



EBook Gratuito

APPENDIMENTO

PowerShell

Free unaffiliated eBook created from
Stack Overflow contributors.

#powershell

Sommario

| | |
|---|-----------|
| Di..... | 1 |
| Capitolo 1: Introduzione a PowerShell..... | 2 |
| Osservazioni..... | 2 |
| Versioni..... | 2 |
| Examples..... | 2 |
| Installazione o configurazione..... | 2 |
| finestre..... | 2 |
| Altre piattaforme..... | 3 |
| Permetti agli script memorizzati sul tuo computer di essere eseguiti senza firma..... | 3 |
| Alias e funzioni simili..... | 4 |
| La pipeline: utilizzo dell'output da un cmdlet di PowerShell..... | 5 |
| Commentando..... | 6 |
| Chiamare i metodi della libreria .Net..... | 6 |
| Creazione di oggetti..... | 7 |
| Capitolo 2: alias..... | 9 |
| Osservazioni..... | 9 |
| Examples..... | 10 |
| Get-Alias..... | 10 |
| Set-Alias..... | 10 |
| Capitolo 3: Analisi CSV..... | 12 |
| Examples..... | 12 |
| Utilizzo di base di Import-Csv..... | 12 |
| Importa da CSV e cast le proprietà per correggere il tipo..... | 12 |
| Capitolo 4: Anonimizza IP (v4 e v6) nel file di testo con PowerShell..... | 14 |
| introduzione..... | 14 |
| Examples..... | 14 |
| Anonimizza l'indirizzo IP nel file di testo..... | 14 |
| Capitolo 5: Automazione dell'infrastruttura..... | 16 |
| introduzione..... | 16 |
| Examples..... | 16 |

| | |
|---|-----------|
| Script semplice per il test di integrazione black-box delle applicazioni della console | 16 |
| Capitolo 6: Classi di PowerShell | 17 |
| introduzione | 17 |
| Examples | 17 |
| Metodi e proprietà | 17 |
| Elenco dei costruttori disponibili per una classe | 17 |
| Sovraccarico del costruttore | 19 |
| Ottieni tutti i membri di un'istanza | 19 |
| Modello di classe base | 19 |
| Ereditarietà dalla classe padre alla classe figlio | 20 |
| Capitolo 7: Codifica / decodifica URL | 21 |
| Osservazioni | 21 |
| Examples | 21 |
| Avvio rapido: codifica | 21 |
| Avvio rapido: decodifica | 22 |
| Codifica stringa di query con `[uri] :: EscapeDataString ()` | 22 |
| Codifica stringa di query con `[System.Web.HttpUtility] :: UriEncode ()` | 23 |
| Decodifica URL con `[uri] :: UnescapeDataString ()` | 23 |
| Decodifica l'URL con `[System.Web.HttpUtility] :: UriDecode ()` | 25 |
| Capitolo 8: Come scaricare l'ultimo artefatto da Artifactory usando lo script Powershell (..... | 28 |
| introduzione | 28 |
| Examples | 28 |
| Script di PowerShell per scaricare l'ultimo artifat | 28 |
| Capitolo 9: Comportamento di restituzione in PowerShell | 29 |
| introduzione | 29 |
| Osservazioni | 29 |
| Examples | 29 |
| uscita anticipata | 29 |
| Gotcha! Rientro in pipeline | 30 |
| Gotcha! Ignorando l'uscita indesiderata | 30 |
| Ritorna con un valore | 30 |
| Come lavorare con i ritorni delle funzioni | 31 |

| | |
|--|-----------|
| Capitolo 10: Comunicazione con API RESTful | 33 |
| introduzione | 33 |
| Examples | 33 |
| Usa i Webhook in arrivo su Slack.com | 33 |
| Invia un messaggio a hipChat | 33 |
| Utilizzo di REST con oggetti PowerShell per ottenere e inserire singoli dati | 33 |
| Utilizzare REST con oggetti PowerShell per OTTENERE e POSTARE molti articoli | 34 |
| Utilizzare REST con PowerShell per eliminare elementi | 34 |
| Capitolo 11: Comunicazione TCP con PowerShell | 35 |
| Examples | 35 |
| Ascoltatore TCP | 35 |
| TCP Sender | 35 |
| Capitolo 12: Configurazione dello stato desiderata | 37 |
| Examples | 37 |
| Semplice esempio: abilitazione di WindowsFeature | 37 |
| Avvio di DSC (mof) sul computer remoto | 37 |
| Importazione di psd1 (file di dati) nella variabile locale | 37 |
| Elenca le risorse DSC disponibili | 38 |
| Importazione di risorse da utilizzare in DSC | 38 |
| Capitolo 13: Convenzioni di denominazione | 39 |
| Examples | 39 |
| funzioni | 39 |
| Capitolo 14: Creazione di risorse basate su classi DSC | 40 |
| introduzione | 40 |
| Osservazioni | 40 |
| Examples | 40 |
| Creare una classe di scheletro delle risorse DSC | 40 |
| Scheletro di risorse DSC con proprietà chiave | 40 |
| Risorsa DSC con proprietà obbligatorie | 41 |
| Risorsa DSC con metodi richiesti | 41 |
| Capitolo 15: Esecuzione di eseguibili | 43 |
| Examples | 43 |

| | |
|--|-----------|
| Applicazioni console..... | 43 |
| Applicazioni GUI..... | 43 |
| Stream console..... | 43 |
| Codici di uscita..... | 44 |
| Capitolo 16: Espressioni regolari..... | 45 |
| Sintassi..... | 45 |
| Examples..... | 45 |
| Partita singola..... | 45 |
| Utilizzando l'operatore -Match..... | 45 |
| Utilizzando Select-String..... | 46 |
| Utilizzo di [Regex] :: Match ()..... | 47 |
| Sostituire..... | 47 |
| Utilizzando -Riposare l'operatore..... | 47 |
| Utilizzo del metodo [Regex]: Sostituisci ()..... | 48 |
| Sostituisci il testo con il valore dinamico usando un valore MatchEvaluator..... | 48 |
| Sfuggire personaggi speciali..... | 49 |
| Più partite..... | 49 |
| Utilizzando Select-String..... | 50 |
| Utilizzo di [Regex] :: Matches ()..... | 50 |
| Capitolo 17: Flussi di lavoro di PowerShell..... | 52 |
| introduzione..... | 52 |
| Osservazioni..... | 52 |
| Examples..... | 52 |
| Esempio di flusso di lavoro semplice..... | 52 |
| Flusso di lavoro con parametri di input..... | 52 |
| Esegui il flusso di lavoro come lavoro in background..... | 53 |
| Aggiungi un blocco parallelo a un flusso di lavoro..... | 53 |
| Capitolo 18: Funzioni di PowerShell..... | 54 |
| introduzione..... | 54 |
| Examples..... | 54 |
| Funzione semplice senza parametri..... | 54 |

| | |
|--|-----------|
| Parametri di base..... | 54 |
| Parametri obbligatori..... | 55 |
| Funzione avanzata..... | 56 |
| Validazione dei parametri..... | 57 |
| ValidateSet..... | 57 |
| ValidateRange..... | 58 |
| ValidatePattern..... | 58 |
| ValidateLength..... | 58 |
| ValidateCount..... | 58 |
| ValidateScript..... | 58 |
| Capitolo 19: Gestione degli errori..... | 60 |
| introduzione..... | 60 |
| Examples..... | 60 |
| Tipi di errore..... | 60 |
| Capitolo 20: Gestione dei pacchetti..... | 62 |
| introduzione..... | 62 |
| Examples..... | 62 |
| Trova un modulo PowerShell usando un modello..... | 62 |
| Creare la Reponsibilità del modulo PowerShell di default..... | 62 |
| Trova un modulo per nome..... | 62 |
| Installa un modulo per nome..... | 62 |
| Disinstallare un modulo il mio nome e versione..... | 62 |
| Aggiorna un modulo per nome..... | 62 |
| Capitolo 21: Gestione di segreti e credenziali..... | 64 |
| introduzione..... | 64 |
| Examples..... | 64 |
| Richiesta di credenziali..... | 64 |
| Accesso alla password in chiaro..... | 64 |
| Lavorare con le credenziali memorizzate..... | 64 |
| Encrypter..... | 65 |
| Il codice che utilizza le credenziali archiviate:..... | 65 |
| Memorizzazione delle credenziali in forma crittografata e passaggio come parametro quando..... | 65 |

| | |
|---|-----------|
| Capitolo 22: GUI in Powershell | 67 |
| Examples | 67 |
| GUI WPF per il cmdlet Get-Service | 67 |
| Capitolo 23: Guida basata sui commenti | 69 |
| introduzione | 69 |
| Examples | 69 |
| Guida basata sui commenti | 69 |
| Guida agli script basata su commenti | 71 |
| Capitolo 24: hashtables | 74 |
| introduzione | 74 |
| Osservazioni | 74 |
| Examples | 74 |
| Creazione di una tabella hash | 74 |
| Accedere a un valore di tabella hash per chiave | 74 |
| Fare il ciclo su un tavolo hash | 75 |
| Aggiungi una coppia di valori chiave a una tabella hash esistente | 75 |
| Enumerazione tramite chiavi e coppie valore-chiave | 75 |
| Rimuovere una coppia di valori chiave da una tabella hash esistente | 76 |
| Capitolo 25: Imporre i prerequisiti di script | 77 |
| Sintassi | 77 |
| Osservazioni | 77 |
| Examples | 77 |
| Applicare la versione minima dell'host PowerShell | 77 |
| Imponi eseguire lo script come amministratore | 77 |
| Capitolo 26: Incorporare codice gestito (C # VB) | 79 |
| introduzione | 79 |
| Parametri | 79 |
| Osservazioni | 79 |
| Rimozione dei tipi aggiunti | 79 |
| Sintassi CSharp e .NET | 79 |
| Examples | 80 |

| | |
|---|-----------|
| C # Esempio..... | 80 |
| Esempio VB.NET..... | 80 |
| Capitolo 27: Introduzione a Pester..... | 82 |
| Osservazioni..... | 82 |
| Examples..... | 82 |
| Iniziare con Pester..... | 82 |
| Capitolo 28: Introduzione a Psake..... | 84 |
| Sintassi..... | 84 |
| Osservazioni..... | 84 |
| Examples..... | 84 |
| Schema di base..... | 84 |
| FormatTaskName esempio..... | 84 |
| Esegui l'attività in modo condizionale..... | 85 |
| ContinueOnError..... | 85 |
| Capitolo 29: Invio di email..... | 86 |
| introduzione..... | 86 |
| Parametri..... | 86 |
| Examples..... | 87 |
| Send-MailMessage semplice..... | 87 |
| Send-MailMessage con parametri predefiniti..... | 87 |
| SMTPClient - Posta con file .txt nel messaggio del corpo..... | 88 |
| Capitolo 30: Lavorare con gli oggetti..... | 89 |
| Examples..... | 89 |
| Aggiornamento degli oggetti..... | 89 |
| Aggiungere proprietà..... | 89 |
| Rimozione delle proprietà..... | 89 |
| Creare un nuovo oggetto..... | 90 |
| Opzione 1: nuovo oggetto..... | 90 |
| Opzione 2: Select-Object..... | 90 |
| Opzione 3: acceleratore di tipo pscustomobject (richiesto PSV3 +)..... | 91 |
| Esaminando un oggetto..... | 91 |

| | |
|---|------------|
| Creazione di istanze di classi generiche..... | 92 |
| Capitolo 31: Lavorare con i file XML..... | 94 |
| Examples..... | 94 |
| Accesso a un file XML..... | 94 |
| Creare un documento XML usando XmlWriter ()..... | 96 |
| Aggiunta di snippets di XML a XmlDocument corrente..... | 97 |
| Dati di esempio..... | 97 |
| Documento XML..... | 97 |
| Nuovi dati..... | 98 |
| Modelli..... | 99 |
| Aggiungere i nuovi dati..... | 99 |
| Profitto..... | 101 |
| miglioramenti..... | 101 |
| Capitolo 32: Lavorare con la pipeline di PowerShell..... | 102 |
| introduzione..... | 102 |
| Sintassi..... | 102 |
| Osservazioni..... | 102 |
| Examples..... | 102 |
| Funzioni di scrittura con ciclo di vita avanzato..... | 103 |
| Supporto per pipeline di base in funzioni..... | 103 |
| Concetto di lavoro della pipeline..... | 104 |
| Capitolo 33: Lavori in background di PowerShell..... | 105 |
| introduzione..... | 105 |
| Osservazioni..... | 105 |
| Examples..... | 105 |
| Creazione di lavoro di base..... | 105 |
| Gestione del lavoro di base..... | 106 |
| Capitolo 34: Logica condizionale..... | 108 |
| Sintassi..... | 108 |
| Osservazioni..... | 108 |
| Examples..... | 108 |

| | |
|---|------------|
| se, altro e altro se | 108 |
| Negazione | 109 |
| Se stenografia condizionale | 109 |
| Capitolo 35: Loops | 111 |
| introduzione | 111 |
| Sintassi | 111 |
| Osservazioni | 111 |
| Per ciascuno | 111 |
| Prestazione | 112 |
| Examples | 112 |
| Per | 112 |
| Per ciascuno | 112 |
| Mentre | 113 |
| ForEach-Object | 113 |
| Utilizzo di base | 114 |
| Utilizzo avanzato | 114 |
| Fare | 115 |
| Metodo ForEach () | 115 |
| Continua | 116 |
| Rompere | 116 |
| Capitolo 36: Moduli PowerShell | 118 |
| introduzione | 118 |
| Examples | 118 |
| Crea un modulo manifest | 118 |
| Esempio di modulo semplice | 118 |
| Esportare una variabile da un modulo | 119 |
| Strutturazione dei moduli PowerShell | 119 |
| Posizione dei moduli | 120 |
| Visibilità membro del modulo | 120 |
| Capitolo 37: Moduli, script e funzioni | 121 |
| introduzione | 121 |
| Examples | 121 |

| | |
|--|------------|
| Funzione..... | 121 |
| dimostrazione..... | 121 |
| copione..... | 122 |
| dimostrazione..... | 122 |
| Modulo..... | 123 |
| dimostrazione..... | 123 |
| Funzioni avanzate..... | 123 |
| Capitolo 38: Modulo ActiveDirectory..... | 127 |
| introduzione..... | 127 |
| Osservazioni..... | 127 |
| Examples..... | 127 |
| Modulo..... | 127 |
| utenti..... | 127 |
| gruppi..... | 128 |
| computers..... | 128 |
| Oggetti..... | 128 |
| Capitolo 39: Modulo delle attività pianificate..... | 130 |
| introduzione..... | 130 |
| Examples..... | 130 |
| Esegui script PowerShell in Attività pianificata..... | 130 |
| Capitolo 40: Modulo di archiviazione..... | 131 |
| introduzione..... | 131 |
| Sintassi..... | 131 |
| Parametri..... | 131 |
| Osservazioni..... | 132 |
| Examples..... | 132 |
| Comprimi-archivio con carattere jolly..... | 132 |
| Aggiorna ZIP esistente con Compress-Archive..... | 132 |
| Estrai un file zip con Expand-Archive..... | 132 |
| Capitolo 41: Modulo di SharePoint..... | 133 |
| Examples..... | 133 |
| Caricamento Snap-in di SharePoint..... | 133 |

| | |
|--|------------|
| Iterazione su tutti gli elenchi di una raccolta siti | 133 |
| Otteni tutte le funzionalità installate in una raccolta siti | 133 |
| Capitolo 42: Modulo ISE | 135 |
| introduzione | 135 |
| Examples | 135 |
| Script di prova | 135 |
| Capitolo 43: MongoDB | 136 |
| Osservazioni | 136 |
| Examples | 136 |
| MongoDB con il driver C # 1.7 usando PowerShell | 136 |
| Ho 3 set di array in PowerShell | 136 |
| Capitolo 44: Nome del cmdlet | 138 |
| introduzione | 138 |
| Examples | 138 |
| verbi | 138 |
| I sostantivi | 138 |
| Capitolo 45: operatori | 139 |
| introduzione | 139 |
| Examples | 139 |
| Operatori aritmetici | 139 |
| Operatori logici | 139 |
| Operatori di assegnazione | 139 |
| Operatori di confronto | 140 |
| Operatori di reindirizzamento | 140 |
| Tipi di operando di messaggio: il tipo dell'operando sinistro determina il comportamento | 141 |
| Operatori di manipolazione delle stringhe | 142 |
| Capitolo 46: Operatori speciali | 143 |
| Examples | 143 |
| Operatore di espressione di array | 143 |
| Operazione di chiamata | 143 |
| Operatore di acquisizione punti | 143 |
| Capitolo 47: Operazioni di base | 144 |

| | |
|---|------------|
| introduzione..... | 144 |
| Sintassi..... | 144 |
| Examples..... | 144 |
| Filtro: dove-oggetto / dove /?..... | 144 |
| Ordinamento: Sort-Object / sort..... | 145 |
| Raggruppamento: gruppo-oggetto / gruppo..... | 146 |
| Proiezione: Seleziona-Oggetto / seleziona..... | 146 |
| Capitolo 48: Parametri comuni..... | 149 |
| Osservazioni..... | 149 |
| Examples..... | 149 |
| Parametro ErrorAction..... | 149 |
| -ErrorAction Continua..... | 149 |
| -ErrorAction Ignora..... | 149 |
| -ErrorAction Informarsi..... | 150 |
| -ErrorAction SilentlyContinue..... | 150 |
| -ErroreAzione Stop..... | 150 |
| -ErrorAction Suspend..... | 151 |
| Capitolo 49: Passare la dichiarazione..... | 152 |
| introduzione..... | 152 |
| Osservazioni..... | 152 |
| Examples..... | 152 |
| Interruttore semplice..... | 152 |
| Switch Statement con Regex Parameter..... | 152 |
| Interruttore semplice con pausa..... | 153 |
| Passa istruzione con parametro jolly..... | 153 |
| Passa istruzione con parametro esatto..... | 154 |
| Switch Statement con parametro CaseSensitive..... | 154 |
| Passa l'istruzione con il parametro File..... | 155 |
| Interruttore semplice con condizione predefinita..... | 155 |
| Cambia istruzione con le espressioni..... | 156 |
| Capitolo 50: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output e Information..... | 157 |

