlsoap.org/soap/envelope/">
  <ns:person id="101">
    <ns:name>Jon Lajoie</ns:name>
  </ns:person>
  <ns:person id="102">
    <ns:name>Lord Gaben</ns:name>
  </ns:person>
  <ns:person id="103">
    <ns:name>Gordon Freeman</ns:name>
  </ns:person>
</ns:people>
"@

PS C:\> $ns = new-object Xml.XmlNamespaceManager $xml.NameTable

```

```
PS C:\> $ns.AddNamespace("ns", $xml.DocumentElement.NamespaceURI)
PS C:\> $xml.SelectNodes("//ns:people/ns:person", $ns)
```

id	name
--	----
101	Jon Lajoie
102	Lord Gaben
103	Gordon Freeman

Création d'un document XML à l'aide de XmlWriter ()

```
# Set The Formatting
$xmlsettings = New-Object System.Xml.XmlWriterSettings
$xmlsettings.Indent = $true
$xmlsettings.IndentChars = "    "

# Set the File Name Create The Document
$xmlWriter = [System.Xml.XmlWriter]::Create("C:\YourXML.xml", $xmlsettings)

# Write the XML Declaration and set the XSL
$xmlWriter.WriteStartDocument()
$xmlWriter.WriteProcessingInstruction("xml-stylesheet", "type='text/xsl' href='style.xsl'")

# Start the Root Element
$xmlWriter.WriteStartElement("Root")

    $xmlWriter.WriteStartElement("Object") # <-- Start <Object>

        $xmlWriter.WriteElementString("Property1", "Value 1")
        $xmlWriter.WriteElementString("Property2", "Value 2")

        $xmlWriter.WriteStartElement("SubObject") # <-- Start <SubObject>
            $xmlWriter.WriteElementString("Property3", "Value 3")
        $xmlWriter.WriteEndElement() # <-- End <SubObject>

    $xmlWriter.WriteEndElement() # <-- End <Object>

$xmlWriter.WriteEndElement() # <-- End <Root>

# End, Finalize and close the XML Document
$xmlWriter.WriteEndDocument()
$xmlWriter.Flush()
$xmlWriter.Close()
```

Fichier XML de sortie

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type='text/xsl' href='style.xsl'?>
<Root>
  <Object>
    <Property1>Value 1</Property1>
    <Property2>Value 2</Property2>
    <SubObject>
      <Property3>Value 3</Property3>
    </SubObject>
  </Object>
</Root>
```

Données d'échantillon

Document XML

Tout d'abord, définissons un exemple de document XML nommé " **books.xml** " dans notre répertoire actuel:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book>
    <title>Of Mice And Men</title>
    <author>John Steinbeck</author>
    <pageCount>187</pageCount>
    <publishers>
      <publisher>
        <isbn>978-88-58702-15-4</isbn>
        <name>Pascal Covici</name>
        <year>1937</year>
        <binding>Hardcover</binding>
        <first>true</first>
      </publisher>
      <publisher>
        <isbn>978-05-82461-46-8</isbn>
        <name>Longman</name>
        <year>2009</year>
        <binding>Hardcover</binding>
      </publisher>
    </publishers>
    <characters>
      <character name="Lennie Small" />
      <character name="Curley's Wife" />
      <character name="George Milton" />
      <character name="Curley" />
    </characters>
    <film>True</film>
  </book>
  <book>
    <title>The Hunt for Red October</title>
    <author>Tom Clancy</author>
    <pageCount>387</pageCount>
    <publishers>
      <publisher>
        <isbn>978-08-70212-85-7</isbn>
        <name>Naval Institute Press</name>
        <year>1984</year>
        <binding>Hardcover</binding>
        <first>true</first>
      </publisher>
      <publisher>
        <isbn>978-04-25083-83-3</isbn>
        <name>Berkley</name>
        <year>1986</year>
        <binding>Paperback</binding>
      </publisher>
    </publishers>
  </book>
</books>
```

```

    <publisher>
      <isbn>978-08-08587-35-4</isbn>
      <name>Penguin Putnam</name>
      <year>2010</year>
      <binding>Paperback</binding>
    </publisher>
  </publishers>
  <characters>
    <character name="Marko Alexadrovich Ramius" />
    <character name="Jack Ryan" />
    <character name="Admiral Greer" />
    <character name="Bart Mancuso" />
    <character name="Vasily Borodin" />
  </characters>
  <film>True</film>
</book>
</books>

```

Nouvelles données

Ce que nous voulons faire, c'est ajouter quelques nouveaux livres à ce document, disons *Patriot Games* de Tom Clancy (oui, je suis un fan des œuvres de Clancy ^ __ ^) et un favori de science-fiction: *The Hitchhiker's Guide to the Galaxy*. par Douglas Adams principalement parce que Zaphod Beeblebrox est juste amusant à lire.

D'une manière ou d'une autre, nous avons acquis les données pour les nouveaux livres et les avons enregistrées en tant que liste de PSCustomObjects:

```

$newBooks = @(
  [PSCustomObject] @{
    "Title" = "Patriot Games";
    "Author" = "Tom Clancy";
    "PageCount" = 540;
    "Publishers" = @(
      [PSCustomObject] @{
        "ISBN" = "978-0-39-913241-4";
        "Year" = "1987";
        "First" = $True;
        "Name" = "Putnam";
        "Binding" = "Hardcover";
      }
    );
    "Characters" = @(
      "Jack Ryan", "Prince of Wales", "Princess of Wales",
      "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
    );
    "film" = $True;
  },
  [PSCustomObject] @{
    "Title" = "The Hitchhiker's Guide to the Galaxy";
    "Author" = "Douglas Adams";
    "PageCount" = 216;
    "Publishers" = @(
      [PSCustomObject] @{
        "ISBN" = "978-0-33-025864-7";
        "Year" = "1979";
        "First" = $True;

```

```

        "Name" = "Pan Books";
        "Binding" = "Hardcover";
    }
};
"Characters" = @(
    "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
    "Trillian", "Slartibartfast", "Dirk Gently"
);
"film" = $True;
}
);

```

Modèles

Maintenant, nous devons définir quelques structures XML squelettes pour nos nouvelles données. Fondamentalement, vous voulez créer un squelette / modèle pour chaque liste de données. Dans notre exemple, cela signifie que nous avons besoin d'un modèle pour le livre, les personnages et les éditeurs. Nous pouvons également l'utiliser pour définir quelques valeurs par défaut, telles que la valeur du tag `film`.

```

$t_book = [xml] @"
<book>
  <title />
  <author />
  <pageCount />
  <publishers />
  <characters />
  <film>False</film>
</book>
"@;

$t_publisher = [xml] @"
<publisher>
  <isbn/>
  <name/>
  <year/>
  <binding/>
  <first>false</first>
</publisher>
"@;

$t_character = [xml] @"
<character name="" />
"@;

```

Nous en avons fini avec l'installation.

Ajouter les nouvelles données

Maintenant que nous sommes tous configurés avec nos exemples de données, ajoutons les objets personnalisés à l'objet Document XML.

```

# Read the xml document
$xml = [xml] Get-Content .\books.xml;

# Let's show a list of titles to see what we've got currently:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Outputs:
# title                author                ISBN
# -----
# Of Mice And Men      John Steinbeck 978-88-58702-15-4
# The Hunt for Red October Tom Clancy      978-08-70212-85-7

# Let's show our new books as well:
$newBooks | Select Title, Author, @{N="ISBN";E={$_Publishers[0].ISBN}};

# Outputs:
# Title                Author                ISBN
# -----
# Patriot Games        Tom Clancy            978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams 978-0-33-025864-7

# Now to merge the two:

ForEach ( $book in $newBooks ) {
    $root = $xml.SelectSingleNode("/books");

    # Add the template for a book as a new node to the root element
    [void]$root.AppendChild($xml.ImportNode($t_book.book, $true));

    # Select the new child element
    $newElement = $root.SelectSingleNode("book[last()]");

    # Update the parameters of that new element to match our current new book data
    $newElement.title      = [String]$book.Title;
    $newElement.author     = [String]$book.Author;
    $newElement.pageCount  = [String]$book.PageCount;
    $newElement.film       = [String]$book.Film;

    # Iterate through the properties that are Children of our new Element:
    ForEach ( $publisher in $book.Publishers ) {
        # Create the new child publisher element
        # Note the use of "SelectSingleNode" here, this allows the use of the "AppendChild"
method as it returns
        # a XmlElement type object instead of the $Null data that is currently stored in that
leaf of the
        # XML document tree

[void]$newElement.SelectSingleNode("publishers").AppendChild($xml.ImportNode($t_publisher.publisher,
$true));

        # Update the attribute and text values of our new XML Element to match our new data
        $newPublisherElement = $newElement.SelectSingleNode("publishers/publisher[last()");
        $newPublisherElement.year = [String]$publisher.Year;
        $newPublisherElement.name = [String]$publisher.Name;
        $newPublisherElement.binding = [String]$publisher.Binding;
        $newPublisherElement.isbn = [String]$publisher.ISBN;
        If ( $publisher.first ) {
            $newPublisherElement.first = "True";
        }
    }
}

```

```

ForEach ( $character in $book.Characters ) {
    # Select the characters xml element
    $charactersElement = $newElement.SelectSingleNode("characters");

    # Add a new character child element
    [void]$charactersElement.AppendChild($xml.ImportNode($t_character.character, $true));

    # Select the new characters/character element
    $characterElement = $charactersElement.SelectSingleNode("character[last()]");

    # Update the attribute and text values to match our new data
    $characterElement.name = [String]$character;
}
}

# Check out the new XML:
$xml.books.book | Select Title, Author, @{N="ISBN";E={If ( $_.Publishers.Publisher.Count ) {
$_Publishers.publisher[0].ISBN} Else { $_.Publishers.publisher.isbn}}};

# Outputs:
# title                author                ISBN
# -----
# Of Mice And Men      John Steinbeck       978-88-58702-15-4
# The Hunt for Red October Tom Clancy           978-08-70212-85-7
# Patriot Games        Tom Clancy           978-0-39-913241-4
# The Hitchhiker's Guide to the Galaxy Douglas Adams        978-0-33-025864-7

```

Nous pouvons maintenant écrire notre XML sur un disque, un écran, un site Web ou n'importe où!

Profit

Bien que cela puisse ne pas être la procédure pour tout le monde, je l'ai trouvée pour éviter tout un tas de

```
[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementName", $textValue));
suivi de $xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue
```

Je pense que la méthode détaillée dans l'exemple est plus propre et plus facile à analyser pour les humains normaux.

Améliorations

Il est possible de modifier le modèle pour inclure des éléments avec des enfants au lieu de séparer chaque section en un modèle distinct. Vous devez juste faire attention à cloner l'élément précédent lorsque vous parcourez la liste.

Lire Travailler avec des fichiers XML en ligne:

<https://riptutorial.com/fr/powershell/topic/4882/travailler-avec-des-fichiers-xml>

Chapitre 62: Travailler avec des objets

Exemples

Mise à jour des objets

Ajout de propriétés

Si vous souhaitez ajouter des propriétés à un objet existant, vous pouvez utiliser l'applet de commande `Add-Member`. Avec `PSObjects`, les valeurs sont conservées dans un type de "Propriétés de la note"

```
$object = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

Add-Member -InputObject $object -Name "SomeNewProp" -Value "A value" -MemberType NoteProperty

# Returns
PS> $Object
Name ID Address SomeNewProp
---- - - - - -
nem 12          A value
```

Vous pouvez également ajouter des propriétés avec la cmdlet `Select-Object` (propriétés dites calculées):

```
$newObject = $Object | Select-Object *, @ {label='SomeOtherProp'; expression={'Another value'}}

# Returns
PS> $newObject
Name ID Address SomeNewProp SomeOtherProp
---- - - - - -
nem 12          A value      Another value
```

La commande ci-dessus peut être raccourcie à ceci:

```
$newObject = $Object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Suppression de propriétés

Vous pouvez utiliser la cmdlet `Select-Object` pour supprimer des propriétés d'un objet:

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address
```

```
# Returns
PS> $Object
Name SomeNewProp SomeOtherProp
---- -
nem  A value      Another value
```

Créer un nouvel objet

PowerShell, contrairement à d'autres langages de script, envoie des objets via le pipeline. Cela signifie que lorsque vous envoyez des données d'une commande à une autre, il est essentiel de pouvoir créer, modifier et collecter des objets.

Créer un objet est simple. La plupart des objets que vous créez seront des objets personnalisés dans PowerShell et le type à utiliser pour cela est `PSObject`. PowerShell vous permettra également de créer tout objet que vous pourriez créer dans `.NET`.

Voici un exemple de création de nouveaux objets avec quelques propriétés:

Option 1: Nouvel objet

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -
nem  12
```

Vous pouvez stocker l'objet dans une variable en faisant précéder la commande de `$newObject =`

Vous devrez peut-être également stocker des collections d'objets. Cela peut être fait en créant une variable de collection vide et en ajoutant des objets à la collection, comme ceci:

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

Vous pouvez ensuite vouloir parcourir cet objet de collection par objet. Pour ce faire, recherchez la section Boucle dans la documentation.

Option 2: Sélectionner un objet

Une manière moins courante de créer des objets que vous trouverez encore sur Internet est la suivante:

```
$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12

# Returns
PS> $newObject
Name ID Address
---- - - -
nem  12
```

Option 3: accélérateur de type pscustomobject (PSv3 + requis)

L'accélérateur de type ordonné oblige PowerShell à conserver nos propriétés dans l'ordre que nous avons défini. Vous n'avez pas besoin de l'accélérateur de type ordonné pour utiliser [PSCustomObject] :

```
$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- - - -
nem  12
```

Examiner un objet

Maintenant que vous avez un objet, il peut être intéressant de le comprendre. Vous pouvez utiliser l'applet de commande Get-Member pour voir ce qu'est un objet et ce qu'il contient:

```
Get-Item c:\windows | Get-Member
```

Cela donne:

```
TypeName: System.IO.DirectoryInfo
```

Suivi d'une liste de propriétés et de méthodes de l'objet.

Une autre façon d'obtenir le type d'un objet consiste à utiliser la méthode GetType, comme ceci:

```
C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()
```

IsPublic	IsSerial	Name	BaseType
True	True	DirectoryInfo	System.IO.FileSystemInfo

Pour afficher la liste des propriétés de l'objet et leurs valeurs, vous pouvez utiliser l'applet de commande `Format-List` avec son paramètre `Property` défini sur: `*` (signifiant tout).

Voici un exemple, avec le résultat obtenu:

```
C:\> Get-Item C:\Windows | Format-List -Property *
```

```

PSPath           : Microsoft.PowerShell.Core\FileSystem::C:\Windows
PSParentPath     : Microsoft.PowerShell.Core\FileSystem::C:\
PSChildName      : Windows
PSDrive          : C
PSProvider       : Microsoft.PowerShell.Core\FileSystem
PSIsContainer    : True
Mode             : d-----
BaseName         : Windows
Target           : {}
LinkType         :
Name             : Windows
Parent           :
Exists           : True
Root             : C:\
FullName         : C:\Windows
Extension        :
CreationTime     : 30/10/2015 06:28:30
CreationTimeUtc  : 30/10/2015 06:28:30
LastAccessTime   : 16/08/2016 17:32:04
LastAccessTimeUtc : 16/08/2016 16:32:04
LastWriteTime    : 16/08/2016 17:32:04
LastWriteTimeUtc : 16/08/2016 16:32:04
Attributes       : Directory

```

Création d'instances de classes génériques

Remarque: exemples écrits pour PowerShell 5.1 Vous pouvez créer des instances de classes génériques

```

#Nullable System.DateTime
[Nullable[datetime]]$nullableDate = Get-Date -Year 2012
$nullableDate
$nullableDate.GetType().FullName
$nullableDate = $null
$nullableDate

#Normal System.DateTime
[datetime]$aDate = Get-Date -Year 2013
$aDate
$aDate.GetType().FullName
$aDate = $null #Throws exception when PowerShell attempts to convert null to

```

Donne la sortie:

```

Saturday, 4 August 2012 08:53:02
System.DateTime
Sunday, 4 August 2013 08:53:02
System.DateTime
Cannot convert null to type "System.DateTime".
At line:14 char:1
+ $aDate = $null
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException

```

Les collections génériques sont également possibles

```

[System.Collections.Generic.SortedDictionary[int, String]]$dict =
[System.Collections.Generic.SortedDictionary[int, String]]::new()
$dict.GetType().FullName

$dict.Add(1, 'a')
$dict.Add(2, 'b')
$dict.Add(3, 'c')

$dict.Add('4', 'd') #powershell auto converts '4' to 4
$dict.Add('5.1', 'c') #powershell auto converts '5.1' to 5

$dict

$dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so it throws an error

```

Donne la sortie:

```

System.Collections.Generic.SortedDictionary`2[[System.Int32, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089]]

Key Value
--- -----
1 a
2 b
3 c
4 d
5 c
Cannot convert argument "key", with value: "z", for "Add" to type "System.Int32": "Cannot
convert value "z" to type "System.Int32". Error: "Input string was not in a correct format."
At line:15 char:1
+ $dict.Add('z', 'z') #powershell can't convert 'z' to System.Int32 so ...
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [], MethodException
+ FullyQualifiedErrorId : MethodArgumentConversionInvalidCastArgument

```

Lire Travailler avec des objets en ligne: <https://riptutorial.com/fr/powershell/topic/1328/travailler-avec-des-objets>

Chapitre 63: Travailler avec le pipeline PowerShell

Introduction

PowerShell introduit un modèle de pipeline d'objets, qui vous permet d'envoyer des objets entiers dans le pipeline vers des commandes consommatrices ou (au moins) la sortie. Contrairement au pipeline à cordes classique, les informations contenues dans les objets canalisés ne doivent pas nécessairement se trouver sur des positions spécifiques. Les commandlets peuvent déclarer interagir avec les objets du pipeline en tant qu'entrée, tandis que les valeurs de retour sont envoyées automatiquement au pipeline.

Syntaxe

- **COMMENCER** Le premier bloc. Exécuté une fois au début. L'entrée du pipeline ici est \$ null, car elle n'a pas été définie.
- **PROCESS** Le deuxième bloc. Exécuté pour chaque élément du pipeline. Le paramètre de pipeline est égal à l'élément actuellement traité.
- **FIN** Dernier bloc. Exécuté une fois à la fin. Le paramètre de pipeline est égal au dernier élément de l'entrée, car il n'a pas été modifié depuis sa définition.

Remarques

Dans la plupart des cas, l'entrée du pipeline sera un tableau d'objets. Bien que le comportement du bloc `PROCESS{}` puisse sembler similaire au bloc `foreach{}`, ignorer un élément du tableau nécessite un processus différent.

Si, comme dans `foreach{}`, vous avez `continue` à l'intérieur du bloc `PROCESS{}`, cela briserait le pipeline en ignorant toutes les instructions suivantes, y compris le bloc `END{}`. Au lieu de cela, utilisez `return` - cela mettra fin uniquement au bloc `PROCESS{}` pour l'élément en cours et passera au suivant.

Dans certains cas, il est nécessaire de générer le résultat des fonctions avec un codage différent. Le codage de la sortie de CmdLets est contrôlé par la variable `$OutputEncoding`. Lorsque la sortie est destinée à être placée dans un pipeline vers des applications natives, il peut être judicieux de corriger l'encodage pour correspondre à la cible `$OutputEncoding = [Console]::OutputEncoding`

Références supplémentaires:

Article de blog avec plus d'informations sur `$OutputEncoding`

<https://blogs.msdn.microsoft.com/powershell/2006/12/11/outputencoding-to-the-rescue/>

Exemples

Fonctions d'écriture avec cycle de vie avancé

Cet exemple montre comment une fonction peut accepter des entrées en pipeline et effectuer une itération efficace.