| | |
|--|------------|
| Osservazioni..... | 157 |
| Examples..... | 157 |
| Write-Output..... | 157 |
| Scrivi Preferenze..... | 157 |
| Capitolo 51: PowerShell Parametri dinamici..... | 159 |
| Examples..... | 159 |
| Parametro dinamico "semplice"..... | 159 |
| Capitolo 52: Profili PowerShell..... | 161 |
| Osservazioni..... | 161 |
| Examples..... | 162 |
| Crea un profilo di base..... | 162 |
| Capitolo 53: Proprietà calcolate..... | 163 |
| introduzione..... | 163 |
| Examples..... | 163 |
| Visualizza le dimensioni del file in KB - Proprietà calcolate..... | 163 |
| Capitolo 54: PSScriptAnalyzer - PowerShell Script Analyzer..... | 164 |
| introduzione..... | 164 |
| Sintassi..... | 164 |
| Examples..... | 164 |
| Analizzare gli script con i set di regole predefiniti incorporati..... | 164 |
| Analizzare gli script contro ogni regola incorporata..... | 165 |
| Elenca tutte le regole integrate..... | 165 |
| Capitolo 55: query sql PowerShell..... | 166 |
| introduzione..... | 166 |
| Parametri..... | 166 |
| Osservazioni..... | 166 |
| Examples..... | 168 |
| SQLExample..... | 168 |
| SQLQuery..... | 168 |
| Capitolo 56: Remot PowerShell..... | 170 |
| Osservazioni..... | 170 |

| | |
|--|------------|
| Examples..... | 170 |
| Abilitazione di Remot PowerShell..... | 170 |
| Solo per ambienti non di dominio..... | 170 |
| Abilitazione dell'autenticazione di base..... | 171 |
| Connessione a un server remoto tramite PowerShell..... | 171 |
| Esegui comandi su un computer remoto..... | 171 |
| Avviso di serializzazione remota..... | 172 |
| Uso di argomenti..... | 173 |
| Una best practice per la pulizia automatica delle PSSession..... | 173 |
| Capitolo 57: Riconoscimento di Amazon Web Services (AWS)..... | 175 |
| introduzione..... | 175 |
| Examples..... | 175 |
| Rileva etichette immagine con Rekognition AWS..... | 175 |
| Confrontare la somiglianza facciale con il rekognition AWS..... | 176 |
| Capitolo 58: Riga di comando PowerShell.exe..... | 177 |
| Parametri..... | 177 |
| Examples..... | 178 |
| Esecuzione di un comando..... | 178 |
| -Command <stringa>..... | 178 |
| -Command {scriptblock}..... | 178 |
| -Command - (input standard)..... | 178 |
| Esecuzione di un file di script..... | 179 |
| Script di base..... | 179 |
| Usando parametri e argomenti..... | 179 |
| Capitolo 59: Script di firma..... | 181 |
| Osservazioni..... | 181 |
| Politiche di esecuzione..... | 181 |
| Examples..... | 182 |
| Firma di uno script..... | 182 |
| Modifica della politica di esecuzione tramite Set-ExecutionPolicy..... | 182 |
| Bypassare i criteri di esecuzione per un singolo script..... | 182 |

| | |
|---|------------|
| Altre politiche di esecuzione: | 183 |
| Ottieni la politica di esecuzione corrente..... | 183 |
| Ottendere la firma da uno script firmato..... | 184 |
| Creazione di un certificato di firma del codice autofirmato per il test..... | 184 |
| Capitolo 60: Servizio di archiviazione semplice Amazon Web Services (AWS) (S3) | 185 |
| introduzione..... | 185 |
| Parametri..... | 185 |
| Examples..... | 185 |
| Crea un nuovo S3 Bucket..... | 185 |
| Carica un file locale in un bucket S3..... | 185 |
| Elimina un bucket S3..... | 186 |
| Capitolo 61: Set di parametri | 187 |
| introduzione..... | 187 |
| Examples..... | 187 |
| Set di parametri semplici..... | 187 |
| Parameterset per forzare l'uso di un parametro quando viene selezionato un altro..... | 187 |
| Parametro impostato per limitare la combinazione di parametri..... | 188 |
| Capitolo 62: Sicurezza e crittografia | 189 |
| Examples..... | 189 |
| Calcolo dei codici hash di una stringa tramite .Net Cryptography..... | 189 |
| Capitolo 63: Splatting | 190 |
| introduzione..... | 190 |
| Osservazioni..... | 190 |
| Examples..... | 190 |
| Splattare i parametri..... | 190 |
| Passare un parametro Switch usando Splatting..... | 191 |
| Piping e Splatting..... | 191 |
| Splatting dalla funzione di primo livello a una serie di funzioni interne..... | 191 |
| Capitolo 64: stringhe | 193 |
| Sintassi..... | 193 |
| Osservazioni..... | 193 |

| | |
|---|------------|
| Examples..... | 193 |
| Creare una stringa di base..... | 193 |
| Stringa..... | 193 |
| Stringa letterale..... | 193 |
| Formato stringa..... | 194 |
| Stringa multilinea..... | 194 |
| Qui-string..... | 194 |
| Qui-string..... | 194 |
| Stringa qui letterale..... | 195 |
| Concatenazione di stringhe..... | 195 |
| Uso delle variabili in una stringa..... | 195 |
| Usando l'operatore +..... | 195 |
| Utilizzo di sottoespressioni..... | 195 |
| Personaggi speciali..... | 196 |
| Capitolo 65: Usando la barra di avanzamento..... | 197 |
| introduzione..... | 197 |
| Examples..... | 197 |
| Semplice utilizzo della barra di avanzamento..... | 197 |
| Utilizzo della barra di avanzamento interna..... | 198 |
| Capitolo 66: Utilizzando le classi statiche esistenti..... | 200 |
| introduzione..... | 200 |
| Examples..... | 200 |
| Creazione immediata di nuovo GUID..... | 200 |
| Utilizzo della classe Math .Net..... | 200 |
| Aggiungere tipi..... | 201 |
| Capitolo 67: Utilizzando ShouldProcess..... | 202 |
| Sintassi..... | 202 |
| Parametri..... | 202 |
| Osservazioni..... | 202 |
| Examples..... | 202 |
| Aggiunta di -WhatIf e -Confermare il supporto per il cmdlet..... | 202 |

| | |
|---|------------|
| Utilizzare ShouldProcess () con un argomento..... | 202 |
| Esempio di utilizzo completo..... | 203 |
| Capitolo 68: Utilizzo del sistema di guida..... | 205 |
| Osservazioni..... | 205 |
| Examples..... | 205 |
| Aggiornamento del sistema di guida..... | 205 |
| Utilizzando Get-Help..... | 205 |
| Visualizzazione della versione online di un argomento della guida..... | 206 |
| Esempi di visualizzazione..... | 206 |
| Visualizzazione della pagina della Guida completa..... | 206 |
| Visualizzazione della guida per un parametro specifico..... | 206 |
| Capitolo 69: variabili ambientali..... | 207 |
| Examples..... | 207 |
| Le variabili di ambiente di Windows sono visibili come un drive PS chiamato Env:..... | 207 |
| Chiamata istantanea di variabili d'ambiente con \$ env:..... | 207 |
| Capitolo 70: Variabili automatiche..... | 208 |
| introduzione..... | 208 |
| Sintassi..... | 208 |
| Examples..... | 208 |
| \$ pid..... | 208 |
| Valori booleani..... | 208 |
| \$ null..... | 208 |
| \$ OFS..... | 209 |
| \$ _ / \$ PSItem..... | 209 |
| \$?..... | 210 |
| \$ error..... | 210 |
| Capitolo 71: Variabili automatiche - parte 2..... | 212 |
| introduzione..... | 212 |
| Osservazioni..... | 212 |
| Examples..... | 212 |
| \$ PSVersionTable..... | 212 |
| Capitolo 72: Variabili in PowerShell..... | 213 |

| | |
|--|------------|
| introduzione..... | 213 |
| Examples..... | 213 |
| Variabile semplice..... | 213 |
| Rimozione di una variabile..... | 213 |
| Scopo..... | 213 |
| Lettura di un output CmdLet..... | 214 |
| Assegna lista di variabili multiple..... | 215 |
| Array..... | 215 |
| Aggiungendo a un array..... | 216 |
| Combinare insieme gli array..... | 216 |
| Capitolo 73: Variabili integrate..... | 217 |
| introduzione..... | 217 |
| Examples..... | 217 |
| \$ PSScriptRoot..... | 217 |
| \$ Args..... | 217 |
| \$ PSItem..... | 217 |
| \$?..... | 218 |
| \$ error..... | 218 |
| Capitolo 74: WMI e CIM..... | 219 |
| Osservazioni..... | 219 |
| CIM vs WMI..... | 219 |
| Risorse aggiuntive..... | 220 |
| Examples..... | 220 |
| Interrogare oggetti..... | 220 |
| Elenca tutti gli oggetti per la classe CIM..... | 220 |
| Utilizzando un filtro..... | 220 |
| Utilizzando una query WQL:..... | 221 |
| Classi e spazi dei nomi..... | 222 |
| Elenca le classi disponibili..... | 222 |
| Cerca una classe..... | 222 |
| Elenca le classi in un diverso spazio dei nomi..... | 223 |

Elenca gli spazi dei nomi disponibili.....224

Titoli di coda.....226

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

Capitolo 1: Introduzione a PowerShell

Osservazioni

Windows PowerShell è un componente di shell e script di Windows Management Framework, un framework di gestione di automazione / configurazione di Microsoft basato su .NET Framework. PowerShell viene installato per impostazione predefinita su tutte le versioni supportate dei sistemi operativi client e server Windows da Windows 7 / Windows Server 2008 R2. Powershell può essere aggiornato in qualsiasi momento scaricando una versione successiva di [Windows Management Framework](#) (WMF). La versione "Alpha" di PowerShell 6 è multiplatforma (Windows, Linux e OS X) e deve essere scaricata e installata da [questa pagina di rilascio](#) .

Risorse aggiuntive:

- Documentazione MSDN: <https://msdn.microsoft.com/en-us/powershell/scripting/powershell-scripting>
- TechNet: <https://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx>
 - [Informazioni sulle pagine](#)
- Galleria PowerShell: <https://www.powershellgallery.com/>
- Blog MSDN: <https://blogs.msdn.microsoft.com/powershell/>
- Github: <https://github.com/powershell>
- Sito comunitario: <http://powershell.com/cs/>

Versioni

| Versione | Incluso con Windows | Gli appunti | Data di rilascio |
|---------------------|--------------------------------------|-------------|------------------|
| 1.0 | XP / Server 2008 | | 2006-11-01 |
| 2.0 | 7 / Server 2008 R2 | | 2009-11-01 |
| 3.0 | 8 / Server 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 Anniversary edition / Server 2016 | | 2017/01/27 |

Examples

Installazione o configurazione

finestre

PowerShell è incluso in Windows Management Framework. L'installazione e l'installazione non sono richieste sulle versioni moderne di Windows.

Gli aggiornamenti a PowerShell possono essere eseguiti installando una versione più recente di Windows Management Framework.

Altre piattaforme

La versione "Beta" di PowerShell 6 può essere installata su altre piattaforme. I pacchetti di installazione sono disponibili [qui](#).

Ad esempio, PowerShell 6, per Ubuntu 16.04, viene pubblicato per il pacchetto di repository per una facile installazione (e aggiornamenti).

Per installare eseguire quanto segue:

```
# 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
```

Dopo aver registrato il repository Microsoft una volta come superutente, da quel momento in poi, sarà sufficiente utilizzare `sudo apt-get upgrade powershell` per aggiornarlo. Quindi esegui `powershell`

Permetti agli script memorizzati sul tuo computer di essere eseguiti senza firma

Per motivi di sicurezza, PowerShell è configurato per impostazione predefinita per consentire solo l'esecuzione di script firmati. L'esecuzione del comando seguente consentirà di eseguire script non firmati (per eseguire questa operazione è necessario eseguire PowerShell come amministratore).

```
Set-ExecutionPolicy RemoteSigned
```

Un altro modo per eseguire script PowerShell consiste nell'utilizzare `Bypass` come `ExecutionPolicy` :

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

Oppure dalla tua console PowerShell o sessione ISE esistente eseguendo:

```
Set-ExecutionPolicy Bypass Process
```

È inoltre possibile eseguire una soluzione temporanea per i criteri di esecuzione eseguendo l'eseguibile di PowerShell e passando qualsiasi criterio valido come parametro `-ExecutionPolicy`. Il criterio è valido solo durante la vita del processo, quindi non è necessario alcun accesso amministrativo al registro.

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

Sono disponibili più altre politiche e i siti online spesso incoraggiano l'utilizzo di `Set-ExecutionPolicy Unrestricted`. Questa politica rimane in vigore fino a quando non viene modificata e riduce la sicurezza del sistema. Questo non è consigliabile. L'uso di `RemoteSigned` è consigliato perché consente il codice memorizzato e scritto localmente e richiede che il codice acquisito a distanza venga firmato con un certificato da una radice attendibile.

Inoltre, fare attenzione che la politica di esecuzione possa essere applicata dai Criteri di gruppo, in modo che, anche se la politica viene modificata su `Non Unrestricted` sistema, i Criteri di gruppo possono ripristinare tale impostazione al successivo intervallo di applicazione (in genere 15 minuti). È possibile visualizzare il criterio di esecuzione impostato nei vari ambiti utilizzando `Get-ExecutionPolicy -List`

Documentazione TechNet:

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

Alias e funzioni simili

In PowerShell, ci sono molti modi per ottenere lo stesso risultato. Questo può essere illustrato bene con l'esempio `Hello World` semplice e familiare:

Utilizzando l' `Write-Host` :

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

Uso di `Write-Output` :

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

Vale la pena notare che anche se `Write-Output` e `Write-Host` scrivono entrambi sullo schermo c'è una sottile differenza. `Write-Host` scrive *solo* su stdout (cioè sullo schermo della console), mentre `Write-Output` scrive sia su stdout *AND* sul flusso di output [successo] che consente il [reindirizzamento](#). Il reindirizzamento (e gli stream in generale) consentono di indirizzare l'output di un comando come input a un altro incluso l'assegnazione a una variabile.

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

```
"Hello World"
```

Queste funzioni simili non sono alias, ma possono produrre gli stessi risultati se si vuole evitare di "inquinare" il flusso di successo.

Write-Output è aliato in Echo o Write

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

Oppure, semplicemente digitando "Hello world"!

```
'Hello world'
```

Tutto ciò si tradurrà nell'output della console previsto

```
Hello world
```

Un altro esempio di alias in PowerShell è rappresentato dalla mappatura comune di comandi e comandi BASH del comando precedenti e dei cmdlet di PowerShell. Tutti i seguenti producono un elenco di directory della directory corrente.

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

Infine, puoi creare il tuo alias con il cmdlet Set-Alias! Ad esempio, `Test-NetConnection`, che è essenzialmente l'equivalente di PowerShell al comando ping del prompt dei comandi, a "ping".

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

Ora puoi usare `ping` invece di `Test-NetConnection` ! Tieni presente che se l'alias è già in uso, sovrascrivi l'associazione.

L'alias sarà vivo finché la sessione non sarà attiva. Una volta che si chiude la sessione e si tenta di eseguire l'alias creato nell'ultima sessione, non funzionerà. Per superare questo problema, puoi importare tutti i tuoi alias da un excel nella tua sessione una volta, prima di iniziare il tuo lavoro.

La pipeline: utilizzo dell'output da un cmdlet di PowerShell

Una delle prime domande che gli utenti hanno quando iniziano a utilizzare PowerShell per lo scripting è come manipolare l'output da un cmdlet per eseguire un'altra azione.

Il simbolo del gasdotto | viene utilizzato alla fine di un cmdlet per raccogliere i dati che esporta e alimentarlo al cmdlet successivo. Un semplice esempio utilizza Select-Object per mostrare solo la proprietà Name di un file mostrato da Get-ChildItem:

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

```
#This may be shortened to:
gci | Select Name
```

L'utilizzo più avanzato della pipeline ci consente di convogliare l'output di un cmdlet in un ciclo foreach:

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

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

Si noti che l'esempio sopra utilizza la variabile automatica `$_`. `$_` è l'alias breve di `$PSItem` che è una variabile automatica che contiene l'elemento corrente nella pipeline.

Commentando

Per commentare gli script di potenza, anteporre la riga usando il simbolo `#` (cancelletto)

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

Puoi anche avere commenti su più righe usando rispettivamente `<#` e `#>` all'inizio e alla fine del commento.

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

Chiamare i metodi della libreria .Net

I metodi della libreria statica .Net possono essere richiamati da PowerShell incapsulando il nome completo della classe nella terza parentesi e chiamando il metodo utilizzando `::`

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

I metodi statici possono essere chiamati dalla classe stessa, ma i metodi non statici richiedono un'istanza della classe .Net (un oggetto).

Ad esempio, il metodo `AddHours` non può essere chiamato dalla classe `System.DateTime`. Richiede un'istanza della 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
```

In questo caso, per prima cosa **creiamo un oggetto** , ad esempio:

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

Quindi, possiamo usare i metodi di quell'oggetto, anche i metodi che non possono essere chiamati direttamente dalla classe `System.DateTime`, come il metodo `AddHours`:

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

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

Creazione di oggetti

Il cmdlet `New-Object` viene utilizzato per creare un oggetto.