Notez que les structures de `begin` et de `end` de la fonction sont facultatives lors du `process` pipeline, mais ce `process` est requis lors de l'utilisation de `ValueFromPipeline` ou

`ValueFromPipelineByPropertyName` .

```
function Write-FromPipeline{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline)]
        $myInput
    )
    begin {
        Write-Verbose -Message "Beginning Write-FromPipeline"
    }
    process {
        Write-Output -InputObject $myInput
    }
    end {
        Write-Verbose -Message "Ending Write-FromPipeline"
    }
}

$foo = 'hello','world',1,2,3

$foo | Write-FromPipeline -Verbose
```

Sortie:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
1
2
3
VERBOSE: Ending Write-FromPipeline
```

Prise en charge de base du pipeline dans les fonctions

Voici un exemple de fonction avec le support le plus simple possible pour le traitement en pipeline.

Toute fonction avec prise en charge de pipeline doit avoir au moins un paramètre avec l'ensemble `ParameterAttribute ValueFromPipeline` ou `ValueFromPipelineByPropertyName` , comme indiqué ci-dessous.

```
function Write-FromPipeline {
    param(
        [Parameter(ValueFromPipeline)] # This sets the ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}
```

```
}  
  
$foo = 'Hello World!'  
  
$foo | Write-FromPipeline
```

Sortie:

Hello World!

Remarque: Dans PowerShell 3.0 et versions ultérieures, les valeurs par défaut pour `ParameterAttributes` sont prises en charge. Dans les versions antérieures, vous devez spécifier `ValueFromPipeline=$true`.

Concept de travail du pipeline

Dans une série de pipeline, chaque fonction est parallèle aux autres, comme les threads parallèles. Le premier objet traité est transmis au pipeline suivant et le traitement suivant est immédiatement exécuté dans un autre thread. Cela explique le gain de vitesse élevé par rapport à la norme `ForEach`

```
@( bigFile_1, bigFile_2, ..., bigFile_n ) | Copy-File | Encrypt-File | Get-Md5
```

1. étape - copier le premier fichier (dans le fichier de `Copy-file`)
2. étape - copier le second fichier (dans `Copy-file` thread de `Copy-file`) et simultanément chiffrer le premier (dans `Encrypt-File`)
3. étape - copier un troisième fichier (dans `Copy-file` thread de `Copy-file`) et chiffrer simultanément le second fichier (dans `Encrypt-File`) et `get-Md5` simultanément le `get-Md5` du premier (dans `Get-Md5`)

Lire [Travailler avec le pipeline PowerShell en ligne](https://riptutorial.com/fr/powershell/topic/3937/travailler-avec-le-pipeline-powershell):

<https://riptutorial.com/fr/powershell/topic/3937/travailler-avec-le-pipeline-powershell>

Chapitre 64: URL encoder / décoder

Remarques

L'expression régulière utilisée dans les exemples d' *URL de décodage* provient de la [RFC 2396, annexe B: analyse d'une référence d'URI avec une expression régulière](#) ; pour la postérité, voici une citation:

La ligne suivante est l'expression régulière permettant de décomposer une référence d'URI en ses composants.

```
^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?  
12           3 4           5           6 7           8 9
```

Les chiffres de la deuxième ligne ci-dessus ne servent qu'à faciliter la lisibilité; ils indiquent les points de référence pour chaque sous-expression (c.-à-d. chaque parenthèse appariée). Nous nous référons à la valeur correspondant à la sous-expression \$. Par exemple, faire correspondre l'expression ci-dessus à

```
http://www.ics.uci.edu/pub/ietf/uri/#Related
```

les résultats dans les sous-expressions suivantes correspondent:

```
$1 = http:  
$2 = http  
$3 = //www.ics.uci.edu  
$4 = www.ics.uci.edu  
$5 = /pub/ietf/uri/  
$6 = <undefined>  
$7 = <undefined>  
$8 = #Related  
$9 = Related
```

Exemples

Démarrage rapide: encodage

```
$url1 = [uri]::EscapeDataString("http://test.com?test=my value")  
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value  
  
$url2 = [uri]::EscapeUriString("http://test.com?test=my value")  
# url2: http://test.com?test=my%20value  
  
# HttpUtility requires at least .NET 1.1 to be installed.  
$url3 = [System.Web.HttpUtility]::UrlEncode("http://test.com?test=my value")  
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
```

Remarque: [Plus d'informations sur HttpUtility](#) .

Démarrage rapide: décodage

Remarque: ces exemples utilisent les variables créées dans la section *Démarrage rapide: encodage* ci-dessus.

```
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[uri]::UnescapeDataString($url1)
# Returns: http://test.com?test=my value

# url2: http://test.com?test=my%20value
[uri]::UnescapeDataString($url2)
# Returns: http://test.com?test=my value

# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[uri]::UnescapeDataString($url3)
# Returns: http://test.com?test=my+value

# Note: There is no `[uri]::UnescapeUriString()`;
#       which makes sense since the `[uri]::UnescapeDataString()`
#       function handles everything it would handle plus more.

# HttpUtility requires at least .NET 1.1 to be installed.
# url1: http%3A%2F%2Ftest.com%3Ftest%3Dmy%20value
[System.Web.HttpUtility]::UrlDecode($url1)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url2: http://test.com?test=my%20value
[System.Web.HttpUtility]::UrlDecode($url2)
# Returns: http://test.com?test=my value

# HttpUtility requires at least .NET 1.1 to be installed.
# url3: http%3a%2f%2ftest.com%3ftest%3dmy+value
[System.Web.HttpUtility]::UrlDecode($url3)
# Returns: http://test.com?test=my value
```

Remarque: [Plus d'informations sur HttpUtility](#) .

Encode Query String avec `[uri]::EscapeDataString ()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qqs_data = @(
    'foo1'='bar1';
    'foo2'='complex;/?:@&=+$, bar''''';
    'complex;/?:@&=+$, foo''''='bar2';
)

[System.Collections.ArrayList] $qs_array = @()
foreach ($qs in $qqs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qs_array.Add("$qs_key=$qs_value") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qs_array -join '&'))
```

Avec `[uri]::EscapeDataString()` , vous remarquerez que l'apostrophe (') n'a pas été encodée:

<https://example.vertigion.com/foos?foo2=complexe%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complexe%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1>

Encoder une chaîne de requête avec `[System.Web.HttpUtility]::UrlEncode()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qqs_data = @{
    'foo1'='bar1';
    'foo2'='complex;/?:@&+$/, bar''''';
    'complex;/?:@&+$/, foo''''='bar2';
}

[System.Collections.ArrayList] $qqs_array = @()
foreach ($qs in $qqs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qqs_array.Add("$qs_key=$qs_value") | Out-Null
}

$url = $url_format -f @([uri]:"UriScheme${scheme}", ($qqs_array -join '&'))
```

Avec `[System.Web.HttpUtility]::UrlEncode()` , vous remarquerez que les espaces sont transformés en signes plus (+) au lieu de `%20` :

<https://example.vertigion.com/foos?foo2=complexe%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complexe%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1>

Décoder l'URL avec `[uri]::UnescapeDataString()`

Encodé avec `[uri]::EscapeDataString()`

Tout d'abord, nous décodons l'URL et la chaîne de requête encodées avec

`[uri]::EscapeDataString()` dans l'exemple ci-dessus:

<https://example.vertigion.com/foos?foo2=complexe%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complexe%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1>

```
$url =
'https://example.vertigion.com/foos?foo2=complexe%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complexe%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1'

$url_parts_regex = '^(([/?#]+):)?(//(?:/?#]*)?([?#]*) (\?([?#]*)?#(?:.*)?)?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{
        'Scheme' = $Matches[2];
        'Server' = $Matches[4];
        'Path' = $Matches[5];
    }
}
```

```

    'QueryString' = $Matches[7];
    'QueryStringParts' = @{}
}

foreach ($qs in $query_string.Split('&')) {
    $qs_key, $qs_value = $qs.Split('=')
    $url_parts.QueryStringParts.Add(
        [uri]::UnescapeDataString($qs_key),
        [uri]::UnescapeDataString($qs_value)
    ) | Out-Null
}
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}

```

Cela vous ramène `[hashtable]$url_parts` ; qui est égal à (**Note:** les espaces dans les parties complexes sont des *espaces*):

```

PS > $url_parts

Name                           Value
----                           -
Scheme                         https
Path                           /foos
Server                         example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=b
QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}

PS > $url_parts.QueryStringParts

Name                           Value
----                           -
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"       bar2
foo1                           bar1

```

Encodé avec `[System.Web.HttpUtility]::UrlEncode()`

Nous allons maintenant décoder l'URL et la chaîne de requête encodées avec

`[System.Web.HttpUtility]::UrlEncode()` dans l'exemple ci-dessus:

<https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1>

```

$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'

$url_parts_regex = '^(([/?:?#]+):)?(//([^/?#]*)?([^?#]*)?(\?([^?#]*)?)?(#.*)?)?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{}
    'Scheme' = $Matches[2];
    'Server' = $Matches[4];
}

```

```

        'Path' = $Matches[5];
        'QueryString' = $Matches[7];
        'QueryStringParts' = @{}
    }

    foreach ($qs in $query_string.Split('&')) {
        $qs_key, $qs_value = $qs.Split('=')
        $url_parts.QueryStringParts.Add(
            [uri]::UnescapeDataString($qs_key),
            [uri]::UnescapeDataString($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}

```

Cela vous donne `[hashtable]$url_parts` , ce qui équivaut à (**Note:** les *espaces* dans les parties complexes sont des *signes plus* (+) dans la première partie et des *espaces* dans la deuxième partie):

```

PS > $url_parts

Name                               Value
----                               -
Scheme                             https
Path                               /foos
Server                             example.vertigion.com
QueryString
foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar1

QueryStringParts                   {foo2, complex;/?:@&=+$, foo', foo1}

PS > $url_parts.QueryStringParts

Name                               Value
----                               -
foo2                               complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"           bar2
foo1                               bar1

```

Décoder l'URL avec `[System.Web.HttpUtility]::UrlDecode()`

Encodé avec `[uri]::EscapeDataString()`

Tout d'abord, nous décodons l'URL et la chaîne de requête encodées avec

`[uri]::EscapeDataString()` dans l'exemple ci-dessus:

<https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1>

```

$url =
'https://example.vertigion.com/foos?foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar2&foo1=bar1'

```

```

$url_parts_regex = '^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?' # See Remarks

if ($url -match $url_parts_regex) {
    $url_parts = @{}
    'Scheme' = $Matches[2];
    'Server' = $Matches[4];
    'Path' = $Matches[5];
    'QueryString' = $Matches[7];
    'QueryStringParts' = @{}
}

foreach ($qs in $query_string.Split('&')) {
    $qs_key, $qs_value = $qs.Split('=')
    $url_parts.QueryStringParts.Add(
        [System.Web.HttpUtility]::UrlDecode($qs_key),
        [System.Web.HttpUtility]::UrlDecode($qs_value)
    ) | Out-Null
}
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}

```

Cela vous ramène `[hashtable]$url_parts` ; qui est égal à (**Note:** les espaces dans les parties complexes sont des espaces):

```

PS > $url_parts

Name                               Value
----                               -
Scheme                             https
Path                               /foos
Server                             example.vertigion.com
QueryString
foo2=complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22&complex%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=bar1
foo1=bar1

QueryStringParts                   {foo2, complex;/?:@&=+$, foo'", foo1}

PS > $url_parts.QueryStringParts

Name                               Value
----                               -
foo2                               complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"           bar2
foo1                               bar1

```

Encodé avec `[System.Web.HttpUtility]::UrlEncode()`

Nous allons maintenant décoder l'URL et la chaîne de requête encodées avec

`[System.Web.HttpUtility]::UrlEncode()` dans l'exemple ci-dessus:

<https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1>

```

$url =
'https://example.vertigion.com/foos?foo2=complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complex%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=bar2&foo1=bar1'

```

Chapitre 65: Utilisation du système d'aide

Remarques

`Get-Help` est une applet de commande permettant de lire les rubriques d'aide dans PowerShell.

En savoir plus sur [TechNet](#)

Exemples

Mise à jour du système d'aide

3.0

À partir de PowerShell 3.0, vous pouvez télécharger et mettre à jour la documentation d'aide hors ligne à l'aide d'une seule applet de commande.

```
Update-Help
```

Pour mettre à jour l'aide sur plusieurs ordinateurs (ou ordinateurs non connectés à Internet).

Exécutez les opérations suivantes sur un ordinateur avec les fichiers d'aide

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

Pour exécuter sur de nombreux ordinateurs à distance

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath \\Server01\Share\Help -Credential $cred}
```

Utilisation de Get-Help

`Get-Help` peut être utilisé pour afficher l'aide de PowerShell. Vous pouvez rechercher des applets de commande, des fonctions, des fournisseurs ou d'autres rubriques.

Pour afficher la documentation d'aide sur les travaux, utilisez:

```
Get-Help about_Jobs
```

Vous pouvez rechercher des sujets à l'aide de caractères génériques. Si vous voulez lister les rubriques d'aide disponibles avec un titre commençant par `about_`, essayez:

```
Get-Help about_*
```

Si vous voulez de l'aide sur `Select-Object`, vous utiliseriez:

```
Get-Help Select-Object
```

Vous pouvez également utiliser l' `help` alias ou `man` .

Affichage de la version en ligne d'une rubrique d'aide

Vous pouvez accéder à la documentation d'aide en ligne en utilisant:

```
Get-Help Get-Command -Online
```

Exemples d'affichage

Afficher des exemples d'utilisation d'une applet de commande spécifique.

```
Get-Help Get-Command -Examples
```

Affichage de la page d'aide complète

Affichez la documentation complète du sujet.

```
Get-Help Get-Command -Full
```

Affichage de l'aide pour un paramètre spécifique

Vous pouvez afficher l'aide pour un paramètre spécifique en utilisant:

```
Get-Help Get-Content -Parameter Path
```

Lire Utilisation du système d'aide en ligne:

<https://riptutorial.com/fr/powershell/topic/5644/utilisation-du-systeme-d-aide>

Chapitre 66: Utiliser des classes statiques existantes

Introduction

Ces classes sont des bibliothèques de référence de méthodes et de propriétés qui ne changent pas d'état, en un mot, immuable. Vous n'avez pas besoin de les créer, vous les utilisez simplement. Les classes et les méthodes telles que celles-ci sont appelées classes statiques car elles ne sont pas créées, détruites ou modifiées. Vous pouvez faire référence à une classe statique en entourant le nom de la classe de crochets.

Exemples

Créer un nouveau GUID instantanément

Utilisez les classes .NET existantes instantanément avec PowerShell en utilisant `[class] :: Method (args)`:

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

De même, dans PowerShell 5+, vous pouvez utiliser l'applet de commande `New-Guid` :

```
PS C:\> New-Guid

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Pour obtenir le GUID en tant que `[String]` uniquement, `.Guid` référence à la propriété `.Guid` :

```
[guid]::NewGuid().Guid
```

Utiliser la classe `Math` .Net

Vous pouvez utiliser la classe .Net `Math` pour effectuer des calculs (`[System.Math]`)

Si vous voulez savoir quelles méthodes sont disponibles, vous pouvez utiliser:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Voici quelques exemples d'utilisation de la classe `Math`:

```
PS C:\> [System.Math]::Floor(9.42)
9
PS C:\> [System.Math]::Ceiling(9.42)
10
PS C:\> [System.Math]::Pow(4,3)
64
PS C:\> [System.Math]::Sqrt(49)
7
```

Ajouter des types

Par nom d'assemblée, ajouter une bibliothèque

```
Add-Type -AssemblyName "System.Math"
```

ou par chemin de fichier:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

Pour utiliser le type ajouté:

```
[CustomMath.Namespace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

[Lire Utiliser des classes statiques existantes en ligne:](https://riptutorial.com/fr/powershell/topic/1522/utiliser-des-classes-statiques-existantes)

<https://riptutorial.com/fr/powershell/topic/1522/utiliser-des-classes-statiques-existantes>

Chapitre 67: Utiliser la barre de progression

Introduction

Une barre de progression peut être utilisée pour montrer que quelque chose est dans un processus. Les barres de progression sont incroyablement utiles lors du débogage pour déterminer quelle partie du script est en cours d'exécution et elles sont satisfaisantes pour les utilisateurs exécutant des scripts pour suivre ce qui se passe. Il est courant d'afficher une sorte de progression lorsqu'un script prend beaucoup de temps à compléter. Lorsqu'un utilisateur lance le script et que rien ne se passe, on commence à se demander si le script a été lancé correctement.

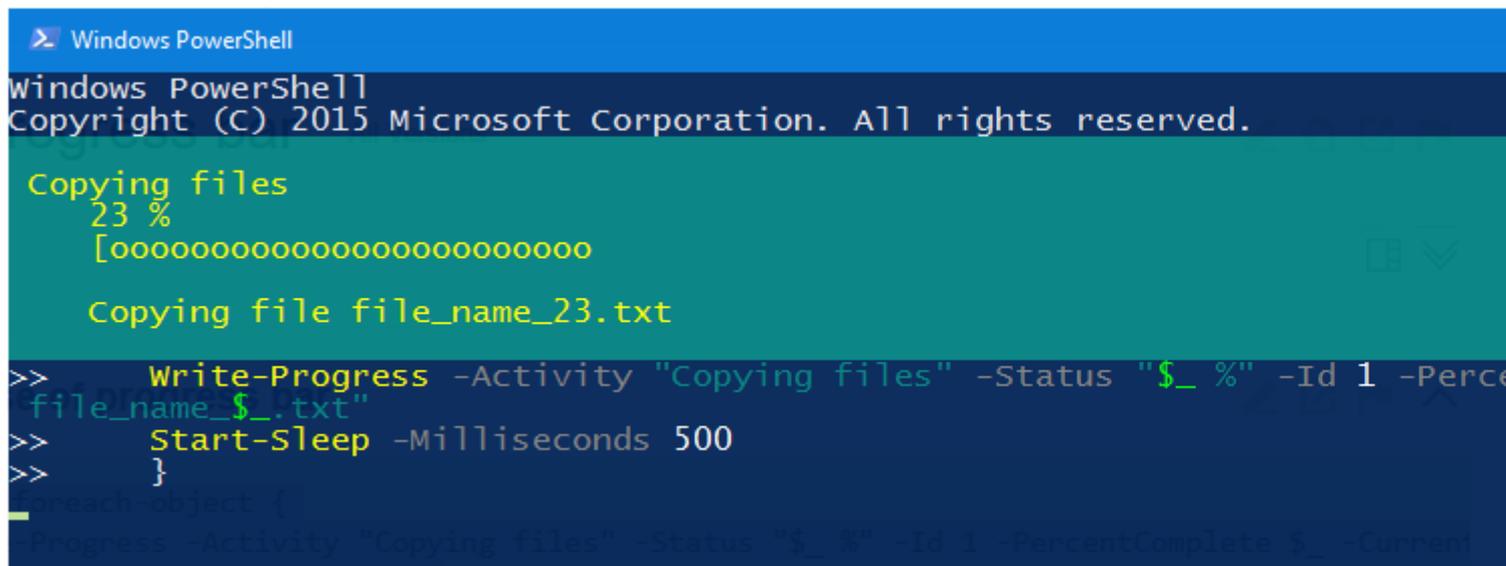
Exemples

Utilisation simple de la barre de progression

```
1..100 | ForEach-Object {
    Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -
CurrentOperation "Copying file file_name_$_.txt"
    Start-Sleep -Milliseconds 500 # sleep simulates working code, replace this line
with your executive code (i.e. file copying)
}
```

Veillez noter que, pour des raisons de concision, cet exemple ne contient aucun code exécutif (simulé avec `Start-Sleep`). Cependant, il est possible de l'exécuter directement tel quel et de le modifier et de le jouer.