```
# 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"
```

In molti casi, verrà creato un nuovo oggetto per esportare i dati o passarli a un altro comando. Questo può essere fatto in questo modo:

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

Esistono molti modi per creare un oggetto. Il seguente metodo è probabilmente il modo più breve e veloce per creare un oggetto `PSCustomObject` :

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

Se hai già un oggetto, ma hai solo bisogno di una o due proprietà extra, puoi semplicemente aggiungere quella proprietà usando `Select-Object` :

```
Get-ChildItem | Select-Object FullName, Name,
    @{Name='DateTime'; Expression={Get-Date}},
    @{Name='PropertieName'; Expression={'CustomValue'}}
```

Tutti gli oggetti possono essere memorizzati in variabili o passati nella pipeline. È inoltre possibile aggiungere questi oggetti a una raccolta e quindi mostrare i risultati alla fine.

Le raccolte di oggetti funzionano bene con Export-CSV (e Import-CSV). Ogni riga del CSV è un oggetto, ogni colonna una proprietà.

I comandi di formattazione convertono oggetti in flussi di testo per la visualizzazione. Evita di usare i comandi Format- * fino al passaggio finale di qualsiasi elaborazione dati, per mantenere l'usabilità degli oggetti.

Leggi **Introduzione a PowerShell** online: <https://riptutorial.com/it/powershell/topic/822/introduzione-a-powershell>

Capitolo 2: alias

Osservazioni

Il sistema di denominazione PowerShell ha regole abbastanza rigide per la denominazione dei cmdlet (modello Verb-Noun; vedere [argomento non ancora creato] per ulteriori informazioni). Ma non è davvero conveniente scrivere `Get-ChildItems` ogni volta che si desidera elencare i file nella directory in modo interattivo.

Pertanto, Powershell abilita l'utilizzo di collegamenti - alias - anziché nomi di cmdlet.

Puoi scrivere `ls`, `dir` o `gci` invece di `Get-ChildItem` e ottenere lo stesso risultato. L'alias è equivalente al suo cmdlet.

Alcuni degli alias comuni sono:

| alias | cmdlet |
|-----------------|-------------------|
| %, per ciascuno | Per-EachObject |
| ?, dove | Where-Object |
| cat, gc, type | Get-Content |
| cd, chdir, sl | Set-Location |
| cls, chiaro | Clear-Host |
| cp, copy, cpi | Copy-Item |
| dir / ls / GCI | Get-ChildItem |
| eco, scrivi | Write-Output |
| fl | Format-List |
| ft | Format-Table |
| fw | Format-Wide |
| gc, pwd | Get-Location |
| gm | Get-Member |
| IEX | Invoke-Expression |
| ii | Invoke-Item |
| mv, sposta | Move-Item |

| alias | cmdlet |
|----------------------------------|-----------------|
| rm, rmdir, del, cancella, rd, ri | Rimuovi oggetto |
| dormire | Start-sonno |
| inizio, schifo | Start-Process |

Nella tabella sopra, puoi vedere come gli alias hanno abilitato la simulazione di comandi noti da altri ambienti (cmd, bash), quindi una maggiore rilevabilità.

Examples

Get-Alias

Per elencare tutti gli alias e le loro funzioni:

```
Get-Alias
```

Per ottenere tutti gli alias per un cmdlet specifico:

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

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

Per trovare gli alias facendo corrispondenze:

```
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 | | |

Set-Alias

Questo cmdlet consente di creare nuovi nomi alternativi per i cmdlet che escono

```
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 |

....

Tieni presente che qualsiasi alias creato verrà mantenuto solo nella sessione corrente. Quando avvii una nuova sessione devi creare nuovamente i tuoi alias. I profili PowerShell (vedi [argomento non ancora creato]) sono grandiosi per questi scopi.

Leggi alias online: <https://riptutorial.com/it/powershell/topic/5287/alias>

Capitolo 3: Analisi CSV

Examples

Utilizzo di base di Import-Csv

Dato il seguente file CSV

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

È possibile importare le righe CSV negli oggetti PowerShell utilizzando il comando `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
```

Importa da CSV e cast le proprietà per correggere il tipo

Per impostazione predefinita, `Import-Csv` importa tutti i valori come stringhe, quindi per ottenere gli oggetti `DateTime` e interi, è necessario eseguire il cast o analizzarli.

Usando l' `Foreach-Object` :

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

    #Output object
    $_
}
```

Utilizzando le proprietà calcolate:

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

Produzione:

```
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
```

Leggi Analisi CSV online: <https://riptutorial.com/it/powershell/topic/5691/analisi-csv>

Capitolo 4: Anonimizza IP (v4 e v6) nel file di testo con PowerShell

introduzione

Manipolazione di Regex per IPv4 e IPv6 e sostituzione tramite un falso indirizzo IP in un file di registro letto

Examples

Anonimizza l'indirizzo IP nel file di testo

```
# 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
}
```

Leggi Anonimizza IP (v4 e v6) nel file di testo con PowerShell online:

<https://riptutorial.com/it/powershell/topic/9171/anonimizza-ip--v4-e-v6--nel-file-di-testo-con-powershell>

Capitolo 5: Automazione dell'infrastruttura

introduzione

L'automazione dei servizi di gestione dell'infrastruttura riduce l'FTE e ottiene un ROI migliore utilizzando più strumenti, orchestrator, orchestration Engine, script e interfaccia utente semplice

Examples

Script semplice per il test di integrazione black-box delle applicazioni della console

Questo è un semplice esempio su come è possibile automatizzare i test per un'applicazione console che interagisce con input standard e output standard.

L'applicazione testata legge e sommare ogni nuova riga e fornirà il risultato dopo che è stata fornita una singola linea bianca. Lo script di power shell scrive "pass" quando l'output corrisponde.

```
$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()
}
```

Leggi Automazione dell'infrastruttura online:

<https://riptutorial.com/it/powershell/topic/10909/automazione-dell'infrastruttura>

Capitolo 6: Classi di PowerShell

introduzione

Una classe è un modello di codice di programma estendibile per la creazione di oggetti, che fornisce valori iniziali per stato (variabili membro) e implementazioni di comportamento (funzioni membro o metodi). Una classe è un modello per un oggetto. È usato come modello per definire la struttura degli oggetti. Un oggetto contiene dati a cui accediamo attraverso le proprietà e che possiamo lavorare sull'uso dei metodi. PowerShell 5.0 ha aggiunto la possibilità di creare classi personalizzate.

Examples

Metodi e proprietà

```
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!"
```

Elenco dei costruttori disponibili per una classe

5.0

In PowerShell 5.0+ è possibile elencare i costruttori disponibili chiamando il `new` metodo statico senza parentesi.

```
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)

```

Questa è la stessa tecnica che è possibile utilizzare per elencare le definizioni di sovraccarico per qualsiasi metodo

```

> 'abc'.CompareTo

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

```

Per le versioni precedenti è possibile creare la propria funzione per elencare i costruttori disponibili:

```

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

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

```

Uso:

```

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
```

Sovraccarico del costruttore

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

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

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

Ottieni tutti i membri di un'istanza

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

Ciò restituirà tutti i membri dell'istanza di tipo. Ecco una parte di un output di esempio per l'istanza 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()
...
```

Modello di classe 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
}
}

```

Ereditarietà dalla classe padre alla classe figlio

```

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()

```

COSÌ, **\$ Inherit.Message** ti darà il

"È sotto la classe genitore"

Leggi **Classi di PowerShell** online: <https://riptutorial.com/it/powershell/topic/1146/classi-di-powershell>

Capitolo 7: Codifica / decodifica URL

Osservazioni

L'espressione regolare utilizzata negli esempi di *URL di decodifica* è stata presa da [RFC 2396, Appendice B: analisi di un riferimento URI con un'espressione regolare](#) ; per i posteri, ecco una citazione:

La seguente riga è l'espressione regolare per suddividere un riferimento URI nei suoi componenti.

```
^((([^\s:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^\s#]*)?)?(#(.*))?)?  
12           3 4           5           6 7           8 9
```

I numeri nella seconda riga in alto servono solo per facilitare la leggibilità; indicano i punti di riferimento per ogni sottoespressione (cioè, ogni parentesi accoppiata). Ci riferiamo al valore abbinato per subexpression come \$. Ad esempio, facendo corrispondere l'espressione sopra a

```
http://www.ics.uci.edu/pub/ietf/uri/#Related
```

restituisce le seguenti corrispondenze di sottoespressione:

```
$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
```

Examples

Avvio rapido: codifica

```
$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
```

Nota: [ulteriori informazioni su HttpUtility](#) .

Avvio rapido: decodifica

Nota: questi esempi utilizzano le variabili create nella sezione *Avvio rapido: Codifica sopra*.

```
# 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
```

Nota: [ulteriori informazioni su HTTPUtility](#) .

Codifica stringa di query con `[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 '&'))
```

Con `[uri]::EscapeDataString()` , noterai che l'apostrofo (') non è stato codificato:

<https://example.vertigion.com/foos?foo2=complesso%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22e%20complesso%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=%20bar2&foo1=bar1>

Codifica stringa di query con `[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] $qqs_array = @()
foreach ($qqs in $qqs_data.GetEnumerator()) {
    $qqs_key = [System.Web.HttpUtility]::UrlEncode($qqs.Name)
    $qqs_value = [System.Web.HttpUtility]::UrlEncode($qqs.Value)
    $qqs_array.Add("{qqs_key}={qqs_value}") | Out-Null
}

$url = $url_format -f @([uri]::"UriScheme${scheme}", ($qqs_array -join '&'))
```

Con `[System.Web.HttpUtility]::UrlEncode()`, noterai che gli spazi diventano più segni (+) invece di %20:

<https://example.vertigion.com/foos?foo2=complesso%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22e%20complesso%3b%2f%3f%3a%40%26%3d%2b%24%2c+foo%27%22=%20bar2&foo1=bar1>

Decodifica URL con `[uri]::UnescapeDataString()`

Codificato con `[uri]::EscapeDataString()`

Innanzitutto, decodificare l'URL e la stringa di query codificati con `[uri]::EscapeDataString()` nell'esempio precedente:

<https://example.vertigion.com/foos?foo2=complesso%3B%2F%3F%3A%40%26%3D%2B%24%2C%20bar'%22e%20complesso%3B%2F%3F%3A%40%26%3D%2B%24%2C%20foo'%22=%20bar2&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=%20bar2&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"
}

```

Questo ti restituisce `[hashtable]$url_parts` ; che è uguale a (**Nota:** gli *spazi* nelle parti complesse sono *spazi*):

```

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

QueryStringParts                {foo2, complex;/?:@&=+$, foo'", foo1}

PS > $url_parts.QueryStringParts

Name                           Value
----                           -
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                            bar1

```

Codificato con `[System.Web.HttpUtility]::UrlEncode()`

Ora, decodificheremo l'URL e la stringa di query codificata con

`[System.Web.HttpUtility]::UrlEncode()` nell'esempio precedente:

<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"
}

```

Questo ti restituisce `[hashtable]$url_parts`, che equivale a (**Nota:** gli *spazi* nelle parti complesse sono *più segni* (+) nella prima parte e *spazi* nella seconda parte):

```

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

```

Decodifica l'URL con `[System.Web.HttpUtility]::UrlDecode()`

Codificato con `[uri]::EscapeDataString()`

Innanzitutto, decodificare l'URL e la stringa di query codificati con `[uri]::EscapeDataString()` nell'esempio precedente:

<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"
}

```

Questo ti restituisce `[hashtable]$url_parts` ; che è uguale a (**Nota:** gli *spazi* nelle parti complesse sono *spazi*):

```

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

QueryStringParts               {foo2, complex;/?:@&=+$, foo'", foo1}

PS > $url_parts.QueryStringParts

Name                           Value
----                           -
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                           bar1

```

Codificato con `[System.Web.HttpUtility]::UrlEncode()`

Ora, decodificheremo l'URL e la stringa di query codificata con `[System.Web.HttpUtility]::UrlEncode()` nell'esempio precedente:

[https://example.vertigion.com/foos ? foo2 = complesso% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + bar% 27% 22 e complesso% 3b% 2f% 3f% 3a% 40% 26% 3d% 2b% 24% 2c + foo% 27% 22 = bar2 & foo1 = bar1](https://example.vertigion.com/foos?foo2=complesso%3b%2f%3f%3a%40%26%3d%2b%24%2c+bar%27%22&complesso%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('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}

```

Questo ti restituisce `[hashtable]$url_parts` ; che è uguale a (**Nota:** gli *spazi* nelle parti complesse sono *spazi*):

```

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=b
QueryStringParts                {foo2, complex;/?:@&=+$, foo'", foo1}

PS > $url_parts.QueryStringParts

Name                           Value
----                           -
foo2                           complex;/?:@&=+$, bar'"
complex;/?:@&=+$, foo'"        bar2
foo1                            bar1

```

Leggi [Codifica / decodifica URL online: https://riptutorial.com/it/powershell/topic/7352/codifica---decodifica-url](https://riptutorial.com/it/powershell/topic/7352/codifica---decodifica-url)

Capitolo 8: Come scaricare l'ultimo artefatto da Artifactory usando lo script Powershell (v2.0 o inferiore)?

introduzione

Questa documentazione spiega e fornisce i passaggi per scaricare gli artefatti più recenti da un repository Artifactory di JFrog usando Powershell Script (v2.0 o inferiore).

Examples

Script di PowerShell per scaricare l'ultimo artifat

```
$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)
```

Leggi [Come scaricare l'ultimo artefatto da Artifactory usando lo script Powershell \(v2.0 o inferiore\)?](https://riptutorial.com/it/powershell/topic/8883/come-scaricare-l-ultimo-artefatto-da-artifactory-usando-lo-script-powershell--v2-0-o-inferiore--) online: <https://riptutorial.com/it/powershell/topic/8883/come-scaricare-l-ultimo-artefatto-da-artifactory-usando-lo-script-powershell--v2-0-o-inferiore-->

Capitolo 9: Comportamento di restituzione in PowerShell

introduzione

Può essere utilizzato per uscire dall'ambito corrente, che può essere una funzione, uno script o un blocco di script. In PowerShell, il risultato di ogni istruzione viene restituito come output, anche senza una parola chiave Return esplicita o per indicare che è stata raggiunta la fine dell'ambito.

Osservazioni

È possibile leggere ulteriori informazioni sulla semantica di ritorno nella pagina [about_Return](#) su TechNet o richiamando il `get-help return` da un prompt di PowerShell.

Domande & risposte notevoli con più esempi / spiegazioni:

- [Funzione restituisce il valore in PowerShell](#)
- [PowerShell: la funzione non ha un valore di ritorno appropriato](#)

[about_return](#) su MSDN lo spiega in modo succinto:

La parola chiave Return restituisce una funzione, uno script o un blocco di script. Può essere utilizzato per uscire da un ambito in un punto specifico, per restituire un valore o per indicare che è stata raggiunta la fine dell'ambito.

Gli utenti che hanno familiarità con linguaggi come C o C # potrebbero voler utilizzare la parola chiave Return per rendere esplicita la logica di lasciare uno scope.

In Windows PowerShell, i risultati di ogni istruzione vengono restituiti come output, anche senza un'istruzione che contiene la parola chiave Return. Lingue come C o C # restituiscono solo il valore o i valori specificati dalla parola chiave Return.

Examples

uscita anticipata

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

"Ciao" sarà posto in cantiere in uscita, "Mondo" non lo farà

Gotcha! Rientro in pipeline

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

I cmdlet della pipeline (ad es. `ForEach-Object`, `Where-Object`, ecc.) Operano sulle chiusure. Il ritorno qui passerà all'elemento successivo sulla pipeline, non all'elaborazione dell'uscita. È possibile utilizzare l' **interruzione** anziché il **ritorno** se si desidera uscire dall'elaborazione.

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

Gotcha! Ignorando l'uscita indesiderata

Ispirato da

- [PowerShell: la funzione non ha un valore di ritorno appropriato](#)

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

L' `Out-Null` è necessario perché il metodo `.NET ArrayList.Add` restituisce il numero di elementi nella raccolta dopo l'aggiunta. Se omissso, la pipeline avrebbe contenuto `1, 2, "a", "b"`

Esistono diversi modi per omettere l'output indesiderato:

```
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"  
}
```

Nota: per saperne di più sul perché preferire `> $null`, vedere [\[argomento non ancora creato\]](#).

Ritorna con un valore

(parafrasato da [about_return](#))

I seguenti metodi avranno gli stessi valori sulla pipeline

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

```

    $a = "Hello"
    $a
    return
}

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

```

Come lavorare con i ritorni delle funzioni

Una funzione restituisce tutto ciò che non è catturato da qualcos'altro.

Se usi la parola chiave **return**, ogni istruzione dopo la riga di ritorno non verrà eseguita!

Come questo:

```

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!"
}

```

Test-Function

Tornerà:

- Inizio
- La chiave di registro appena creata (questo perché ci sono alcune istruzioni che creano un output che potresti non aspettarti)
- Sì, ha funzionato!

Test-Function -ExceptionalReturn Restituirà:

- Inizio
- Dannazione, non ha funzionato!

Se lo fai in questo modo:

```

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
}
```

Test-Function

Tornerà:

- Sì, ha funzionato!

Test-Function -ExceptionalReturn Restituirà:

- Dannazione, non ha funzionato!

Con questo trucco puoi controllare l'output restituito anche se non sei sicuro di cosa sputerà ogni dichiarazione.

Funziona così

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

il `.` rende il seguente scriptblock incluso nel codice

il `{}` contrassegna il blocco di script

il `| Out-Null` pipe qualsiasi output inaspettato a Out-Null (quindi è andato!)

Poiché lo scriptblock è incluso, ottiene lo stesso scope del resto della funzione.

In questo modo è possibile accedere alle variabili create all'interno dello scriptblock.

Leggi Comportamento di restituzione in PowerShell online:

<https://riptutorial.com/it/powershell/topic/4781/comportamento-di-restituzione-in-powershell>

Capitolo 10: Comunicazione con API RESTful

introduzione

REST sta per Representational State Transfer (a volte si scrive "ReST"). Si basa su un protocollo di comunicazione memorizzabile da zero, client-server e in cache e viene utilizzato principalmente il protocollo HTTP. Viene principalmente utilizzato per creare servizi Web leggeri, manutenibili e scalabili. Un servizio basato su REST è chiamato servizio RESTful e le API che vengono utilizzate per questo sono API RESTful. In PowerShell, `Invoke-RestMethod` viene utilizzato per `Invoke-RestMethod`.

Examples

Usa i Webhook in arrivo su Slack.com

Definisci il tuo carico utile per inviare possibili dati più complessi

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

Utilizzare il cmdlet `ConvertTo-Json` e `Invoke-RestMethod` per eseguire la chiamata

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

Invia un messaggio a 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
```

Utilizzo di REST con oggetti PowerShell per ottenere e inserire singoli dati

OTTIENI i tuoi dati REST e archivia in un oggetto PowerShell:

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

Modifica i tuoi dati:

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

Metti i dati REST indietro

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

Utilizzare REST con oggetti PowerShell per OTTENERE e POSTARE molti articoli

OTTIENI i tuoi dati REST e archivia in un oggetto PowerShell:

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

Modifica molti articoli nei tuoi dati:

```
$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 tutti i dati REST indietro:

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

Utilizzare REST con PowerShell per eliminare elementi

Identifica l'elemento che deve essere cancellato ed eliminalo:

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

Leggi Comunicazione con API RESTful online:

<https://riptutorial.com/it/powershell/topic/3869/comunicazione-con-api-restful>

Capitolo 11: Comunicazione TCP con PowerShell

Examples

Ascoltatore TCP

```
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]
        }
    }
}
```

Inizia ad ascoltare con il seguente e acquisisci qualsiasi messaggio nella variabile `$msg` :

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

TCP Sender

```
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 wrtier
    $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()
}
}

```

Invia un messaggio con:

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

Nota : i messaggi TCP potrebbero essere bloccati dal firewall del software o da eventuali firewall esterni che stai cercando di attraversare. Assicurarsi che la porta TCP impostata nel comando precedente sia aperta e che sia stato impostato il listener sulla stessa porta.

Leggi Comunicazione TCP con PowerShell online:

<https://riptutorial.com/it/powershell/topic/5125/comunicazione-tcp-con-powershell>

Capitolo 12: Configurazione dello stato desiderata

Examples

Semplice esempio: abilitazione di WindowsFeature

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

Se si esegue questa configurazione in Powershell (EnableIISFeature), verrà generato un file localhost.mof. Questa è la configurazione "compilata" che puoi eseguire su una macchina.

Per testare la configurazione DSC sul tuo localhost, puoi semplicemente richiamare quanto segue:

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

Avvio di DSC (mof) sul computer remoto

Avviare un DSC su una macchina remota è quasi altrettanto semplice. Supponendo che tu abbia già impostato il servizio remoto di Powershell (o WSMAN abilitato).

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

Nb: Supponendo che tu abbia compilato una configurazione per il tuo nodo sulla tua macchina locale (e che il file myserver.somedomain.com.mof sia presente prima di avviare la configurazione)

Importazione di psd1 (file di dati) nella variabile locale

A volte può essere utile testare i file di dati di PowerShell e scorrere tra i nodi e i server.

Powershell 5 (WMF5) ha aggiunto questa piccola e semplice funzionalità per fare ciò chiamata Import-PowerShellDataFile.

Esempio:

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

Elenca le risorse DSC disponibili

Per elencare le risorse DSC disponibili sul nodo di creazione:

```
Get-DscResource
```

Questo elencherà tutte le risorse per tutti i moduli installati (che sono nel tuo PSModulePath) sul tuo nodo di creazione.

Per elencare tutte le risorse DSC disponibili che possono essere trovate nei sorgenti online (PSGallery ++) su WMF 5:

```
Find-DSCResource
```

Importazione di risorse da utilizzare in DSC

Prima di poter utilizzare una risorsa in una configurazione, è necessario importarla in modo esplicito. Basta averlo installato sul tuo computer, non ti permetterà di usare la risorsa implicitamente.

Importare una risorsa utilizzando `Import-DscResource`.

Esempio che mostra come importare la risorsa `PSDesiredStateConfiguration` e la risorsa `File`.

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

    Import-DscResource PSDesiredStateConfiguration

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

Nota : affinché le risorse DSC possano funzionare, è necessario che i moduli siano installati sui computer di destinazione quando si esegue la configurazione. Se non li hai installati, la configurazione fallirà.

Leggi [Configurazione dello stato desiderata online](https://riptutorial.com/it/powershell/topic/5662/configurazione-dello-stato-desiderata):

<https://riptutorial.com/it/powershell/topic/5662/configurazione-dello-stato-desiderata>

Capitolo 13: Convenzioni di denominazione

Examples

funzioni

```
Get-User ()
```

- Utilizzare lo schema *Verbale-Nome* durante la denominazione di una funzione.
- Il verbo implica un'azione come `Get` , `Set` , `New` , `Read` , `Write` e molti altri. Vedi i [verbi approvati](#)
- .
- Noun dovrebbe essere singolare anche se agisce su più elementi. `Get-User ()` può restituire uno o più utenti.
- Utilizzare il caso Pascal per entrambi i verbi e i nomi. Ad esempio `Get-UserLogin ()`

Leggi [Convenzioni di denominazione online](#):

<https://riptutorial.com/it/powershell/topic/9714/convenzioni-di-denominazione>

Capitolo 14: Creazione di risorse basate su classi DSC

introduzione

A partire da PowerShell versione 5.0, è possibile utilizzare le definizioni di classe PowerShell per creare risorse DSC (Desired State Configuration).

Per facilitare la creazione di risorse DSC, esiste un `[DscResource()]` applicato alla definizione di classe e una `[DscProperty()]` per designare le proprietà come configurabili dall'utente DSC Resource.

Osservazioni

Una risorsa DSC basata sulla classe deve:

- Essere decorato con l' `[DscResource()]`
- Definire un metodo `Test()` che restituisce `[bool]`
- Definire un metodo `Get()` che restituisce il proprio tipo di oggetto (ad esempio `[Ticket]`)
- Definire un metodo `Set()` che restituisce `[void]`
- Almeno una proprietà `Key` DSC

Dopo aver creato una risorsa DSC PowerShell basata su classi, deve essere "esportata" da un modulo, utilizzando un file manifest (.psd1) del modulo. All'interno del manifest del modulo, la chiave hash `DscResourcesToExport` viene utilizzata per dichiarare un array di risorse DSC (nomi di classi) da "esportare" dal modulo. Ciò consente ai consumatori del modulo DSC di "vedere" le risorse basate sulla classe all'interno del modulo.

Examples

Creare una classe di scheletro delle risorse DSC

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

Questo esempio dimostra come creare la sezione esterna di una classe PowerShell, che dichiara una risorsa DSC. Devi ancora compilare il contenuto della definizione della classe.

Scheletro di risorse DSC con proprietà chiave

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

```
[string] $TicketId
}
```

Una risorsa DSC deve dichiarare almeno una proprietà chiave. La proprietà chiave è ciò che identifica in modo univoco la risorsa da altre risorse. Ad esempio, supponiamo che tu stia creando una risorsa DSC che rappresenta un ticket in un sistema di ticketing. Ogni ticket sarebbe rappresentato in modo univoco con un ticket ID.

Ogni proprietà che verrà esposta *all'utente* della risorsa DSC deve essere decorata con l'`[DscProperty()]`. Questo attributo accetta un parametro `key`, per indicare che la proprietà è un attributo chiave per la risorsa DSC.

Risorsa DSC con proprietà obbligatorie

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

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

Quando crei una risorsa DSC, troverai spesso che non tutte le proprietà devono essere obbligatorie. Tuttavia, ci sono alcune proprietà di base che vorrete assicurare siano configurate dall'utente della risorsa DSC. Si utilizza il parametro `Mandatory` dell'attributo `[DscResource()]` per dichiarare una proprietà come richiesto dall'utente della risorsa DSC.

Nell'esempio sopra, abbiamo aggiunto una proprietà `Subject` a una risorsa `Ticket`, che rappresenta un ticket univoco in un sistema di ticketing, e lo abbiamo designato come proprietà `Mandatory`.

Risorsa DSC con metodi richiesti

```
[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
}
}
```

Questa è una risorsa DSC completa che dimostra tutti i requisiti fondamentali per creare una risorsa valida. Le implementazioni del metodo non sono complete, ma sono fornite con l'intenzione di mostrare la struttura di base.

Leggi Creazione di risorse basate su classi DSC online:

<https://riptutorial.com/it/powershell/topic/8733/creazione-di-risorse-basate-su-classi-dsc>

Capitolo 15: Esecuzione di eseguibili

Examples

Applicazioni console

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

Applicazioni 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)
```

Le applicazioni GUI vengono avviate in un processo diverso e restituiscono immediatamente il controllo all'host PowerShell. A volte è necessario che l'applicazione completi l'elaborazione prima dell'esecuzione della successiva istruzione PowerShell. Questo può essere ottenuto collegando l'output dell'applicazione a \$ null (3) o usando Start-Process con l'opzione -Wait (5).

Stream 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)
```

Stream 2 contiene oggetti System.Management.Automation.ErrorRecord. Si noti che alcune applicazioni come git.exe utilizzano il "flusso di errori" a scopo informativo, che non sono necessariamente errori. In questo caso è meglio guardare il codice di uscita per determinare se il flusso di errori debba essere interpretato come errore.

PowerShell comprende questi flussi: Output, Error, Warning, Verbose, Debug, Progress. Le applicazioni native generalmente usano solo questi flussi: Output, Error, Warning.

In PowerShell 5, tutti gli stream possono essere reindirizzati allo standard output / success stream (6).

Nelle precedenti versioni di PowerShell, solo gli stream specifici possono essere reindirizzati allo standard output / success stream (7). In questo esempio, il "flusso di errori" verrà reindirizzato al flusso di output.

Codici di uscita

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

Si tratta di variabili PowerShell integrate che forniscono informazioni aggiuntive sull'errore più recente. `$LastExitCode` è il codice di uscita finale dell'ultima applicazione nativa che è stata eseguita. `$?` e `$Error[0]` è l'ultimo record di errore generato da PowerShell.

Leggi **Esecuzione di eseguibili online**: <https://riptutorial.com/it/powershell/topic/7707/esecuzione-di-eseguibili>

Capitolo 16: Espressioni regolari

Sintassi

- 'text' -match 'RegexPattern'
- 'text' -replace 'RegexPattern', 'newvalue'
- [regex] :: Corrispondenza ("testo", "modello") # Corrispondenza singola
- [regex] :: Corrispondenze ("testo", "modello") # Corrispondenze multiple
- [Regex] :: Sostituire ("testo", "modello", "newValue")
- [regex] :: Replace ("text", "pattern", {param (\$ m)}) #MatchEvaluator
- [regex] :: Escape ("input") #Escape caratteri speciali

Examples

Partita singola

È possibile determinare rapidamente se un testo include uno schema specifico utilizzando Regex. Esistono diversi modi per lavorare con Regex in PowerShell.

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

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

Utilizzando l'operatore -Match

Per determinare se una stringa corrisponde a un modello usando l'operatore built-in `-matches`, utilizzare la sintassi `'input' -match 'pattern'`. Ciò restituirà `true` o `false` seconda del risultato della ricerca. Se c'era una corrispondenza è possibile visualizzare la partita e i gruppi (se definiti nel modello) accedendo alla variabile `$Matches`.

```
> $text -match $pattern
True

> $Matches

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

Puoi anche usare `-match` per filtrare attraverso una serie di stringhe e restituire solo le stringhe

contenenti una corrispondenza.

```
> $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

Utilizzando Select-String

PowerShell 2.0 ha introdotto un nuovo cmdlet per la ricerca nel testo mediante espressioni regolari. Esso restituisce un `MatchInfo` struttura per `textinput` che contiene una corrispondenza. Puoi accedere alle sue proprietà per trovare gruppi corrispondenti ecc.

```
> $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)}
```

Come `-match`, `Select-String` può essere utilizzato anche per filtrare attraverso una serie di stringhe convogliando una matrice su di essa. Crea un oggetto `MatchInfo` per stringa che include una corrispondenza.

```
> $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` può anche cercare usando un normale pattern di testo (nessuna regex) aggiungendo l' `-SimpleMatch`.

Utilizzo di `[Regex]::Match()`

È anche possibile utilizzare il metodo statico `Match()` disponibile nella classe `[Regex].NET`.

```
> [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)
```

Sostituire

Un compito comune per regex è sostituire il testo che corrisponde a un modello con un nuovo valore.

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

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

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

Utilizzando -Riposare l'operatore

L'operatore `-replace` in PowerShell può essere utilizzato per sostituire il testo che corrisponde a un modello con un nuovo valore utilizzando la sintassi `'input' -replace 'pattern', 'newvalue'`.

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

Utilizzo del metodo [Regex]: Sostituisci ()

La sostituzione delle corrispondenze può essere eseguita anche utilizzando il metodo `Replace()` nella classe `[Regex]` .NET.

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

Sostituisci il testo con il valore dinamico usando un valore MatchEvaluator

A volte è necessario sostituire un valore corrispondente a un modello con un nuovo valore basato su quella specifica corrispondenza, rendendo impossibile la previsione del nuovo valore. Per questi tipi di scenari, un `MatchEvaluator` può essere molto utile.

In PowerShell, `MatchEvaluator` è semplice come un blocco di script con un singolo parametro che contiene un oggetto `Match` per la corrispondenza corrente. L'output dell'azione sarà il nuovo valore per quella specifica partita. `MatchEvaluator` può essere utilizzato con il metodo statico

```
[Regex]::Replace() .
```

Esempio : sostituzione del testo dentro `()` con la sua lunghezza

```
#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
}
```

```
}
```

Produzione:

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

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

Esempio: crea un `sample` maiuscolo

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

$MatchEvaluator = {
    param($match)

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

Produzione:

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

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

Sfuggire personaggi speciali

Un modello regex utilizza molti caratteri speciali per descrivere un modello. Es., `.` significa "qualsiasi carattere", `+` è "uno o più" ecc.

Per utilizzare questi caratteri, come `.`, `+` ecc., in uno schema, è necessario sfuggire a loro per rimuovere il loro significato speciale. Questo viene fatto usando il carattere di escape che è un backslash `\` in espressioni regolari. Esempio: per cercare `+`, dovresti usare il pattern `\+`.

Può essere difficile ricordare tutti i caratteri speciali nella regex, quindi per sfuggire a tutti i caratteri speciali di una stringa che si desidera cercare, è possibile utilizzare il metodo

```
[Regex]::Escape("input") .
```

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

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

Più partite

Esistono diversi modi per trovare tutte le corrispondenze per un modello in un testo.

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

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

Utilizzando Select-String

È possibile trovare tutte le corrispondenze (corrispondenza globale) aggiungendo l' `-AllMatches` a `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)
```

Utilizzo di [Regex] :: Matches ()

Il metodo `Matches()` nella classe `.NET [regex]` può anche essere utilizzato per eseguire una ricerca globale per più corrispondenze.

```
> [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)
```

Leggi Espressioni regolari online: <https://riptutorial.com/it/powershell/topic/6674/espressioni-regolari>

Capitolo 17: Flussi di lavoro di PowerShell

introduzione

PowerShell Workflow è una funzionalità introdotta a partire da PowerShell versione 3.0. Le definizioni del flusso di lavoro sono molto simili alle definizioni delle funzioni di PowerShell, tuttavia vengono eseguite all'interno dell'ambiente Windows Workflow Foundation, anziché direttamente nel motore PowerShell.

Diverse caratteristiche uniche "out of box" sono incluse nel motore del flusso di lavoro, in particolare la persistenza del lavoro.

Osservazioni

La funzionalità del flusso di lavoro PowerShell è supportata esclusivamente sulla piattaforma Microsoft Windows, in PowerShell Desktop Edition. PowerShell Core Edition, che è supportato su Linux, Mac e Windows, non supporta la funzione Flusso di lavoro di PowerShell.

Quando si crea un flusso di lavoro di PowerShell, tenere presente che i flussi di lavoro chiamano le attività, non i cmdlet. È ancora possibile chiamare i cmdlet da un flusso di lavoro di PowerShell, ma il `InlineScript Workflow` avvolgerà implicitamente la chiamata del cmdlet in un'attività `InlineScript`. Puoi anche includere esplicitamente il codice all'interno dell'attività `InlineScript`, che esegue il codice PowerShell; per impostazione predefinita, l'attività `InlineScript` viene eseguita in un processo separato e restituisce il risultato al Flusso di lavoro chiamante.

Examples

Esempio di flusso di lavoro semplice

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

Questo è un esempio di base di una definizione del flusso di lavoro di PowerShell.

Flusso di lavoro con parametri di input

Proprio come le funzioni di PowerShell, i flussi di lavoro possono accettare parametri di input. I parametri di input possono essere associati a un tipo di dati specifico, come una stringa, un numero intero, ecc. Utilizzare la parola chiave `param` standard per definire un blocco di parametri di input, direttamente dopo la dichiarazione del flusso di lavoro.