Voici comment les résultats apparaissent dans la console PS:



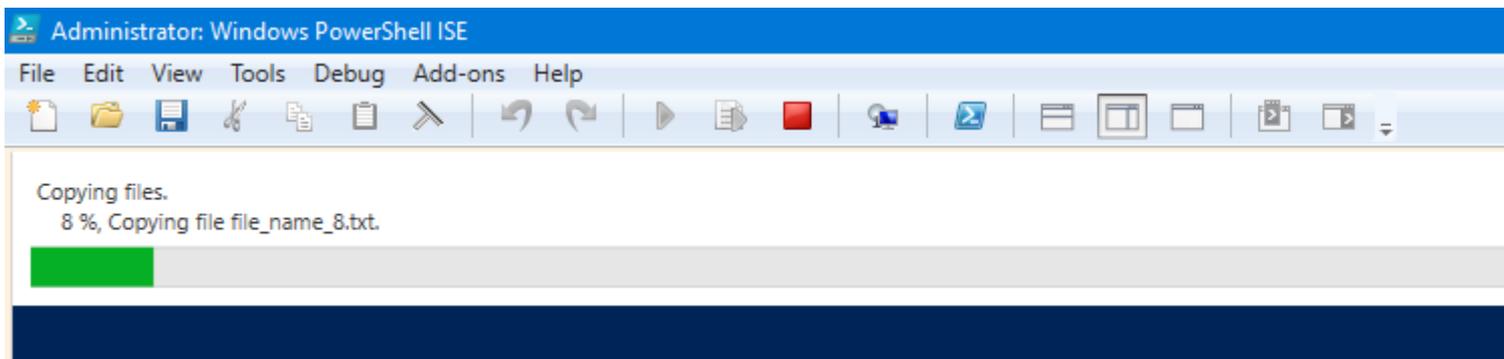
```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

Copying files
23 %
[ooooooooooooooooooooooooooooo

Copying file file_name_23.txt

>> Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -PercentComplete $_ -CurrentOperation "Copying file file_name_$_.txt"
>> Start-Sleep -Milliseconds 500
>> }
```

Voici comment les résultats apparaissent dans PS ISE:



Utilisation de la barre de progression interne

```
1..10 | foreach-object {
    $fileName = "file_name_$.txt"
    Write-Progress -Activity "Copying files" -Status "$($_*10) %" -Id 1 -PercentComplete
    ($_*10) -CurrentOperation "Copying file $fileName"

    1..100 | foreach-object {
        Write-Progress -Activity "Copying contents of the file $fileName" -Status "$_ %" -
        Id 2 -ParentId 1 -PercentComplete $_ -CurrentOperation "Copying $_. line"

        Start-Sleep -Milliseconds 20 # sleep simulates working code, replace this line
        with your executive code (i.e. file copying)
    }

    Start-Sleep -Milliseconds 500 # sleep simulates working code, replace this line with
    your executive code (i.e. file search)
}
```

Veillez noter que, pour des raisons de concision, cet exemple ne contient aucun code exécutif (simulé avec `Start-Sleep`). Cependant, il est possible de l'exécuter directement tel quel et de le modifier et de le jouer.

Voici comment les résultats apparaissent dans la console PS:

```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

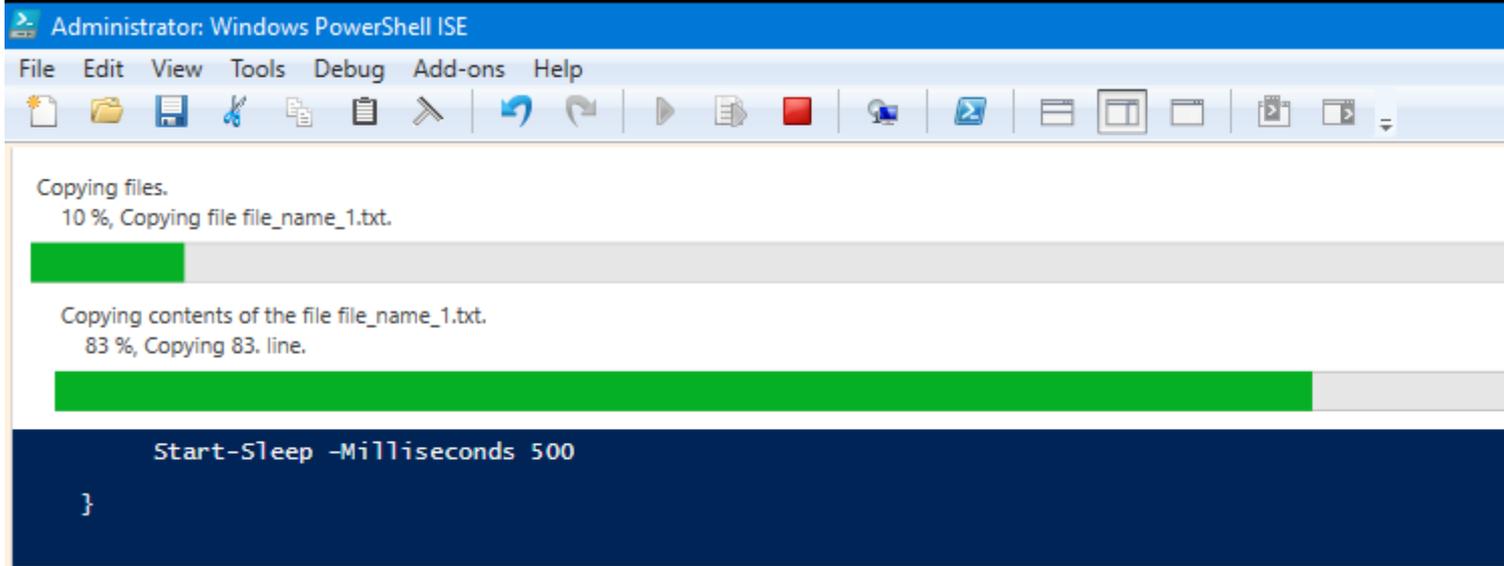
Copying files
30 %
[oooooooooooooooooooooooooooooooooooo

Copying file file_name_3.txt
Copying contents of the file file_name_3.txt
46 %
[oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo

Copying 46. line

>>>     $fileName = "file_name_$_.txt"
>>>     Write-Progress -Activity "Copying files" -Status "$($_*10) %" -I
n "Copying file $fileName"
>>>
>>>     1..100 | foreach-object {
>>>         Write-Progress -Activity "Copying contents of the file $file
ntComplete $_ -CurrentOperation "Copying $_. line"
>>>         Start-Sleep -Milliseconds 20
>>>     }
>>>
>>>     Start-Sleep -Milliseconds 500
>>> }
```

Voici comment les résultats apparaissent dans PS ISE:



Lire Utiliser la barre de progression en ligne: <https://riptutorial.com/fr/powershell/topic/5020/utiliser-la-barre-de-progression>

Chapitre 68: Utiliser ShouldProcess

Syntaxe

- \$PSCmdlet.ShouldProcess ("Target")
- \$PSCmdlet.ShouldProcess ("Target", "Action")

Paramètres

Paramètre	Détails
Cible	La ressource en cours de modification
action	L'opération en cours. Par défaut, le nom de l'applet de commande.

Remarques

\$PSCmdlet.ShouldProcess() écrira également automatiquement un message à la sortie détaillée.

```
PS> Invoke-MyCmdlet -Verbose
VERBOSE: Performing the operation "Invoke-MyCmdlet" on target "Target of action"
```

Exemples

Ajout de la prise en charge de -WhatIf et -Confirm à votre applet de commande

```
function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}
```

Utiliser ShouldProcess () avec un argument

```
if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Do the thing
}
```

Lors de l'utilisation de -WhatIf :

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

En utilisant -Confirm :

```
Are you sure you want to perform this action?  
Performing operation "Invoke-MyCmdlet" on target "Target of action"  
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "Y"):
```

Exemple d'utilisation complète

D'autres exemples ne pouvaient pas m'expliquer clairement comment déclencher la logique conditionnelle.

Cet exemple montre également que les commandes sous-jacentes écoutent également le drapeau `-Confirm`!

```
<#  
Restart-Win32Computer  
#>  
  
function Restart-Win32Computer  
{  
    [CmdletBinding(SupportsShouldProcess=$true,ConfirmImpact="High")]  
    param (  
        [parameter(Mandatory=$true,ValueFromPipeline=$true,ValueFromPipelineByPropertyName=$true)]  
        [string[]]$computerName,  
        [parameter(Mandatory=$true)]  
        [string][ValidateSet("Restart","LogOff","Shutdown","PowerOff")] $action,  
        [boolean]$force = $false  
    )  
    BEGIN {  
        # translate action to numeric value required by the method  
        switch($action) {  
            "Restart"  
            {  
                $_action = 2  
                break  
            }  
            "LogOff"  
            {  
                $_action = 0  
                break  
            }  
            "Shutdown"  
            {  
                $_action = 2  
                break  
            }  
            "PowerOff"  
            {  
                $_action = 8  
                break  
            }  
        }  
        # to force, add 4 to the value  
        if($force)  
        {  
            $_action += 4  
        }  
        write-verbose "Action set to $action"  
    }  
    PROCESS {
```

```
write-verbose "Attempting to connect to $computername"
# this is how we support -whatif and -confirm
# which are enabled by the SupportsShouldProcess
# parameter in the cmdlet bindnig
if($pscmdlet.ShouldProcess($computername)) {
    get-wmiobject win32_operatingsystem -computername $computername | invoke-wmimethod -
name Win32Shutdown -argumentlist $_action
}
}
}
#Usage:
#This will only output a description of the actions that this command would execute if -WhatIf
is removed.
'localhost','server1'| Restart-Win32Computer -action LogOff -whatif

#This will request the permission of the caller to continue with this item.
#Attention: in this example you will get two confirmation request because all cmdlets called
by this cmdlet that also support ShouldProcess, will ask for their own confirmations...
'localhost','server1'| Restart-Win32Computer -action LogOff -Confirm
```

Lire Utiliser ShouldProcess en ligne: <https://riptutorial.com/fr/powershell/topic/1145/utiliser-shouldprocess>

Chapitre 69: Variables Automatiques

Introduction

Les variables automatiques sont créées et gérées par Windows PowerShell. On a la possibilité d'appeler une variable à peu près n'importe quel nom dans le livre; Les seules exceptions sont les variables déjà gérées par PowerShell. Ces variables seront sans aucun doute les objets les plus répétitifs que vous utiliserez dans PowerShell à côté des fonctions (comme `$?` - indique le statut Success / Failure de la dernière opération).