```
workflow DoSomeWork {
    param (
        [string[]] $ComputerName
    )
}
```

```
)  
Get-Process -ComputerName $ComputerName  
}  
  
DoSomeWork -ComputerName server01, server02, server03
```

Esegui il flusso di lavoro come lavoro in background

I flussi di lavoro di PowerShell sono intrinsecamente equipaggiati con la possibilità di essere eseguiti come lavoro in background. Per chiamare un flusso di lavoro come lavoro in background di PowerShell, utilizzare il parametro `-AsJob` durante il `-AsJob` del flusso di lavoro.

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

Aggiungi un blocco parallelo a un flusso di lavoro

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

Una delle caratteristiche uniche di PowerShell Workflow è la capacità di definire un blocco di attività come parallelo. Per utilizzare questa funzione, utilizzare la parola chiave `parallel` all'interno del flusso di lavoro.

Chiamare le attività del flusso di lavoro in parallelo può aiutare a migliorare le prestazioni del tuo flusso di lavoro.

Leggi **Flussi di lavoro di PowerShell online**: <https://riptutorial.com/it/powershell/topic/8745/flussi-di-lavoro-di-powershell>

Capitolo 18: Funzioni di PowerShell

introduzione

Una funzione è fondamentalmente un blocco di codice con nome. Quando si chiama il nome della funzione, viene eseguito il blocco di script all'interno di quella funzione. È un elenco di istruzioni di PowerShell con un nome che si assegna. Quando si esegue una funzione, si digita il nome della funzione. È un metodo per risparmiare tempo quando si affrontano attività ripetitive. Formati di PowerShell in tre parti: la parola chiave 'Funzione', seguita da un Nome, infine, il payload contenente il blocco di script, che è racchiuso tra parentesi graffe / parentesi.

Examples

Funzione semplice senza parametri

Questo è un esempio di una funzione che restituisce una stringa. Nell'esempio, la funzione viene chiamata in un'istruzione che assegna un valore a una variabile. Il valore in questo caso è il valore di ritorno della funzione.

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

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` dichiara che il seguente codice è una funzione.

`Get-Greeting` è il nome della funzione. Ogni volta che la funzione deve essere utilizzata nello script, la funzione può essere richiamata invocandola per nome.

`{ ... }` è il blocco di script che viene eseguito dalla funzione.

Se il codice sopra è eseguito nell'ISE, i risultati sarebbero qualcosa di simile:

```
Hello World
Hello World
```

Parametri di base

Una funzione può essere definita con parametri usando il blocco `param`:

```
function Write-Greeting {
```

```

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

```

O usando la sintassi della funzione semplice:

```

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

```

Nota: i parametri di trasmissione non sono richiesti in entrambi i tipi di definizione dei parametri.

La sintassi della funzione semplice (SFS) ha funzionalità molto limitate rispetto al blocco param. Sebbene sia possibile definire i parametri da esporre all'interno della funzione, non è possibile specificare [Attributi parametro](#), utilizzare [Parameter Validation](#), includere `[CmdletBinding()]`, con SFS (e questo è un elenco non esaustivo).

Le funzioni possono essere richiamate con parametri ordinati o con nome.

L'ordine dei parametri sull'invocazione corrisponde all'ordine della dichiarazione nell'intestazione della funzione (per impostazione predefinita) oppure può essere specificato utilizzando l'Attributo parametro `Position` (come mostrato nell'esempio di funzione avanzato, sopra).

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

In alternativa, questa funzione può essere invocata con parametri denominati

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

Parametri obbligatori

I parametri di una funzione possono essere contrassegnati come obbligatori

```

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

```

Se la funzione viene invocata senza un valore, la riga di comando chiederà il valore:

```

$greeting = Get-Greeting

cmdlet Get-Greeting at command pipeline position 1

```

Supply values for the following parameters:
name:

Funzione avanzata

Questa è una copia dello snippet di funzione avanzato del Powershell ISE. Fondamentalmente questo è un modello per molte delle cose che puoi usare con le funzioni avanzate in PowerShell.

Punti chiave da notare:

- integrazione get-help - l'inizio della funzione contiene un blocco di commenti che viene impostato per essere letto dal cmdlet get-help. Il blocco funzione può essere posizionato alla fine, se lo si desidera.
- cmdletbinding - la funzione si comporterà come un cmdlet
- parametri
- set di parametri

```
<#  
.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')]  
  
    [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
{
}
}

```

Validazione dei parametri

Esistono diversi modi per convalidare l'immissione dei parametri, in PowerShell.

Invece di scrivere codice all'interno di funzioni o script per convalidare i valori dei parametri, questi `ParameterAttributes` generano se vengono passati valori non validi.

ValidateSet

A volte è necessario limitare i possibili valori che un parametro può accettare. Diciamo che vogliamo consentire solo il rosso, il verde e il blu per il parametro `$Color` in uno script o una funzione.

Possiamo usare l'attributo del parametro `ValidateSet` per limitare questo. Ha l'ulteriore vantaggio di consentire il completamento della tabulazione quando si imposta questo argomento (in alcuni ambienti).

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

È inoltre possibile specificare `IgnoreCase` per disabilitare la distinzione tra maiuscole e minuscole.

ValidateRange

Questo metodo di validazione dei parametri richiede un valore `Int32` minimo e massimo e richiede che il parametro sia compreso in tale intervallo.

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

ValidatePattern

Questo metodo di validazione dei parametri accetta parametri che corrispondono al modello regex specificato.

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

ValidateLength

Questo metodo di validazione dei parametri verifica la lunghezza della stringa passata.

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

ValidateCount

Questo metodo di validazione dei parametri verifica la quantità di argomenti passati, ad esempio, una serie di stringhe.

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

ValidateScript

Infine, il metodo `ValidateScript` è straordinariamente flessibile, prende un blocco di script e lo valuta usando `$ _` per rappresentare l'argomento passato. Passa quindi l'argomento se il risultato è `$ true` (incluso qualsiasi output valido).

Questo può essere usato per verificare che esista un file:

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

Per verificare che un utente esista in AD:

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

E praticamente qualsiasi altra cosa tu possa scrivere (dato che non si limita ai oneliner):

```
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
)
```

Leggi Funzioni di PowerShell online: <https://riptutorial.com/it/powershell/topic/1673/funzioni-di-powershell>

Capitolo 19: Gestione degli errori

introduzione

Questo argomento illustra i tipi di errore e la gestione degli errori in PowerShell.

Examples

Tipi di errore

Un errore è un errore, ci si potrebbe chiedere come potrebbero esserci tipi in esso. Bene, con powershell l'errore rientra ampiamente in due criteri,

- Errore di terminazione
- Errore non terminante

Come dice il nome, gli errori di terminazione terminano l'esecuzione e un errore non terminante consente di continuare con l'istruzione successiva.

Questo è vero supponendo che il valore **\$ ErrorActionPreference** sia predefinito (Continua). **\$ ErrorActionPreference** è una [variabile di prefrence](#) che dice a PowerShell cosa fare in caso di errore "non terminante".

Errore di terminazione

Un errore di terminazione può essere gestito con un tipico tentativo di cattura, come di seguito

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

Il frammento di cui sopra verrà eseguito e l'errore verrà catturato attraverso il blocco catch.

Errore non terminante

Per impostazione predefinita, un errore senza terminazione nell'altra mano non verrà catturato nel blocco catch. Il motivo dietro questo è un errore non terminante non è considerato un errore critico.

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

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

Se si esegue la riga sopra riportata, non si otterrà l'output dal blocco catch poiché l'errore non è considerato critico e l'esecuzione continuerà semplicemente dal comando successivo. Tuttavia, l'errore verrà visualizzato nella console. Per gestire un errore senza interruzione, è necessario modificare la preferenza dell'errore.

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

Ora, con la preferenza di errore aggiornata, questo errore verrà considerato un errore di chiusura e verrà catturato nel blocco catch.

Richiamo degli errori di terminazione e non di terminazione:

Il cmdlet **Write-Error** scrive semplicemente l'errore nel programma host invocante. Non si ferma l'esecuzione. Dove il **throw** ti darà un errore di chiusura e interromperà l'esecuzione

```
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"
```

Leggi Gestione degli errori online: <https://riptutorial.com/it/powershell/topic/8075/gestione-degli-errori>

Capitolo 20: Gestione dei pacchetti

introduzione

La gestione dei pacchetti di PowerShell consente di trovare, installare, aggiornare e disinstallare i moduli PowerShell e altri pacchetti.

[PowerShellGallery.com](https://powershellgallery.com) è la fonte predefinita per i moduli PowerShell. È anche possibile sfogliare il sito per i pacchetti disponibili, comandare e visualizzare in anteprima il codice.

Examples

Trova un modulo PowerShell usando un modello

Per trovare un modulo che termini con `DSC`

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

Creare la Reposibilità del modulo PowerShell di default

Se per qualche motivo, il repository di moduli PowerShell predefinito viene rimosso `PSGallery`. Dovrai crearlo. Questo è il comando.

```
Register-PSRepository -Default
```

Trova un modulo per nome

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

Installa un modulo per nome

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

Disinstallare un modulo il mio nome e versione

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

Aggiorna un modulo per nome

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

Leggi [Gestione dei pacchetti online](https://riptutorial.com/it/powershell/topic/8698/gestione-dei-pacchetti): <https://riptutorial.com/it/powershell/topic/8698/gestione-dei-pacchetti>

pacchetti

Capitolo 21: Gestione di segreti e credenziali

introduzione

In Powershell, per evitare di memorizzare la password in *chiaro*, utilizziamo diversi metodi di crittografia e la memorizziamo come stringa sicura. Quando non si specifica una chiave o una chiave di sicurezza, ciò funzionerà solo per lo stesso utente sullo stesso computer sarà in grado di decrittografare la stringa crittografata se non si sta utilizzando Keys / SecureKeys. Qualsiasi processo eseguito sotto lo stesso account utente sarà in grado di decrittografare quella stringa crittografata sullo stesso computer.

Examples

Richiesta di credenziali

Per richiedere le credenziali, si dovrebbe quasi sempre usare il cmdlet `Get-Credential`:

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

Nome utente pre-compilato:

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

Aggiungi un messaggio di richiesta personalizzato:

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

Accesso alla password in chiaro

La password in un oggetto credenziale è una `[SecureString]` crittografata. Il modo più semplice è ottenere un `[NetworkCredential]` che non memorizzi la password crittografata:

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

Il metodo helper (`.GetNetworkCredential()`) esiste solo su oggetti `[PSCredential]`.

Per trattare direttamente con `[SecureString]`, utilizzare i metodi .NET:

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

Lavorare con le credenziali memorizzate

Per archiviare e recuperare facilmente le credenziali crittografate, utilizzare la serializzazione XML

incorporata di PowerShell (Clixml):

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

Per reimportare:

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

La cosa importante da ricordare è che per impostazione predefinita viene utilizzata l'API di protezione dati di Windows e la chiave utilizzata per crittografare la password è specifica sia per l'utente che per la macchina su cui è in esecuzione il codice.

Di conseguenza, le credenziali crittografate non possono essere importate da un utente diverso né lo stesso utente su un altro computer.

Con la crittografia di diverse versioni della stessa credenziale con diversi utenti in esecuzione e su computer diversi, è possibile avere lo stesso segreto disponibile per più utenti.

Inserendo il nome utente e il nome del computer nel nome del file, è possibile memorizzare tutti i segreti crittografati in un modo che consente allo stesso codice di utilizzarli senza alcun codice fisso:

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"
```

Il codice che utilizza le credenziali archiviate:

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

La versione corretta del file per l'utente in esecuzione verrà caricata automaticamente (o fallirà perché il file non esiste).

Memorizzazione delle credenziali in forma crittografata e passaggio come parametro quando richiesto

```
$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.
```

Leggi Gestione di segreti e credenziali online:

<https://riptutorial.com/it/powershell/topic/2917/gestione-di-segreti-e-credenziali>

Capitolo 22: GUI in Powershell

Examples

GUI WPF per il cmdlet 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
```

Ciò crea una finestra di dialogo che consente all'utente di selezionare un nome di computer, quindi visualizzerà una tabella di servizi e i relativi stati su quel computer. Questo esempio utilizza WPF anziché Windows Form.

Leggi GUI in Powershell online: <https://riptutorial.com/it/powershell/topic/7141/gui-in-powershell>

Capitolo 23: Guida basata sui commenti

introduzione

PowerShell presenta un meccanismo di documentazione chiamato guida basata su commenti. Permette di documentare script e funzioni con commenti al codice. La guida basata su commenti è la maggior parte delle volte scritta in blocchi di commenti contenenti più parole chiave di aiuto. Aiuta le parole chiave a iniziare con i punti e identifica le sezioni di aiuto che verranno visualizzate eseguendo il cmdlet `Get-Help`.

Examples

Guida basata sui commenti

```
<#  
  
.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 documentazione della funzione sopra può essere visualizzata eseguendo `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>
```

Si noti che le parole chiave basate su commenti iniziano con a . abbinare le sezioni dei risultati Get-Help .

Guida agli script basata su commenti

```
<#
.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 documentazione di script sopra può essere visualizzata eseguendo `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>

```

Leggi Guida basata sui commenti online: <https://riptutorial.com/it/powershell/topic/9530/guida-basata-sui-commenti>

Capitolo 24: hashtables

introduzione

Una tabella hash è una struttura che mappa le chiavi ai valori. Vedi [Tabella hash](#) per i dettagli.

Osservazioni

Un concetto importante che si basa su Hash Tables è [Splatting](#) . È molto utile per fare un gran numero di chiamate con parametri ripetitivi.

Examples

Creazione di una tabella hash

Esempio di creazione di una HashTable vuota:

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

Esempio di creazione di una HashTable con dati:

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

Accedere a un valore di tabella hash per chiave.

Un esempio di definizione di una tabella hash e accesso a un valore tramite la chiave

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

Un esempio di accesso a una chiave con caratteri non validi per un nome di proprietà:

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

```
Value3
```

Fare il ciclo su un tavolo hash

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

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

#Output
Key1 : Value1
Key2 : Value2
```

Aggiungi una coppia di valori chiave a una tabella hash esistente

Un esempio, per aggiungere una chiave "Chiave2" con un valore di "Valore2" alla tabella hash, utilizzando l'operatore addizione:

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

#Output

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

Un esempio, per aggiungere una chiave "Key2" con un valore di "Value2" alla tabella hash usando il metodo Add:

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

#Output

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

Enumerazione tramite chiavi e coppie valore-chiave

Enumerazione tramite chiavi

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

Enumerazione tramite coppie di valori-chiave

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

Rimuovere una coppia di valori chiave da una tabella hash esistente

Un esempio, per rimuovere una chiave "Key2" con un valore di "Value2" dalla tabella hash, utilizzando l'operatore remove:

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

Leggi hashtables online: <https://riptutorial.com/it/powershell/topic/8083/hashtables>

Capitolo 25: Imporre i prerequisiti di script

Sintassi

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

Osservazioni

#requires istruzione #requires può essere posizionata su qualsiasi riga dello script (non deve essere la prima riga) ma deve essere la prima istruzione su quella riga.

#requires possibile utilizzare più istruzioni #requires in uno script.

Per ulteriori riferimenti, fare riferimento alla documentazione ufficiale su Technet - [about_about_Requires](#) .

Examples

Applicare la versione minima dell'host PowerShell

```
#requires -version 4
```

Dopo aver provato a eseguire questo script nella versione precedente, verrà visualizzato questo messaggio di errore

```
. \script.ps1: non è possibile eseguire lo script 'script.ps1' perché conteneva un'istruzione "#requires" sulla riga 1 per Windows PowerShell versione 5.0. La versione richiesta dallo script non corrisponde alla versione corrente di Windows PowerShell versione 2.0.
```

Imponi eseguire lo script come amministratore

4.0

```
#requires -RunAsAdministrator
```

Dopo aver provato a eseguire questo script senza privilegi di amministratore, vedrai questo messaggio di errore

```
. \script.ps1: non è possibile eseguire lo script 'script.ps1' perché contiene un'istruzione "#requires" per l'esecuzione come amministratore. La sessione corrente
```

di Windows PowerShell non è in esecuzione come amministratore. Avviare Windows PowerShell utilizzando l'opzione Esegui come amministratore, quindi provare di nuovo a eseguire lo script.

Leggi Imporre i prerequisiti di script online: <https://riptutorial.com/it/powershell/topic/5637/imporre-i-prerequisiti-di-script>

Capitolo 26: Incorporare codice gestito (C # | VB)

introduzione

Questo argomento descrive brevemente come il codice gestito C # o VB .NET può essere copiato e utilizzato all'interno di uno script PowerShell. Questo argomento non sta esplorando tutti gli aspetti del cmdlet Add-Type.

Per ulteriori informazioni sul cmdlet Add-Type, consultare la documentazione MSDN (per 5.1) qui: <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add- genere>

Parametri

| Parametro | Dettagli |
|------------------------------|--|
| -TypeDefinition <String_> | Accetta il codice come una stringa |
| -Language <String_> | Specifica il linguaggio del codice gestito. Valori accettati: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript |

Osservazioni

Rimozione dei tipi aggiunti

Nelle versioni successive di PowerShell, Remove-TypeData è stato aggiunto alle librerie dei cmdlet di PowerShell che possono consentire la rimozione di un tipo all'interno di una sessione. Per maggiori dettagli su questo cmdlet, vai qui: <https://msdn.microsoft.com/en-us/powershell/reference/4.0/microsoft.powershell.utility/remove-typedata>

Sintassi CSharp e .NET

Per quelle esperienze con .NET è ovvio che le diverse versioni di C # possono essere abbastanza radicalmente differenti nel loro livello di supporto per certe sintassi.

Se si utilizza Powershell 1.0 e / o -Language CSharp, il codice gestito utilizzerà .NET 2.0, che è carente in un certo numero di funzionalità che gli sviluppatori C # utilizzano normalmente senza pensarci in questi giorni, come Generics, Linq e Lambda. A ciò si aggiunge il polimorfismo formale, che viene gestito con i parametri predefiniti nelle versioni successive di C # /. NET.

Examples

C # Esempio

Questo esempio mostra come incorporare un C # di base in uno script PowerShell, aggiungerlo allo spazio di esecuzione / sessione e utilizzare il codice all'interno della sintassi di 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();
```

Esempio VB.NET

Questo esempio mostra come incorporare un C # di base in uno script PowerShell, aggiungerlo allo spazio di esecuzione / sessione e utilizzare il codice all'interno della sintassi di 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();
```

Leggi Incorporare codice gestito (C # | VB) online:

<https://riptutorial.com/it/powershell/topic/9823/incorporare-codice-gestito--c-sharp---vb->

Capitolo 27: Introduzione a Pester

Osservazioni

Pester è un framework di test per PowerShell che consente di eseguire test case per il proprio codice PowerShell. Può essere usato per correre ex. unit test per aiutarti a verificare che i tuoi moduli, script, ecc. funzionino come previsto.

[Cos'è Pester e perché dovrei preoccuparmi?](#)

Examples

Iniziare con Pester

Per iniziare con il codice di PowerShell per il collaudo dell'unità utilizzando il modulo Pester, è necessario avere familiarità con tre parole chiave / comandi:

- **Descrivi** : Definisce un gruppo di test. Tutti i file di test di Pester richiedono almeno un blocco Describe.
- **Esso** : definisce un test individuale. Puoi avere più blocchi It all'interno di un blocco Describe.
- **Dovrebbe** : il comando verifica / prova. È usato per definire il risultato che dovrebbe essere considerato un test riuscito.

Campione:

```
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
    }
}
```

Produzione:

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

Leggi **Introduzione a Pester** online: <https://riptutorial.com/it/powershell/topic/5753/introduzione-a-pester>

Capitolo 28: Introduzione a Psake

Sintassi

- Attività - funzione principale per eseguire un passo del tuo script di compilazione
- Depends: proprietà che specificano da cosa dipende il passaggio corrente
- impostazione predefinita: deve essere sempre presente un'attività predefinita che verrà eseguita se non viene specificata alcuna attività iniziale
- FormatTaskName: specifica il modo in cui ciascun passaggio viene visualizzato nella finestra dei risultati.

Osservazioni

[psake](#) è uno strumento di automazione build scritto in PowerShell ed è ispirato a Rake (Ruby make) e Bake (Boo make). È usato per creare build usando il pattern di dipendenza.

Documentazione disponibile [qui](#)

Examples

Schema di base

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

Task Build {
    "Build"
}

Task Clean {
    "Clean"
}

Task default -Depends Build
```

FormatTaskName esempio

```
# 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

```

Esegui l'attività in modo condizionale

```

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

```

Leggi Introduzione a Psake online: <https://riptutorial.com/it/powershell/topic/5019/introduzione-a-psake>

Capitolo 29: Invio di email

introduzione

Una tecnica utile per gli amministratori di Exchange Server è la possibilità di inviare messaggi e-mail tramite SMTP da PowerShell. A seconda della versione di PowerShell installata sul computer o sul server, esistono diversi modi per inviare e-mail tramite PowerShell. Esiste un'opzione del cmdlet nativo che è semplice e facile da usare. Utilizza il cmdlet **Send-MailMessage**.

Parametri

| Parametro | Dettagli |
|----------------------------|---|
| Allegati <string []> | Nome percorso e file dei file da allegare al messaggio. Percorsi e nomi di file possono essere inoltrati a Send-MailMessage. |
| Ccn <string []> | Indirizzi email che ricevono una copia di un messaggio e-mail ma che non appaiono come destinatari nel messaggio. Inserisci i nomi (facoltativo) e l'indirizzo email (obbligatorio), ad esempio Nome utente@esempio.com o utente@esempio.com. |
| Corpo <String_> | Contenuto del messaggio di posta elettronica. |
| BodyAsHtml | Indica che il contenuto è in formato HTML. |
| Cc <string []> | Indirizzi email che ricevono una copia di un messaggio di posta elettronica. Inserisci i nomi (facoltativo) e l'indirizzo email (obbligatorio), ad esempio Nome utente@esempio.com o utente@esempio.com. |
| Credenziali | Specifica un account utente che dispone dell'autorizzazione per inviare messaggi dall'indirizzo email specificato. L'impostazione predefinita è l'utente corrente. Inserisci il nome come Utente o Dominio \ Utente o inserisci un oggetto PSCredential. |
| DeliveryNotificationOption | Specifica le opzioni di notifica di consegna per il messaggio di posta elettronica. È possibile specificare più valori. Le notifiche di consegna vengono inviate nel messaggio all'indirizzo specificato in Per parametro. Valori accettabili: Nessuno, OnSuccess, OnFailure, Delay, Never. |
| Codifica | Codifica per il corpo e il soggetto. Valori accettabili: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Default, OEM. |
| A partire dal | Indirizzi email da cui viene inviata la posta. Inserisci i nomi (facoltativo) e l'indirizzo email (obbligatorio), ad esempio Nome |

| Parametro | Dettagli |
|-----------|--|
| | utente@esempio.com o utente@esempio.com. |
| Porta | Porta alternativa sul server SMTP. Il valore predefinito è 25. Disponibile da Windows PowerShell 3.0. |
| Priorità | Priorità del messaggio di posta elettronica. Valori accettabili: normale, alta, bassa. |
| SmtServer | Nome del server SMTP che invia il messaggio di posta elettronica. Il valore predefinito è il valore della variabile \$PSEmailServer. |
| Soggetto | Oggetto del messaggio di posta elettronica. |
| A | Indirizzi email a cui viene inviata la posta. Inserisci i nomi (facoltativo) e l'indirizzo email (obbligatorio), ad esempio Nome utente@esempio.com o utente@esempio.com |
| UseSSL | Utilizza il protocollo Secure Sockets Layer (SSL) per stabilire una connessione al computer remoto per inviare posta |

Examples

Send-MailMessage semplice

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

Send-MailMessage con parametri predefiniti

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

# Notice: Splatting requires @ instead of $ in front of variable name
```

SMTPClient - Posta con file .txt nel messaggio del corpo

```
# 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"
    $SMTPClient = New-Object Net.Mail.SmtpClient($SmtpServer, 25)
    $SMTPClient.EnableSsl = $false
    $SMTPClient.Send($EmailFrom, $EmailTo, $Subject, $Body)
}

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

Send_mail
```

Leggi Invio di email online: <https://riptutorial.com/it/powershell/topic/2040/invio-di-email>

Capitolo 30: Lavorare con gli oggetti

Examples

Aggiornamento degli oggetti

Aggiungere proprietà

Se si desidera aggiungere proprietà a un oggetto esistente, è possibile utilizzare il cmdlet `Add-Member`. Con `PSObjects`, i valori vengono mantenuti in un tipo di "Proprietà nota"

```
$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
```

È anche possibile aggiungere proprietà con Cmdlet `Select-Object` (le cosiddette proprietà calcolate):

```
$newObject = $object | Select-Object *, @{{label='SomeOtherProp'; expression={'Another value'}}}

# Returns
PS> $newObject
Name ID Address SomeNewProp SomeOtherProp
---- -- -
nem 12          A value      Another value
```

Il comando sopra può essere abbreviato a questo:

```
$newObject = $object | Select *,@{l='SomeOtherProp';e={'Another value'}}
```

Rimozione delle proprietà

È possibile utilizzare il cmdlet `Select-Object` per rimuovere le proprietà da un oggetto:

```
$object = $newObject | Select-Object * -ExcludeProperty ID, Address

# Returns
```

```
PS> $Object
Name SomeNewProp SomeOtherProp
---- -
nem  A value      Another value
```

Creare un nuovo oggetto

PowerShell, a differenza di altri linguaggi di scripting, invia oggetti attraverso la pipeline. Ciò significa che quando si inviano dati da un comando a un altro, è essenziale essere in grado di creare, modificare e raccogliere oggetti.

Creare un oggetto è semplice. La maggior parte degli oggetti che crei sono oggetti personalizzati in PowerShell e il tipo da utilizzare è `PSObject`. PowerShell ti permetterà anche di creare qualsiasi oggetto che potresti creare in .NET.

Ecco un esempio di creazione di nuovi oggetti con alcune proprietà:

Opzione 1: nuovo oggetto

```
$newObject = New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -
nem  12
```

È possibile memorizzare l'oggetto in una variabile precedendo il comando con `$newObject =`

Potrebbe anche essere necessario memorizzare raccolte di oggetti. Questo può essere fatto creando una variabile di raccolta vuota e aggiungendo oggetti alla raccolta, in questo modo:

```
$newCollection = @()
$newCollection += New-Object -TypeName PSObject -Property @{
    Name = $env:username
    ID = 12
    Address = $null
}
```

Si può quindi desiderare di ripetere questo oggetto di raccolta per oggetto. Per fare ciò, individuare la sezione `Loop` nella documentazione.

Opzione 2: Select-Object

Un modo meno comune di creare oggetti che troverai ancora su Internet è il seguente:

```

$newObject = 'unuseddummy' | Select-Object -Property Name, ID, Address
$newObject.Name = $env:username
$newObject.ID = 12

# Returns
PS> $newObject
Name ID Address
---- -- -
nem 12

```

Opzione 3: acceleratore di tipo pscustomobject (richiesto PSV3 +)

L'acceleratore di tipo ordinato forza PowerShell a mantenere le nostre proprietà nell'ordine in cui le abbiamo definite. Non è necessario l'acceleratore di tipo ordinato per utilizzare `[PSCustomObject]` :

```

$newObject = [PSCustomObject][Ordered]@{
    Name = $env:Username
    ID = 12
    Address = $null
}

# Returns
PS> $newObject
Name ID Address
---- -- -
nem 12

```

Esaminando un oggetto

Ora che hai un oggetto, potrebbe essere bello capire di cosa si tratta. È possibile utilizzare il cmdlet `Get-Member` per vedere cosa è un oggetto e cosa contiene:

```
Get-Item c:\windows | Get-Member
```

Questo produce:

```
TypeName: System.IO.DirectoryInfo
```

Seguito da un elenco di proprietà e metodi che l'oggetto ha.

Un altro modo per ottenere il tipo di un oggetto è utilizzare il metodo `GetType`, in questo modo:

```

C:\> $Object = Get-Item C:\Windows
C:\> $Object.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DirectoryInfo                                     System.IO.FileSystemInfo

```

Per visualizzare un elenco di proprietà che l'oggetto ha, insieme ai relativi valori, è possibile utilizzare il cmdlet `Format-List` con il parametro `Property` impostato su: `*` (che significa tutto).

Ecco un esempio, con l'output risultante:

```
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 |

Creazione di istanze di classi generiche

Nota: esempi scritti per PowerShell 5.1 È possibile creare istanze di Classi generiche

```
#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
```

Fornisce l'output:

```
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
```

Sono anche possibili raccolte generiche

```
[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
```

Fornisce l'output:

```
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
```

Leggi Lavorare con gli oggetti online: <https://riptutorial.com/it/powershell/topic/1328/lavorare-con-gli-oggetti>

Capitolo 31: Lavorare con i file XML

Examples

Accesso a un file 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>
```

Caricamento di un file XML

Per caricare un file XML, puoi utilizzare uno di questi:

```
# 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")
```

Accesso a XML come oggetti

```
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

```

Accesso a XML con 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

```

Accesso a XML contenente spazi dei nomi con 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 |

Creare un documento XML usando 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()
```

File XML di output

```
<?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>
```

Dati di esempio

Documento XML

Per prima cosa, definiamo un documento XML di esempio denominato " **books.xml** " nella nostra directory corrente:

```
<?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>

```

Nuovi dati

Quello che vogliamo fare è aggiungere alcuni nuovi libri a questo documento, diciamo *Patriot Games* di Tom Clancy (sì, sono un fan delle opere di Clancy ^ __ ^) e un favorito della Fantascienza: *la Guida galattica per gli autostoppisti* di Douglas Adams principalmente perché Zaphod Beeblebrox è semplicemente divertente da leggere.

In qualche modo abbiamo acquisito i dati per i nuovi libri e li abbiamo salvati come elenco di oggetti personalizzati PSC:

```

$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;
}
);

```

Modelli

Ora dobbiamo definire alcune strutture XML scheletro per i nostri nuovi dati. Fondamentalmente, vuoi creare uno scheletro / modello per ogni lista di dati. Nel nostro esempio, ciò significa che abbiamo bisogno di un modello per il libro, i personaggi e gli editori. Possiamo anche usarlo per definire alcuni valori predefiniti, come il valore per il 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="" />
"@;

```

Abbiamo finito con il set-up.

Aggiungere i nuovi dati

Ora che siamo tutti configurati con i nostri dati di esempio, aggiungiamo gli oggetti personalizzati all'oggetto documento 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

```

Ora possiamo scrivere il nostro XML su disco, schermo, web o ovunque!

Profitto

Anche se questa potrebbe non essere la procedura per chiunque, l'ho trovata utile per evitare un sacco di

```
[void]$xml.SelectSingleNode("/complicated/xpath/goes[here]").AppendChild($xml.CreateElement("newElementName", $textValue));
```

seguito da `$xml.SelectSingleNode("/complicated/xpath/goes/here/newElementName") = $textValue`

Penso che il metodo descritto nell'esempio sia più pulito e più facile da analizzare per gli umani normali.

miglioramenti

Potrebbe essere possibile cambiare il modello per includere elementi con i bambini invece di scomporre ogni sezione come modello separato. Devi solo fare attenzione a clonare l'elemento precedente quando passi in rassegna l'elenco.

Leggi Lavorare con i file XML online: <https://riptutorial.com/it/powershell/topic/4882/lavorare-con-i-file-xml>

Capitolo 32: Lavorare con la pipeline di PowerShell

introduzione

PowerShell introduce un modello di pipeline degli oggetti, che consente di inviare interi oggetti attraverso la pipeline al consumo di commandlet o (almeno) all'output. A differenza del pipelining basato su stringhe classiche, non è necessario che le informazioni negli oggetti collegati siano su posizioni specifiche. I Commandlet possono dichiarare di interagire con Oggetti dalla pipeline come input, mentre i valori di ritorno vengono inviati automaticamente alla pipeline.

Sintassi

- **INIZIA** Il primo blocco. Eseguito una volta all'inizio. L'input della pipeline qui è \$ null, poiché non è stato impostato.
- **PROCESSO** Il secondo blocco. Eseguito per ciascun elemento della pipeline. Il parametro pipeline è uguale all'elemento attualmente elaborato.
- **FINE** Ultimo blocco. Eseguito una volta alla fine. Il parametro pipeline è uguale all'ultimo elemento dell'input, poiché non è stato modificato da quando è stato impostato.

Osservazioni

Nella maggior parte dei casi, l'input della pipeline sarà una serie di oggetti. Sebbene il comportamento del blocco `PROCESS{}` possa sembrare simile al blocco `foreach{}`, saltare un elemento nell'array richiede un processo diverso.

Se, come in `foreach{}`, hai usato `continue` nel blocco `PROCESS{}`, interromperà la pipeline, saltando tutte le seguenti istruzioni incluso il blocco `END{}`. Invece, usa `return` - termina solo il blocco `PROCESS{}` per l'elemento corrente e passa al successivo.

In alcuni casi, è necessario produrre il risultato di funzioni con codifica diversa. La codifica dell'output di CmdLets è controllata dalla variabile `$OutputEncoding`. Quando l'output è destinato a essere inserito in una pipeline per le applicazioni native, potrebbe essere una buona idea correggere la codifica in modo che corrisponda alla destinazione `$OutputEncoding = [Console]::OutputEncoding`

Ulteriori riferimenti:

Articolo del blog con più informazioni su `$OutputEncoding`

<https://blogs.msdn.microsoft.com/powershell/2006/12/11/outputencoding-to-the-rescue/>

Examples

Funzioni di scrittura con ciclo di vita avanzato

Questo esempio mostra come una funzione può accettare input pipeline e iterare in modo efficiente.

Si noti che le `begin` e `end` strutture della funzione sono facoltativi quando `pipeline`, ma che `process` è necessaria quando si utilizza `ValueFromPipeline` o `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
```

Produzione:

```
VERBOSE: Beginning Write-FromPipeline
hello
world
1
2
3
VERBOSE: Ending Write-FromPipeline
```

Supporto per pipeline di base in funzioni

Questo è un esempio di una funzione con il supporto più semplice possibile per il `pipeline`. Qualsiasi funzione con supporto `pipeline` deve avere almeno un parametro con `ParameterAttribute ValueFromPipeline` o `ValueFromPipelineByPropertyName` impostato, come illustrato di seguito.