Syntaxe

- `$$` - Contient le dernier jeton de la dernière ligne reçue par la session.
- `$$^` - Contient le premier jeton de la dernière ligne reçue par la session.
- `$$?` - Contient le statut d'exécution de la dernière opération.
- `$$_` - Contient l'objet en cours dans le pipeline

Exemples

`$ pid`

Contient l'ID de processus du processus d'hébergement en cours.

```
PS C:\> $pid
26080
```

Valeurs booléennes

`$true` et `$false` sont deux variables représentant VRAI et FAUX.

Notez que vous devez spécifier le signe dollar comme premier caractère (différent de C #).

```
$boolExpr = "abc".Length -eq 3 # length of "abc" is 3, hence $boolExpr will be True
if($boolExpr -eq $true){
    "Length is 3"
}
# result will be "Length is 3"
$boolExpr -ne $true
#result will be False
```

Notez que lorsque vous utilisez booléen true / false dans votre code, vous écrivez `$true` ou `$false`, mais lorsque Powershell renvoie un booléen, il ressemble à `True` ou `False`

`$ null`

`$null` est utilisé pour représenter une valeur absente ou indéfinie.

`$null` peut être utilisé comme espace réservé vide pour les valeurs vides dans les tableaux:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

Lorsque nous utilisons le même tableau que la source de `ForEach-Object`, il traitera les trois éléments (y compris `$ null`):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Faites attention! Cela signifie que `ForEach-Object` traitera même `$null` par lui - même:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Ce qui est un résultat très inattendu si vous le comparez à la boucle `foreach` classique:

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

\$ OFS

La variable appelée Output Field Separator contient une valeur de chaîne utilisée lors de la conversion d'un tableau en chaîne. Par défaut `$OFS = " "` (*un espace*), mais il peut être modifié:

```
PS C:\> $array = 1,2,3
PS C:\> "$array" # default OFS will be used
1 2 3
PS C:\> $OFS = ",." # we change OFS to comma and dot
PS C:\> "$array"
1,.2,.3
```

\$ _ / \$ PSItem

Contient l'objet / élément en cours de traitement par le pipeline.

```
PS C:\> 1..5 | % { Write-Host "The current item is $_" }
The current item is 1
The current item is 2
The current item is 3
The current item is 4
The current item is 5
```

`$PSItem` et `$_` sont identiques et peuvent être utilisés indifféremment, mais `$_` est de loin le plus utilisé.

\$?

Contient le statut de la dernière opération. Lorsqu'il n'y a pas d'erreur, il est défini sur `True` :

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

S'il y a une erreur, elle est définie sur `False` :

```
PS C:\> wrt-host
wrt-host : The term 'wrt-host' is not recognized as the name of a cmdlet, function, script
file, or operable program.
Check the spelling of the name, or if a path was included, verify that the path is correct and
try again.
At line:1 char:1
+ wrt-host
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (wrt-host:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\> $?
False
```

\$ erreur

Tableau des objets d'erreur les plus récents. Le premier dans le tableau est le plus récent:

```
PS C:\> throw "Error" # resulting output will be in red font
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~
+ CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
+ FullyQualifiedErrorId : Error

PS C:\> $error[0] # resulting output will be normal string (not red)
Error
At line:1 char:1
+ throw "Error"
+ ~~~~~
+ CategoryInfo          : OperationStopped: (Error:String) [], RuntimeException
+ FullyQualifiedErrorId : Error
```

Conseils d'utilisation: Lorsque vous utilisez la variable `$error` dans une applet de commande de format (par exemple, une liste de format), `-Force` à utiliser le commutateur `-Force` . Sinon, l'applet de commande de format va afficher le contenu d' `$error` ci-dessus.

Les entrées d'erreur peuvent être supprimées via par exemple `$Error.Remove($Error[0])` .

Lire Variables Automatiques en ligne: <https://riptutorial.com/fr/powershell/topic/5353/variables-automatiques>

Chapitre 70: Variables automatiques - partie 2

Introduction

Le sujet "Variables automatiques" a déjà 7 exemples listés et nous ne pouvons pas en ajouter plus. Ce sujet aura une suite de variables automatiques.

Les variables automatiques sont des variables qui stockent des informations d'état pour PowerShell. Ces variables sont créées et gérées par Windows PowerShell.

Remarques

Je ne suis pas sûr que ce soit la meilleure façon de gérer la documentation des variables automatiques, mais cela vaut mieux que rien. S'il vous plaît commenter si vous trouvez un meilleur moyen :)

Exemples

\$ PSVersionTable

Contient une table de hachage en lecture seule (constante, AllScope) qui affiche des détails sur la version de PowerShell en cours d'exécution dans la session en cours.

```
$PSVersionTable      #this call results in this:
Name                 Value
----                 -
PSVersion            5.0.10586.117
PSCompatibleVersions {1.0, 2.0, 3.0, 4.0...}
BuildVersion         10.0.10586.117
CLRVersion           4.0.30319.42000
WSManStackVersion   3.0
PSRemotingProtocolVersion 2.3
SerializationVersion 1.1.0.1
```

Le moyen le plus rapide d'obtenir une version de PowerShell en cours d'exécution:

```
$PSVersionTable.PSVersion
# result :
Major  Minor  Build  Revision
-----
5      0      10586  117
```

Lire [Variables automatiques - partie 2 en ligne](#):

<https://riptutorial.com/fr/powershell/topic/8639/variables-automatiques---partie-2>

Chapitre 71: Variables d'environnement

Exemples

Les variables d'environnement Windows sont visibles en tant que lecteur PS appelé Env:

Vous pouvez voir la liste avec toutes les variables d'environnement avec:
Get-Childitem env:

Appel instantané des variables d'environnement avec \$ env:

```
$env:COMPUTERNAME
```

Lire Variables d'environnement en ligne: <https://riptutorial.com/fr/powershell/topic/5635/variables-d-environnement>

Chapitre 72: Variables dans PowerShell

Introduction

Les variables sont utilisées pour stocker des valeurs. Soit la valeur de n'importe quel type, nous devons la stocker quelque part afin de pouvoir l'utiliser dans toute la console / le script. Les noms de variable dans PowerShell commencent par \$, comme dans \$ Variable1 , et les valeurs sont affectées avec = , comme \$ Variable1 = "Value 1" .PowerShell prend en charge un grand nombre de types de variables; comme les chaînes de texte, les entiers, les décimales, les tableaux et même les types avancés tels que les numéros de version ou les adresses IP.

Exemples

Variable simple

Toutes les variables dans Powershell commencent par un signe dollar américain (\$). L'exemple le plus simple est le suivant:

```
$foo = "bar"
```

Cette instruction alloue une variable appelée `foo` avec une valeur de chaîne de "bar".

Supprimer une variable

Pour supprimer une variable de la mémoire, vous pouvez utiliser l'applet de commande `Remove-Item` . Note: Le nom de la variable n'inclut pas le \$.

```
Remove-Item Variable:\foo
```

Variable a un fournisseur pour permettre à la plupart des applets de commande * -item de fonctionner comme les systèmes de fichiers.

Une autre méthode pour supprimer une variable consiste à utiliser l'applet de commande `Remove-Variable` et son alias `rv`

```
$var = "Some Variable" #Define variable 'var' containing the string 'Some Variable'
$var #For test and show string 'Some Variable' on the console

Remove-Variable -Name var
$var

#also can use alias 'rv'
rv var
```

Portée

La **portée** par défaut d'une variable est le conteneur englobant. En dehors d'un script ou d'un autre conteneur, la portée est `Global`. Pour spécifier une **étendue**, elle est préfixée par le nom de la variable `$scope:varname` comme ceci:

```
$foo = "Global Scope"
function myFunc {
    $foo = "Function (local) scope"
    Write-Host $global:foo
    Write-Host $local:foo
    Write-Host $foo
}
myFunc
Write-Host $local:foo
Write-Host $foo
```

Sortie:

```
Global Scope
Function (local) scope
Function (local) scope
Global Scope
Global Scope
```

Lecture d'une sortie CmdLet

Par défaut, powershell renverrait la sortie à l'entité appelante. Considérez l'exemple ci-dessous,

```
Get-Process -Name excel
```

Cela renverrait simplement à l'entité appelante le processus en cours d'exécution qui correspond au nom Excel. Dans ce cas, l'hôte PowerShell. Il imprime quelque chose comme,

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	-----	--	--	-----
1037	54	67632	62544	617	5.23	4544	1	EXCEL

Maintenant, si vous affectez la sortie à une variable, elle n'imprimera tout simplement rien. Et bien sûr, la variable contient la sortie. (Que ce soit une chaîne, Object - N'importe quel type d'ailleurs)

```
$allExcel = Get-Process -Name excel
```

Donc, disons que vous avez un scénario où vous voulez assigner une variable par un nom dynamique, vous pouvez utiliser le paramètre `-OutVariable`

```
Get-Process -Name excel -OutVariable AllRunningExcel
```

Notez que le "\$" est manquant ici. Une différence majeure entre ces deux affectations est que, il imprime également la sortie en dehors de l'affecter à la variable `AllRunningExcel`. Vous pouvez

également choisir de l'assigner à une autre variable.

```
$VarOne = Get-Process -Name excel -OutVariable VarTwo
```

Bien que le scénario ci-dessus soit très rare, les deux variables \$ VarOne & \$ VarTwo auront la même valeur.

Maintenant considérez ceci,

```
Get-Process -Name EXCEL -OutVariable MSOFFICE  
Get-Process -Name WINWORD -OutVariable +MSOFFICE
```

La première instruction obtiendrait simplement un processus Excel et l'assignerait à la variable MSOFFICE, et ensuite, les processus de mot ms seraient exécutés et "Ajouter" à la valeur existante de MSOFFICE. Cela ressemblerait à quelque chose comme ça,

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	-----	-----	--	--	-----
1047	54	67720	64448	618	5.70	4544	1	EXCEL
1172	70	50052	81780	584	1.83	14968	1	WINWORD

Affectation de liste de variables multiples

Powershell permet d'affecter plusieurs variables et traite presque tout comme un tableau ou une liste. Cela signifie qu'au lieu de faire quelque chose comme ça:

```
$input = "foo.bar.baz"  
$parts = $input.Split(".")  
$foo = $parts[0]  
$bar = $parts[1]  
$baz = $parts[2]
```

Vous pouvez simplement faire ceci:

```
$foo, $bar, $baz = $input.Split(".")
```

Étant donné que Powershell traite les affectations de cette manière comme des listes, si la liste contient plus de valeurs que les éléments de votre liste de variables, la dernière variable devient un tableau des valeurs restantes. Cela signifie que vous pouvez aussi faire des choses comme ceci:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]  
$bar = $leftover[0] # $bar = "bar"  
$baz = $leftover[1] # $baz = "baz"
```

Tableaux

La déclaration de tableau dans Powershell est presque identique à l'instanciation de toute autre variable, c'est-à-dire que vous utilisez une syntaxe `$name =` . Les éléments du tableau sont déclarés en les séparant par des virgules (,) :

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Ajout à un arry

L'ajout à un tableau est aussi simple que l'utilisation de l'opérateur + :

```
$myArrayOfInts = $myArrayOfInts + 5
//now contains 1,2,3,4 & 5!
```

Combiner des tableaux ensemble

Encore une fois, c'est aussi simple que d'utiliser l'opérateur +

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
//now 1,2,3,4,5,6,7
```

Lire Variables dans PowerShell en ligne: <https://riptutorial.com/fr/powershell/topic/3457/variables-dans-powershell>

Chapitre 73: Variables intégrées

Introduction

PowerShell offre une variété de variables "automatiques" utiles. Certaines variables automatiques ne sont remplies que dans des circonstances spéciales, tandis que d'autres sont disponibles globalement.

Exemples

\$ PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

Cet exemple récupère la liste des éléments enfants (répertoires et fichiers) du dossier dans lequel réside le fichier de script.

La variable automatique `$PSScriptRoot` est `$null` si elle est utilisée à l'extérieur d'un fichier de code PowerShell. S'il est utilisé *dans* un script PowerShell, il définit automatiquement le chemin d'accès complet au système de fichiers vers le répertoire contenant le fichier de script.

Dans Windows PowerShell 2.0, cette variable est valide uniquement dans les modules de script (.psm1). Depuis Windows PowerShell 3.0, il est valide dans tous les scripts.

\$ Args

```
$Args
```

Contient un tableau des paramètres non déclarés et / ou des valeurs de paramètres transmis à une fonction, à un script ou à un bloc de script. Lorsque vous créez une fonction, vous pouvez déclarer les paramètres en utilisant le mot-clé `param` ou en ajoutant une liste de paramètres séparés par des virgules entre parenthèses après le nom de la fonction.

Dans une action d'événement, la variable `$ Args` contient des objets qui représentent les arguments d'événement de l'événement en cours de traitement. Cette variable est renseignée uniquement dans le bloc Action d'une commande d'enregistrement d'événement. La valeur de cette variable se trouve également dans la propriété `SourceArgs` de l'objet `PSEventArgs` (System.Management.Automation.PSEventArgs) renvoyée par `Get-Event`.

\$ PSItem

```
Get-Process | ForEach-Object -Process {  
    $PSItem.Name  
}
```

Identique à `$_`. Contient l'objet actuel dans l'objet pipeline. Vous pouvez utiliser cette variable dans des commandes qui exécutent une action sur chaque objet ou sur des objets sélectionnés dans un pipeline.

`$?`

```
Get-Process -Name doesnotexist  
Write-Host -Object "Was the last operation successful? $?"
```

Contient le statut d'exécution de la dernière opération. Il contient `TRUE` si la dernière opération a réussi et `FALSE` si elle a échoué.

`$ erreur`

```
Get-Process -Name doesnotexist  
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Contient un tableau d'objets d'erreur qui représentent les erreurs les plus récentes. L'erreur la plus récente est le premier objet d'erreur du tableau (`$ Error [0]`).

Pour éviter qu'une erreur ne soit ajoutée au tableau `$ Error`, utilisez le paramètre commun `ErrorAction` avec une valeur `Ignore`. Pour plus d'informations, consultez `about_CommonParameters` (<http://go.microsoft.com/fwlink/?LinkID=113216>) .

Lire Variables intégrées en ligne: <https://riptutorial.com/fr/powershell/topic/8732/variables-integrees>

Chapitre 74: WMI et CIM

Remarques

CIM vs WMI

À partir de PowerShell 3.0, il existe deux manières de travailler avec des classes de gestion dans PowerShell, WMI et CIM. PowerShell 1.0 et 2.0 ne prenaient en charge que le module WMI, désormais remplacé par le nouveau module CIM amélioré. Dans une version ultérieure de PowerShell, les cmdlets WMI seront supprimées.

Comparaison des modules CIM et WMI:

CIM-cmdlet	WMI-cmdlet	Ce qu'il fait
Get-CimInstance	Get-WmiObject	Obtient des objets CIM / WMI pour une classe
Invoke-CimMethod	Invoke-WmiMethod	Invoque une méthode de classe CIM / WMI
Register-CimIndicationEvent	Register-WmiEvent	Enregistre l'événement pour une classe CIM / WMI
Remove-CimInstance	Supprimer-WmiObject	Supprimer l'objet CIM / WMI
Set-CimInstance	Set-WmiInstance	Met à jour / enregistre l'objet CIM / WMI
Get-CimAssociatedInstance	N / A	Obtenir les instances associées (objet / classes liés)
Get-CimClass	Get-WmiObject-List	Liste des classes CIM / WMI
New-CimInstance	N / A	Créer un nouvel objet CIM
Get-CimSession	N / A	Listes des sessions CIM
New-CimSession	N / A	Créer une nouvelle session CIM
New-CimSessionOption	N / A	Crée un objet avec des options de session; protocole, encodage, désactivation du cryptage, etc. (à utiliser avec <code>New-CimSession</code>)
Remove-CimSession	N / A	Supprime / Arrête la session CIM

Ressources additionnelles

[Dois-je utiliser CIM ou WMI avec Windows PowerShell? @ Hé, le scripteur! Blog](#)

Exemples

Interroger des objets

CIM / WMI est le plus souvent utilisé pour interroger des informations ou la configuration sur un périphérique. Une classe qui représente une configuration, un processus, un utilisateur, etc. Dans PowerShell, il existe plusieurs manières d'accéder à ces classes et instances, mais les méthodes les plus courantes sont les applets de commande `Get-CimInstance` (CIM) ou `Get-WmiObject` (WMI).

Liste tous les objets pour la classe CIM

Vous pouvez répertorier toutes les instances d'une classe.

3.0

CIM:

```
> Get-CimInstance -ClassName Win32_Process
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
0	System Idle Process	0	4096	65536
4	System	1459	32768	3563520
480	Secure System	0	3731456	0
484	smss.exe	52	372736	2199029891072
....				
....				

WMI:

```
Get-WmiObject -Class Win32_Process
```

En utilisant un filtre

Vous pouvez appliquer un filtre pour obtenir uniquement des instances spécifiques d'une classe CIM / WMI. Les filtres sont écrits en utilisant `WQL` (par défaut) ou `CQL` (ajoutez `-QueryDialect CQL`). `-Filter` utilise la partie `WHERE` d'une requête `WQL` / `CQL` complète.

3.0

CIM:

```
Get-CimInstance -ClassName Win32_Process -Filter "Name = 'powershell.exe'"
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
4800	powershell.exe	676	88305664	2199697199104

WMI:

```
Get-WmiObject -Class Win32_Process -Filter "Name = 'powershell.exe'"
```

```
...
Caption                : powershell.exe
CommandLine            : "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
CreationClassName      : Win32_Process
CreationDate           : 20160913184324.393887+120
CSCreationClassName    : Win32_ComputerSystem
CSName                 : STACKOVERFLOW-PC
Description            : powershell.exe
ExecutablePath         : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
ExecutionState         :
Handle                 : 4800
HandleCount            : 673
....
```

En utilisant une requête WQL:

Vous pouvez également utiliser une requête WQL / CQL pour interroger et filtrer des instances.

3.0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"
```

ProcessId	Name	HandleCount	WorkingSetSize	VirtualSize
4800	powershell.exe	673	88387584	2199696674816

Interrogation d'objets dans un espace de noms différent:

3.0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct
```

```
displayName            : Windows Defender
instanceGuid           : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState           : 397568
timestamp              : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName         :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct

__GENUS                : 2
__CLASS                : AntiVirusProduct
__SUPERCLASS           :
__DYNASTY              : AntiVirusProduct
__RELPATH              : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
__PROPERTY_COUNT       : 6
__DERIVATION           : {}
__SERVER               : STACKOVERFLOW-PC
__NAMESPACE            : ROOT\SecurityCenter2
__PATH                 : \\STACKOVERFLOW-PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName            : Windows Defender
instanceGuid           : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState           : 397568
timestamp              : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName         : STACKOVERFLOW-PC
```

Classes et espaces de noms

Il existe de nombreuses classes disponibles dans CIM et WMI qui sont séparées en plusieurs espaces de noms. L'espace de noms le plus courant (et par défaut) dans Windows est `root/cimv2`. Pour trouver la bonne classe, il peut être utile de lister tout ou chercher.