```
function Write-FromPipeline {
    param(
        [Parameter(ValueFromPipeline)] # This sets the ParameterAttribute
        [String]$Input
    )
    Write-Host $Input
}

$foo = 'Hello World!'
```

```
$foo | Write-FromPipeline
```

Produzione:

```
Hello World!
```

Nota: in PowerShell 3.0 e versioni successive, è supportato Valori predefiniti per `ParameterAttributes`. Nelle versioni precedenti, è necessario specificare `ValueFromPipeline=$true`.

Concetto di lavoro della pipeline

In una serie di condotte ciascuna funzione viene eseguita parallelamente alle altre, come i thread paralleli. Il primo oggetto elaborato viene trasmesso alla pipeline successiva e l'elaborazione successiva viene immediatamente eseguita in un altro thread. Questo spiega il guadagno ad alta velocità rispetto allo standard `ForEach`

```
@( bigFile_1, bigFile_2, ..., bigFile_n ) | Copy-File | Encrypt-File | Get-Md5
```

1. passo - copia il primo file (in Thread `Copy-file`)
2. passo - copia il secondo file (in Thread `Copy-file`) e contemporaneamente Cripta il primo (in `Encrypt-File`)
3. step - copia il terzo file (in Thread del `Copy-file`) e simultaneamente cripta il secondo file (in `Encrypt-File`) e contemporaneamente `get-Md5` del primo (in `Get-Md5`)

Leggi Lavorare con la pipeline di PowerShell online:

<https://riptutorial.com/it/powershell/topic/3937/lavorare-con-la-pipeline-di-powershell>

Capitolo 33: Lavori in background di PowerShell

introduzione

I lavori sono stati introdotti in PowerShell 2.0 e hanno contribuito a risolvere un problema inerente agli strumenti da riga di comando. In poche parole, se si avvia un'attività a lunga esecuzione, il prompt non è disponibile fino al termine dell'attività. Come esempio di un'attività a lunga esecuzione, pensa a questo semplice comando di PowerShell:

```
Get-ChildItem -Path c: \ -Recurse
```

Ci vorrà un po' di tempo per recuperare l'elenco completo delle directory della tua unità C :. Se lo esegui come lavoro, la console riprenderà il controllo e in seguito potrai acquisire il risultato.

Osservazioni

I lavori di PowerShell vengono eseguiti in un nuovo processo. Questo ha pro e contro che sono correlati.

Professionisti:

1. Il lavoro viene eseguito in un processo pulito, incluso l'ambiente.
2. Il lavoro può essere eseguito in modo asincrono per il processo PowerShell principale

Contro:

1. Le modifiche dell'ambiente di processo non saranno presenti nel lavoro.
2. I parametri passano e i risultati restituiti sono serializzati.
 - Ciò significa che se si modifica un oggetto parametro mentre il lavoro è in esecuzione, ciò non si rifletterà nel lavoro.
 - Ciò significa anche che se un oggetto non può essere serializzato non è possibile passarlo o restituirlo (sebbene PowerShell possa copiare qualsiasi parametro e passare / restituire un oggetto PSObject).

Examples

Creazione di lavoro di base

Avviare un blocco di script come processo in background:

```
$job = Start-Job -ScriptBlock {Get-Process}
```

Avvia uno script come processo in background:

```
$job = Start-Job -FilePath "C:\YourFolder\Script.ps1"
```

Avvia un lavoro utilizzando `Invoke-Command` su una macchina remota:

```
$job = Invoke-Command -ComputerName "ComputerName" -ScriptBlock {Get-Service winrm} -JobName "WinRM" -ThrottleLimit 16 -AsJob
```

Avvia lavoro come utente diverso (richiede password):

```
Start-Job -ScriptBlock {Get-Process} -Credential "Domain\Username"
```

O

```
Start-Job -ScriptBlock {Get-Process} -Credential (Get-Credential)
```

Avvia lavoro come utente diverso (Nessun prompt):

```
$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
```

Gestione del lavoro di base

Ottieni un elenco di tutti i lavori nella sessione corrente:

```
Get-Job
```

In attesa di un lavoro da terminare prima di ottenere il risultato:

```
$job | Wait-job | Receive-Job
```

Timeout di un lavoro se viene eseguito troppo a lungo (10 secondi in questo esempio)

```
$job | Wait-job -Timeout 10
```

Arresto di un lavoro (completa tutte le attività che sono in sospeso in quella coda di lavoro prima di terminare):

```
$job | Stop-Job
```

Rimuovi il lavoro dall'elenco dei lavori in background della sessione corrente:

```
$job | Remove-Job
```

Nota : quanto segue funzionerà solo sui lavori del `Workflow` lavoro.

Sospendi un processo di `Workflow` lavoro (Pausa):

```
$job | Suspend-Job
```

Riprendi un processo di `Workflow` lavoro:

```
$job | Resume-Job
```

Leggi Lavori in background di PowerShell online:

<https://riptutorial.com/it/powershell/topic/3970/lavori-in-background-di-powershell>

Capitolo 34: Logica condizionale

Sintassi

- if (espressione) {}
- if (espressione) {} else {}
- if (espressione) {} elseif (espressione) {}
- if (espressione) {} elseif (espressione) {} else {}

Osservazioni

Vedi anche [Operatori di confronto](#) , che possono essere usati in espressioni condizionali.

Examples

se, altro e altro se

Powershell supporta gli operatori di logica condizionale standard, proprio come molti linguaggi di programmazione. Ciò consente a determinate funzioni o comandi di essere eseguiti in circostanze particolari.

Con un `if` i comandi all'interno delle parentesi (`{ }`) vengono eseguiti solo se sono soddisfatte le condizioni all'interno di `if (())`

```
$test = "test"
if ($test -eq "test"){
    Write-Host "if condition met"
}
```

Puoi anche fare un `else` . Qui i comandi `else` vengono eseguiti se le condizioni `if` **non** sono soddisfatte:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
else{
    Write-Host "if condition not met"
}
```

o un `elseif` . Un altro `se` esegue i comandi se le condizioni `if` non sono soddisfatte e se le condizioni `elseif` sono soddisfatte:

```
$test = "test"
if ($test -eq "test2"){
    Write-Host "if condition met"
}
```

```
elseif ($test -eq "test"){
    Write-Host "ifelse condition met"
}
```

Nota l'uso sopra `-eq` (uguaglianza) CmdLet e non `=` o `==` come fanno molti altri linguaggi per l'equità.

Negazione

Potresti voler annullare un valore booleano, cioè inserire un'istruzione `if` se una condizione è falsa invece che vera. Questo può essere fatto utilizzando la `-Not` cmdlet

```
$test = "test"
if (-Not $test -eq "test2"){
    Write-Host "if condition not met"
}
```

Puoi anche usare `!` :

```
$test = "test"
if (!$test -eq "test2"){
    Write-Host "if condition not met"
}
```

c'è anche l'operatore `-ne` (non uguale):

```
$test = "test"
if ($test -ne "test2"){
    Write-Host "variable test is not equal to 'test2'"
}
```

Se stenografia condizionale

Se si desidera utilizzare la scorciatoia, è possibile utilizzare la logica condizionale con la seguente stenografia. Solo la stringa 'false' valuterà su 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."
}
```

Leggi Logica condizionale online: <https://riptutorial.com/it/powershell/topic/7208/logica-condizionale>

Capitolo 35: Loops

introduzione

Un ciclo è una sequenza di istruzioni che viene ripetuta continuamente finché non viene raggiunta una determinata condizione. Essere in grado di far eseguire ripetutamente il proprio codice a un blocco di codice è uno dei compiti più semplici ma utili nella programmazione. Un ciclo consente di scrivere un'istruzione molto semplice per produrre un risultato significativamente maggiore semplicemente mediante ripetizione. Se la condizione è stata raggiunta, l'istruzione successiva "passa attraverso" alla successiva istruzione sequenziale o rami all'esterno del ciclo.

Sintassi

- `for (<Initialization>; <Condition>; <Repetition>) {<Script_Block>}`
- `<Collezione> | Foreach-Object {<Script_Block_with _ $ __ as_current_item>}`
- `foreach (<Item> in <Collection>) {<Script_Block>}`
- `while (<Condition>) {<Script_Block>}`
- `do {<Script_Block>} while (<Condition>)`
- `fare {<Script_Block>} fino a (<Condizione>)`
- `<Collection> .foreach ({<Script_Block_with _ $ __ as_current_item>})`

Osservazioni

Per ciascuno

Esistono diversi modi per eseguire un ciclo foreach in PowerShell e presentano tutti i loro vantaggi e svantaggi:

| Soluzione | vantaggi | svantaggi |
|--------------------------|---|---|
| Dichiarazione Foreach | Più veloce. Funziona al meglio con le raccolte statiche (memorizzate in una variabile). | Nessun input o output della pipeline |
| Metodo ForEach () | La stessa sintassi di scriptblock di <code>Foreach-Object</code> , ma più veloce. Funziona al meglio con le raccolte statiche (memorizzate in una variabile). Supporta l'output della pipeline. | Nessun supporto per l'input della pipeline. Richiede PowerShell 4.0 o versioni successive |
| Foreach-Object | Supporta input e output della pipeline. | più lento |

| Soluzione | vantaggi | svantaggi |
|-----------|---|-----------|
| (cmdlet) | Supporta i blocchi di script di inizio e fine per l'inizializzazione e la chiusura delle connessioni ecc. Soluzione più flessibile. | |

Prestazione

```
$foreach = Measure-Command { foreach ($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
```

Mentre `ForEach-Object` è il più lento, il supporto della pipeline potrebbe essere utile in quanto consente di elaborare gli elementi non appena arrivano (durante la lettura di un file, la ricezione di dati, ecc.). Questo può essere molto utile quando si lavora con big data e poca memoria dato che non è necessario caricare tutti i dati nella memoria prima dell'elaborazione.

Examples

Per

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

Un tipico utilizzo del ciclo `for` è quello di operare su un sottoinsieme dei valori di un array. Nella maggior parte dei casi, se si desidera ripetere tutti i valori in una matrice, considerare l'uso di un'istruzione `foreach`.

Per ciascuno

`ForEach` ha due diversi significati in PowerShell. Uno è una [parola chiave](#) e l'altro è un alias per il cmdlet [ForEach-Object](#). Il primo è descritto qui.

Questo esempio dimostra come stampare tutti gli elementi in una matrice sull'host della console:

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

ForEach ($Name in $Names)
{
```

```
Write-Host "Hi, my name is $Name!"  
}
```

Questo esempio dimostra come catturare l'output di un ciclo ForEach:

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

Come l'ultimo esempio, questo esempio, invece, dimostra la creazione di un array prima di memorizzare il ciclo:

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

Mentre

Un ciclo while valuterà una condizione e se true eseguirà un'azione. Finché la condizione diventa vera, l'azione continuerà ad essere eseguita.

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

Nell'esempio seguente viene creato un ciclo che eseguirà il conto alla rovescia da 10 a 0

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

A differenza del ciclo `do -While`, la condizione viene valutata prima della prima esecuzione dell'azione. L'azione non verrà eseguita se la condizione iniziale è falsa.

Nota: durante la valutazione della condizione, PowerShell considererà true l'esistenza di un oggetto restituito. Questo può essere utilizzato in diversi modi, ma di seguito è riportato un esempio per monitorare un processo. Questo esempio genererà un processo di blocco note e quindi interromperà la shell corrente finché il processo è in esecuzione. Quando si chiude manualmente l'istanza del blocco note, la condizione while non riuscirà e il ciclo si interromperà.

```
Start-Process notepad.exe  
while(Get-Process notepad -ErrorAction SilentlyContinue){  
    Start-Sleep -Milliseconds 500  
}
```

ForEach-Object

Il cmdlet `ForEach-Object` funziona in modo simile all'istruzione `foreach` , ma prende il suo input dalla pipeline.

Utilizzo di base

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

Esempio:

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

`ForEach-Object` ha due alias predefiniti, `foreach` e `%` (sintassi abbreviata). Il più comune è `%` perché `foreach` può essere confuso con l' [istruzione foreach](#) . Esempi:

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

Utilizzo avanzato

`ForEach-Object` si distingue dalle soluzioni `foreach` alternative perché è un cmdlet che significa che è progettato per utilizzare la pipeline. Per questo motivo, supporta tre scriptblock proprio come un cmdlet o una funzione avanzata:

- **Inizia** : eseguito una volta prima di scorrere gli elementi che arrivano dalla pipeline. Solitamente utilizzato per creare funzioni da utilizzare nel loop, creando variabili, aprendo connessioni (database, web +) ecc.
- **Processo** : eseguito una volta per elemento arrivato dalla pipeline. "Normal" `foreach` codeblock. Questo è il default usato negli esempi sopra quando il parametro non è specificato.
- **Fine** : eseguito una volta dopo l'elaborazione di tutti gli elementi. Solitamente utilizzato per chiudere le connessioni, generare un report, ecc.

Esempio:

```
"Any", "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
}
```

Fare

I loop di do sono utili quando si desidera eseguire sempre un codice di blocco almeno una volta. Un Do-loop valuterà la condizione dopo aver eseguito il codice, diversamente da un ciclo while che lo esegue prima di eseguire il codice.

Puoi usare loop di do in due modi:

- Loop *mentre* la condizione è vera:

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

- Loop *fino a quando* la condizione è vera, in altre parole, loop mentre la condizione è falsa:

```
Do {
    code_block
} until (condition)
```

Esempi reali:

```
$i = 0

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

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

Do-While e Do-Until sono loop antonymous. Se il codice è all'interno dello stesso, la condizione verrà invertita. L'esempio sopra illustra questo comportamento.

Metodo ForEach ()

4.0

Invece del cmdlet `ForEach-Object`, qui è anche la possibilità di utilizzare un metodo `ForEach` direttamente su array di oggetti come questo

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

oppure - se lo si desidera - le parentesi intorno al blocco di script possono essere omesse

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

Entrambe genereranno l'output di seguito

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

Continua

L'operatore `Continue` lavora nei cicli `For` , `ForEach` , `While` e `Do` Salta l'iterazione corrente del loop, saltando in cima al ciclo più interno.

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

Quanto sopra uscirà da 1 a 20 alla console ma mancherà il numero 7.

Nota : quando si utilizza un loop di pipeline, è necessario utilizzare `return` anziché `Continue` .

Rompere

L'operatore di `break` uscirà immediatamente da un loop di programma. Può essere utilizzato nei cicli `For` , `ForEach` , `While` e `Do` o in un'istruzione `Switch` .

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

Quanto sopra conterà fino a 15 ma si fermerà non appena 7 sarà raggiunto.

Nota : quando si utilizza un loop di pipeline, `break` si comporterà come `continue` . Per simulare l' `break` del ciclo della pipeline, è necessario incorporare logica aggiuntiva, `cmdlet`, ecc. È più semplice attaccare con loop non di pipeline se è necessario utilizzare l' `break` .

Rompere etichette

Break può anche chiamare un'etichetta posizionata davanti all'istanza di un ciclo:

```
$i = 0
:mainLoop While ($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++
}
```

Nota: questo codice incrementerà i a 8 e j a 13 che causerà k uguale a 104 . Poiché k supera 100 , il codice uscirà da entrambi i cicli.

Leggi Loops online: <https://riptutorial.com/it/powershell/topic/1067/loops>

Capitolo 36: Moduli PowerShell

introduzione

A partire da PowerShell versione 2.0, gli sviluppatori possono creare moduli PowerShell. I moduli PowerShell racchiudono un insieme di funzionalità comuni. Ad esempio, esistono moduli PowerShell specifici del fornitore che gestiscono vari servizi cloud. Esistono anche moduli PowerShell generici che interagiscono con i servizi di social media ed eseguono attività di programmazione comuni, come la codifica Base64, lavorando con Named Pipes e altro.

I moduli possono esporre alias di comandi, funzioni, variabili, classi e altro.

Examples

Crea un modulo manifest

```
@{
  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 = @()
}
```

Ogni buon modulo PowerShell ha un manifest di modulo. Il manifest del modulo contiene semplicemente metadati relativi a un modulo PowerShell e non definisce i contenuti effettivi del modulo.

Il file manifest è un file di script PowerShell, con estensione `.psd1`, che contiene un HashTable. L'hashTable nel manifest deve contenere chiavi specifiche, in modo che PowerShell possa interpretarlo correttamente come un file del modulo PowerShell.

L'esempio precedente fornisce un elenco delle chiavi HashTable principali che compongono un manifest di modulo, ma ce ne sono molte altre. Il comando `New-ModuleManifest` ti aiuta a creare un nuovo scheletro manifest del modulo.

Esempio di modulo semplice

```
function Add {
  [CmdletBinding()]
```

```
param (
    [int] $x
    , [int] $y
)

return $x + $y
}

Export-ModuleMember -Function Add
```

Questo è un semplice esempio di come potrebbe apparire un file del modulo di script PowerShell. Questo file si chiamerebbe `MyCoolModule.psm1` e viene fatto riferimento dal file manifest (`.psd1`) del modulo. Noterai che il comando `Export-ModuleMember` ci consente di specificare quali funzioni nel modulo vogliamo "esportare" o esporre all'utente del modulo. Alcune funzioni saranno solo interne e non dovrebbero essere esposte, quindi queste saranno omesse dalla chiamata a `Export-ModuleMember`.

Esportare una variabile da un modulo

```
$FirstName = 'Bob'
Export-ModuleMember -Variable FirstName
```

Per esportare una variabile da un modulo, si utilizza il comando `Export-ModuleMember`, con il parametro `-Variable`. Ricorda, tuttavia, che se la variabile non viene esportata esplicitamente nel file manifest del modulo (`.psd1`), la variabile non sarà visibile al consumatore del modulo. Pensa al modulo manifest come un "gatekeeper". Se una funzione o una variabile non è consentita nel manifest del modulo, non sarà visibile al consumatore del modulo.

Nota: ! l'esportazione di una variabile è simile alla creazione di un campo in una classe pubblica. Non è consigliabile. Sarebbe meglio esporre una funzione per ottenere il campo e una funzione per impostare il campo.

Strutturazione dei moduli PowerShell

Piuttosto che definire tutte le tue funzioni in un singolo file di modulo di script PowerShell `.psm1`, potresti voler dividere la tua funzione in singoli file. È quindi possibile dot-source questi file dal file del modulo di script, che in sostanza li tratta come se fossero parte del file `.psm1` stesso.

Considera questa struttura di directory del modulo:

```
\MyCoolModule
  \Functions
    Function1.ps1
    Function2.ps1
    Function3.ps1
MyCoolModule.psd1
MyCoolModule.psm1
```

All'interno del tuo file `MyCoolModule.psm1` puoi inserire il seguente codice:

```
Get-ChildItem -Path $PSScriptRoot\Functions |  
ForEach-Object -Process { . $PSItem.FullName }
```

Ciò farebbe dot-source i singoli file di funzione nel file del modulo `.psm1`.

Posizione dei moduli

PowerShell cerca i moduli nelle directory elencate in `$ Env: PSModulepath`.

Un modulo chiamato `foo`, in una cartella chiamata `foo`, verrà trovato con `Import-Module foo`

In tale cartella, PowerShell cercherà un manifest del modulo (`foo.psd1`), un file del modulo (`foo.psm1`), una DLL (`foo.dll`).

Visibilità membro del modulo

Per impostazione predefinita, solo le funzioni definite in un modulo sono visibili al di fuori del modulo. In altre parole, se si definiscono variabili e alias in un modulo, non saranno disponibili se non nel codice del modulo.

Per sovrascrivere questo comportamento, è possibile utilizzare il cmdlet `Export-ModuleMember`. Ha parametri chiamati `-Function`, `-Variable`, e `-Alias` che consentono di specificare esattamente quali membri vengono esportati.

È importante notare che se si utilizza `Export-ModuleMember`, saranno visibili **solo** gli elementi specificati.

Leggi Moduli PowerShell online: <https://riptutorial.com/it/powershell/topic/8734/moduli-powershell>

Capitolo 37: Moduli, script e funzioni

introduzione

I moduli PowerShell portano l'estendibilità all'amministratore di sistemi, DBA e sviluppatore. Che si tratti semplicemente di un metodo per condividere funzioni e script.

Le funzioni di Powershell servono per evitare codici ripetitivi. Fare riferimento a [Funzioni PS] [1] [1]: [Funzioni PowerShell](#)

Gli script di PowerShell vengono utilizzati per automatizzare le attività di amministrazione che comprendono shell della riga di comando e cmdlet associati creati su .NET Framework.

Examples

Funzione

Una funzione è un blocco di codice denominato che viene utilizzato per definire il codice riutilizzabile che dovrebbe essere facile da usare. Di solito è incluso in uno script per aiutare a riutilizzare il codice (per evitare il codice duplicato) o distribuito come parte di un modulo per renderlo utile per gli altri in più script.

Scenari in cui una funzione potrebbe essere utile:

- Calcola la media di un gruppo di numeri
- Genera un report per i processi in esecuzione
- Scrivi una funzione che verifica che un computer sia "sano" eseguendo il ping del computer e accedendo a `c$ -share`

Le funzioni vengono create utilizzando la parola chiave `function`, seguita da un nome di una sola parola e da un blocco di script contenente il codice da eseguire quando viene chiamato il nome della funzione.