Liste des classes disponibles

Vous pouvez répertorier toutes les classes disponibles dans l'espace de noms par défaut (`root/cimv2`) sur un ordinateur.

3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Rechercher un cours

Vous pouvez rechercher des classes spécifiques à l'aide de caractères génériques. Ex: Trouver des classes contenant le mot `process`.

3.0

CIM:

```
> Get-CimClass -ClassName "*Process*"

    Namespace: ROOT/CIMV2

CimClassName                CimClassMethods          CimClassProperties
-----
Win32_ProcessTrace          {}                        {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace     {}                        {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace      {}                        {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                 {}                        {Caption, Description, InstallDate,
Name...}
Win32_Process               {Create, Terminat... {Caption, Description, InstallDate,
Name...}
CIM_Processor               {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
Win32_Processor             {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
...
```

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

Liste des classes dans un autre espace de noms

L'espace de noms racine est simplement appelé `root` . Vous pouvez répertorier les classes dans un autre espace de noms à l'aide du paramètre `-Namespace` .

3.0

CIM:

```
> Get-CimClass -Namespace "root/SecurityCenter2"

    Namespace: ROOT/SecurityCenter2

CimClassName                CimClassMethods          CimClassProperties
-----
....
AntiSpywareProduct          {}                        {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
AntiVirusProduct           {}                        {displayName, instanceGuid,
pathToSignedProductExe, pathToSignedReportingE...
```

```
FirewallProduct          {}                               {displayName, instanceGuid,  
pathToSignedProductExe, pathToSignedReportingE...
```

WMI:

```
Get-WmiObject -Class "__Namespace" -Namespace "root"
```

Liste des espaces de noms disponibles

Pour rechercher des espaces de noms enfants disponibles de `root` (ou d'un autre espace de noms), interrogez les objets de la classe `__NAMESPACE` pour cet espace de noms.

3.0

CIM:

```
> Get-CimInstance -Namespace "root" -ClassName "__Namespace"
```

Name	PSComputerName
----	-----
subscription	
DEFAULT	
CIMV2	
msdtc	
Cli	
SECURITY	
HyperVCluster	
SecurityCenter2	
RSOP	
PEH	
StandardCimv2	
WMI	
directory	
Policy	
virtualization	
Interop	
Hardware	
ServiceModel	
SecurityCenter	
Microsoft	
aspnet	
Appv	

WMI:

```
Get-WmiObject -List -Namespace "root"
```

Lire WMI et CIM en ligne: <https://riptutorial.com/fr/powershell/topic/6808/wmi-et-cim>

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec PowerShell	4444 , autosvet , Brant Bobby , Chris N , Clijsters , Community , DarkLite1 , DAXaholic , Eitan , FoxDeploy , Gordon Bell , Greg Bray , Ian Miller , It-Z , JNYRanger , Jonas , Luboš Turek , Mark Wragg , Mathieu Buisson , Mrk , Nacimota , оswаl , Poorkenny , Sam Martin , th1rdey3 , TheIncorrigible1 , Tim , tjrobinson , TravisEz13 , vonPryz , Xalorous
2	Aide basée sur les commentaires	Christophe
3	Alias	jumbo
4	Analyse CSV	Andrei Epure , Frode F.
5	Anonymiser les adresses IP (v4 et v6) dans un fichier texte avec Powershell	NooJ
6	Application des prérequis du script	autosvet , Frode F. , jumbo , RamenChef
7	Automatisation de l'infrastructure	Giulio Caccin , Ranadip Dutta
8	Boucles	Blockhead , Christopher G. Lewis , Clijsters , CmdrTchort , DAXaholic , Eris , Frode F. , Gomibushi , Gordon Bell , Jay Bazuzi , Jon , jumbo , mákos , Poorkenny , Ranadip Dutta , Richard , Roman , SeeuD1 , Shawn Esterman , StephenP , TessellatingHeckler , TheIncorrigible1 , VertigoRay
9	Classes PowerShell	boeproxy , Brant Bobby , Frode F. , Jaqueline Vanek , Mert Gülsoy , Ranadip Dutta , xvorsx
10	Cmdlet Naming	TravisEz13
11	Comment télécharger le dernier artefact d'Artifactory en utilisant le script Powershell (v2.0 ou	ANIL

	inférieur)?	
12	Communication avec les API RESTful	autosvet , Clijsters , HAL9256 , kdtong , RamenChef , Ranadip Dutta , Sam Martin , YChi Lu
13	Communication TCP avec PowerShell	autosvet , RamenChef , Richard
14	Comportement de retour dans PowerShell	Bert Levrau , camilohe , Eris , jumbo , Ranadip Dutta , Thomas Gerot
15	Configuration d'état souhaitée	autosvet , CmdrTchort , Frode F. , RamenChef
16	Conventions de nommage	niksofteng
17	Cordes	Frode F. , restless1987 , void
18	Création de ressources DSC basées sur les classes	Trevor Sullivan
19	Déclaration de changement	Anthony Neace , Frode F. , jumbo , ошәј , Ranadip Dutta , TravisEz13
20	Envoi d'email	Adam M. , jimmyb , megamorf , NooJ , Ranadip Dutta , void , Yusuke Arakawa
21	Executables en cours d'exécution	RamenChef , W1M0R
22	Expressions régulières	Frode F.
23	Flux de travail PowerShell	Trevor Sullivan
24	Fonctions PowerShell	Bert Levrau , Eris , James Ruskin , Luke Ryan , niksofteng , Ranadip Dutta , Richard , TessellatingHeckler , TravisEz13 , Xalorous
25	Gestion des paquets	TravisEz13
26	Gestion des secrets et des informations d'identification	4444 , briantist , Ranadip Dutta , TravisEz13

27	GUI dans Powershell	Sam Martin
28	HashTables	Florian Meyer , Ranadip Dutta , TravisEz13
29	Incorporation de code géré (C # VB)	ajb101
30	Introduction à Pester	Frode F. , Sam Martin
31	Introduction à Psake	Roman
32	Jeux de paramètres	Bert Levrau , Poorkenny
33	La gestion des erreurs	Prageeth Saravanan
34	Les opérateurs	Anthony Neace , Bevo , Clijsters , Gordon Bell , JPBlanc , Mark Wragg , Ranadip Dutta
35	Ligne de commande PowerShell.exe	Frode F.
36	Logique conditionnelle	Liam , lloyd , miken32 , TravisEz13
37	Module ActiveDirectory	Lachie White
38	Module d'archive	James Ruskin , RapidCoder
39	Module de tâches planifiées	Sam Martin
40	Module ISE	Florian Meyer
41	Module SharePoint	Raziel
42	Modules Powershell	autosvet , Mike Shepard , TravisEz13 , Trevor Sullivan
43	Modules, scripts et fonctions	Frode F. , Ranadip Dutta , Xalorous
44	MongoDB	Thomas Gerot , Zteffer
45	Opérateurs spéciaux	TravisEz13
46	Opérations d'ensemble de base	Euro Micelli , Ranadip Dutta , TravisEz13
47	Paramètres communs	autosvet , jumbo , RamenChef

48	Paramètres dynamiques PowerShell	Poorkenny
49	Postes de travail PowerShell	Clijsters , mattnicola , Ranadip Dutta , Richard , TravisEz13
50	PowerShell "Streams"; Debug, Verbose, Warning, Error, Output et Information	DarkLite1 , Dave Anderson , megamorf
51	Powershell Remoting	Avshalom , megamorf , Moerwald , Sam Martin , ShaneC
52	Profils Powershell	Frode F. , Kolob Canyon
53	Propriétés calculées	Prageeth Saravanan
54	PSScriptAnalyzer - Analyseur de script PowerShell	Mark Wragg , mattnicola
55	Reconnaissance Amazon Web Services (AWS)	Trevor Sullivan
56	requêtes sql powershell	Venkatakrishnan
57	Scripts de signature	AP. , Frode F.
58	Sécurité et cryptographie	YChi Lu
59	Service de stockage simple Amazon Web Services (AWS) (S3)	Trevor Sullivan
60	Splatting	autosvet , Frode F. , Moerwald , Petru Zaharia , Poorkenny , RamenChef , Ranadip Dutta , TravisEz13 , xXhRQ8sD2L7Z
61	Travailler avec des fichiers XML	autosvet , Frode F. , Giorgio Gambino , Lieven Keersmaekers , RamenChef , Richard , Rowshi
62	Travailler avec des objets	Chris N , djwork , Mathieu Buisson , megamorf
63	Travailler avec le pipeline PowerShell	Alban , Atsch , Clijsters , Deptor , James Ruskin , Keith , оωωε , Sam Martin

64	URL encoder / décodeur	VertigoRay
65	Utilisation du système d'aide	Frode F. , Madniz , mattnicola , RamenChef
66	Utiliser des classes statiques existantes	Austin T French , briantist , motcke , Ranadip Dutta , Xenophane
67	Utiliser la barre de progression	Clijsters , jumbo , Ranadip Dutta
68	Utiliser ShouldProcess	Brant Bobby , Charlie Joynt , Schwarzie2478
69	Variables Automatiques	Brant Bobby , jumbo , Mateusz Piotrowski , Moerwald , Ranadip Dutta , Roman
70	Variables automatiques - partie 2	Roman
71	Variables d'environnement	autosvet
72	Variables dans PowerShell	autosvet , Eris , Liam , Prageeth Saravanan , Ranadip Dutta , restless1987 , Steve K
73	Variables intégrées	Trevor Sullivan
74	WMI et CIM	Frode F.