```
function NameOfFunction {  
    Your code  
}
```

dimostrazione

```
function HelloWorld {  
    Write-Host "Greetings from PowerShell!"  
}
```

Uso:

```
> HelloWorld
Greetings from PowerShell!
```

copione

Uno script è un file di testo con l'estensione `.ps1` che contiene i comandi di PowerShell che verranno eseguiti quando viene chiamato lo script. Poiché gli script sono file salvati, sono facili da trasferire tra computer.

Gli script sono spesso scritti per risolvere un problema specifico, es .:

- Esegui un'attività di manutenzione settimanale
- Per installare e configurare una soluzione / applicazione su un computer

dimostrazione

MyFirstScript.ps1:

```
Write-Host "Hello World!"
2+2
```

Puoi eseguire uno script inserendo il percorso del file usando un:

- Percorso assoluto, es. `c:\MyFirstScript.ps1`
- Percorso relativo, ex `.\MyFirstScript.ps1` se la directory corrente della console PowerShell era `C:\`

Uso:

```
> .\MyFirstScript.ps1
Hello World!
4
```

Uno script può anche importare moduli, definirne le proprie funzioni, ecc.

MySecondScript.ps1:

```
function HelloWorld {
    Write-Host "Greetings from PowerShell!"
}

HelloWorld
Write-Host "Let's get started!"
2+2
HelloWorld
```

Uso:

```
> .\MySecondScript.ps1
Greetings from PowerShell!
```

```
Let's get started!
4
Greetings from PowerShell!
```

Modulo

Un modulo è una raccolta di funzioni riutilizzabili correlate (o cmdlet) che possono essere facilmente distribuite ad altri utenti PowerShell e utilizzate in più script o direttamente nella console. Un modulo viene solitamente salvato nella propria directory ed è composto da:

- Uno o più file di codice con estensione `.psm1` contenente funzioni o assembly binari (`.dll`) contenenti cmdlet
- Un modulo manifest `.psd1` descrive il nome, la versione, l'autore, la descrizione dei moduli, quali funzioni / cmdlet fornisce ecc.
- Altri requisiti per farlo funzionare incl. dipendenze, script ecc.

Esempi di moduli:

- Un modulo contenente funzioni / cmdlet che eseguono statistiche su un set di dati
- Un modulo per interrogare e configurare database

Per semplificare il reperimento e l'importazione di un modulo da parte di PowerShell, questo viene spesso inserito in uno dei percorsi del modulo PowerShell noti, definiti in `$env:PSModulePath`.

dimostrazione

Elenca i moduli che vengono installati in una delle posizioni dei moduli conosciute:

```
Get-Module -ListAvailable
```

Importa un modulo, es. Modulo `Hyper-V` :

```
Import-Module Hyper-V
```

Elenca i comandi disponibili in un modulo, es. il modulo `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 |

Funzioni avanzate

Le funzioni avanzate si comportano allo stesso modo dei cmdlet. PowerShell ISE include due scheletri di funzioni avanzate. Accedi a questi tramite il menu, modifica, frammenti di codice o Ctrl

+ J. (A partire da PS 3.0, le versioni successive potrebbero differire)

Le cose chiave che includono le funzioni avanzate sono

- Guida integrata e personalizzata per la funzione, accessibile tramite `Get-Help`
- può usare `[CmdletBinding()]` che fa sì che la funzione si comporti come un cmdlet
- ampie opzioni di parametri

Versione semplice:

```
<#
.Synopsis
    Short description
.DESCRPTION
    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
    {
    }
}
}
```

Versione completa:

```
<#
.Synopsis
    Short description
.DESCRPTION
    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
{
}
}
```

Leggi Moduli, script e funzioni online: <https://riptutorial.com/it/powershell/topic/5755/moduli--script-e-funzioni>

Capitolo 38: Modulo ActiveDirectory

introduzione

Questo argomento introdurrà alcuni dei cmdlet di base utilizzati all'interno del Modulo Active Directory per PowerShell, per manipolare Utenti, Gruppi, Computer e Oggetti.

Osservazioni

Si ricorda che il sistema di guida di PowerShell è una delle migliori risorse che è possibile utilizzare.

```
Get-Help Get-ADUser -Full
Get-Help Get-ADGroup -Full
Get-Help Get-ADComputer -Full
Get-Help Get-ADObject -Full
```

Tutta la documentazione della guida fornirà esempi, sintassi e guida ai parametri.

Examples

Modulo

```
#Add the ActiveDirectory Module to current PowerShell Session
Import-Module ActiveDirectory
```

utenti

Recupera utente di Active Directory

```
Get-ADUser -Identity JohnSmith
```

Recupera tutte le proprietà associate all'utente

```
Get-ADUser -Identity JohnSmith -Properties *
```

Recupera proprietà selezionate per l'utente

```
Get-ADUser -Identity JohnSmith -Properties * | Select-Object -Property sAMAccountName, Name, Mail
```

Nuovo utente AD

```
New-ADUser -Name "MarySmith" -GivenName "Mary" -Surname "Smith" -DisplayName "MarySmith" -Path "CN=Users,DC=Domain,DC=Local"
```

gruppi

Recupera gruppo di Active Directory

```
Get-ADGroup -Identity "My-First-Group" #Ensure if group name has space quotes are used
```

Recupera tutte le proprietà associate al gruppo

```
Get-ADGroup -Identity "My-First-Group" -Properties *
```

Recupera tutti i membri di un gruppo

```
Get-ADGroupMember -Identity "My-First-Group" | Select-Object -Property sAMAccountName  
Get-ADgroup "MY-First-Group" -Properties Members | Select -ExpandProperty Members
```

Aggiungi utente AD a un gruppo di annunci

```
Add-ADGroupMember -Identity "My-First-Group" -Members "JohnSmith"
```

Nuovo gruppo di annunci

```
New-ADGroup -GroupScope Universal -Name "My-Second-Group"
```

computers

Recupera computer AD

```
Get-ADComputer -Identity "JohnLaptop"
```

Recupera tutte le proprietà associate al computer

```
Get-ADComputer -Identity "JohnLaptop" -Properties *
```

Recupera Seleziona Proprietà del computer

```
Get-ADComputer -Identity "JohnLaptop" -Properties * | Select-Object -Property Name, Enabled
```

Oggetti

Recupera un oggetto Active Directory

```
#Identity can be ObjectGUID, Distinguished Name or many more  
Get-ADObject -Identity "ObjectGUID07898"
```

Sposta un oggetto Active Directory

```
Move-ADObject -Identity "CN=JohnSmith,OU=Users,DC=Domain,DC=Local" -TargetPath  
"OU=SuperUser,DC=Domain,DC=Local"
```

Modifica un oggetto Active Directory

```
Set-ADObject -Identity "CN=My-First-Group,OU=Groups,DC=Domain,DC=local" -Description "This is  
My First Object Modification"
```

Leggi Modulo ActiveDirectory online: <https://riptutorial.com/it/powershell/topic/8213/modulo-activedirectory>

Capitolo 39: Modulo delle attività pianificate

introduzione

Esempi di come utilizzare il modulo Attività pianificate disponibile in Windows 8 / Server 2012 e in.

Examples

Esegui script PowerShell in Attività pianificata

Crea un'attività pianificata che viene eseguita immediatamente, quindi all'avvio per eseguire

C:\myscript.ps1 **come** 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
```

Leggi Modulo delle attività pianificate online:

<https://riptutorial.com/it/powershell/topic/10940/modulo-delle-attivit a-pianificate>

Capitolo 40: Modulo di archiviazione

introduzione

Il modulo `Archive` `Microsoft.PowerShell.Archive` fornisce funzioni per l'archiviazione di file negli archivi ZIP (`Compress-Archive`) ed estraendoli (`Expand-Archive`). Questo modulo è disponibile in PowerShell 5.0 e versioni successive.

Nelle versioni precedenti di PowerShell potrebbero essere utilizzate le [estensioni](#) della [community](#) o .NET [System.IO.Compression.FileSystem](#) .

Sintassi

- **Expand-Archive / Compress-Archive**
- **-Sentiero**
 - il percorso dei file da comprimere (Compress-Archive) o il percorso dell'archivio per estrarre il modulo dei file (Expand-Archive)
 - ci sono molte altre opzioni relative al percorso, vedi sotto.
- **-DestinationPath** (opzionale)
 - se non si fornisce questo percorso, l'archivio verrà creato nella directory di lavoro corrente (Compress-Archive) o il contenuto dell'archivio verrà estratto nella directory di lavoro corrente (Expand-Archive)

Parametri

| Parametro | Dettagli |
|------------------|---|
| CompressionLevel | (<i>Compress-Archive only</i>) Imposta il livello di compressione su <code>Fastest</code> , <code>Optimal</code> o <code>NoCompression</code> |
| Confermare | Richiede conferma prima di iniziare |
| Vigore | Forza l'esecuzione del comando senza conferma |
| LiteralPath | Percorso che viene utilizzato letteralmente, <i>senza caratteri jolly supportati</i> , utilizza <code>,</code> per specificare più percorsi |
| Sentiero | Percorso che può contenere caratteri jolly, utilizzare <code>,</code> per specificare più percorsi |
| Aggiornare | (<i>Comprimi solo archivio</i>) Aggiorna archivio esistente |
| Cosa succede se | Simula il comando |

Osservazioni

Vedere [MSDN Microsoft.PowerShell.Archive \(5.1\)](#) per ulteriori riferimenti

Examples

Comprimi-archivio con carattere jolly

```
Compress-Archive -Path C:\Documents\* -CompressionLevel Optimal -DestinationPath  
C:\Archives\Documents.zip
```

Questo comando:

- Comprime tutti i file in `C:\Documents`
- Utilizza una compressione `Optimal`
- Salvare l'archivio risultante in `C:\Archives\Documents.zip`
 - `-DestinationPath` aggiungerà `.zip` se non presente.
 - `-LiteralPath` può essere utilizzato se è necessario denominarlo senza `.zip`.

Aggiorna ZIP esistente con Compress-Archive

```
Compress-Archive -Path C:\Documents\* -Update -DestinationPath C:\Archives\Documents.zip
```

- questo aggiungerà o sostituirà tutti i file `Documents.zip` con i nuovi da `C:\Documents`

Estrai un file zip con Expand-Archive

```
Expand-Archive -Path C:\Archives\Documents.zip -DestinationPath C:\Documents
```

- questo estrae tutti i file da `Documents.zip` nella cartella `C:\Documents`

Leggi Modulo di archiviazione online: <https://riptutorial.com/it/powershell/topic/9896/modulo-di-archiviazione>

Capitolo 41: Modulo di SharePoint

Examples

Caricamento Snap-in di SharePoint

Il caricamento di SharePoint Snapin può essere effettuato utilizzando quanto segue:

```
Add-PSSnapin "Microsoft.SharePoint.PowerShell"
```

Funziona solo nella versione a 64 bit di PowerShell. Se la finestra dice "Windows PowerShell (x86)" nel titolo stai usando la versione errata.

Se lo Snap-In è già caricato, il codice sopra causerà un errore. L'utilizzo di quanto segue verrà caricato solo se necessario, che può essere utilizzato in Cmdlet / funzioni:

```
if ((Get-PSSnapin "Microsoft.SharePoint.PowerShell" -ErrorAction SilentlyContinue) -eq $null)
{
    Add-PSSnapin "Microsoft.SharePoint.PowerShell"
}
```

In alternativa, se si avvia SharePoint Management Shell, verrà automaticamente incluso lo snap-in.

Per ottenere un elenco di tutti i cmdlet SharePoint disponibili, eseguire quanto segue:

```
Get-Command -Module Microsoft.SharePoint.PowerShell
```

Iterazione su tutti gli elenchi di una raccolta siti

Stampa tutti i nomi delle liste e il numero degli articoli.

```
$site = Get-SPSite -Identity https://mysharepointsite/sites/test
foreach ($web in $site.AllWebs)
{
    foreach ($list in $web.Lists)
    {
        # Prints list title and item count
        Write-Output "$($list.Title), Items: $($list.ItemCount)"
    }
}
$site.Dispose()
```

Ottieni tutte le funzionalità installate in una raccolta siti

```
Get-SPFeature -Site https://mysharepointsite/sites/test
```

Get-SPFeature può anche essere eseguito su ambito Web (`-Web <WebUrl>`), farm scope (`-Farm`) e ambito applicazione Web (`-WebApplication <WebAppUrl>`).

Ottieni tutte le funzionalità orfane in una raccolta siti

Un altro utilizzo di Get-SPFeature può essere quello di trovare tutte le funzionalità che non hanno ambito:

```
Get-SPFeature -Site https://mysharepointsite/sites/test |? { $_.Scope -eq $null }
```

Leggi Modulo di SharePoint online: <https://riptutorial.com/it/powershell/topic/5147/modulo-di-sharepoint>

Capitolo 42: Modulo ISE

introduzione

Windows PowerShell Integrated Scripting Environment (ISE) è un'applicazione host che consente di scrivere, eseguire e testare script e moduli in un ambiente grafico e intuitivo. Le funzionalità principali di Windows PowerShell ISE includono la colorazione della sintassi, il completamento delle schede, Intellisense, il debug visivo, la conformità Unicode e la Guida sensibile al contesto e offrono una ricca esperienza di scripting.

Examples

Script di prova

L'uso semplice ma potente dell'ISE è ad esempio scrivere codice nella sezione superiore (con una colorazione intuitiva della sintassi) ed eseguire il codice semplicemente marcandolo e premendo il tasto F8.

```
function Get-Sum
{
    foreach ($i in $Input)
    {$Sum += $i}
    $Sum
}

1..10 | Get-Sum

#output
55
```

Leggi Modulo ISE online: <https://riptutorial.com/it/powershell/topic/10954/modulo-ise>

Capitolo 43: MongoDB

Osservazioni

La parte più difficile è quella di allegare un **documento secondario** nel documento che non è stato ancora creato se abbiamo bisogno che il documento secondario sia nel modo previsto, avremo bisogno di iterare con un ciclo for l'array in una variabile e usare `$doc2.add("Key", "Value")` invece utilizza l'array corrente `foreach` con indice. Questo renderà il documento secondario in due righe come puoi vedere nella sezione `"Tags" = [MongoDB.Bson.BsonDocument] $doc2 .`

Examples

MongoDB con il driver C # 1.7 usando PowerShell

Devo interrogare tutti i dettagli dalla macchina virtuale e aggiornarli in 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"
}
```

Ho 3 set di array in 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);
    }
}

```

Leggi MongoDB online: <https://riptutorial.com/it/powershell/topic/7438/mongodb>

Capitolo 44: Nome del cmdlet

introduzione

I CmdLets dovrebbero essere nominati usando uno schema di denominazione `<verb>-<noun>` per migliorare la rilevabilità.

Examples

verbi

I verbi usati per nominare CmdLets dovrebbero essere nominati dai verbi dall'elenco fornito da `Get-Verb`

Ulteriori dettagli su come utilizzare i verbi possono essere trovati su [Verbs approvati per Windows PowerShell](#)

I sostantivi

I nomi dovrebbero sempre essere singolari.

Sii coerente con i nomi. Ad esempio, `Find-Package` necessita di un provider, il nome è `PackageProvider ProviderPackage` .

Leggi Nome del cmdlet online: <https://riptutorial.com/it/powershell/topic/8703/nome-del-cmdlet>

Capitolo 45: operatori

introduzione

Un operatore è un personaggio che rappresenta un'azione. Indica al compilatore / interprete di eseguire operazioni matematiche, relazionali o logiche specifiche e di produrre risultati finali. PowerShell interpreta in modo specifico e categorizza di conseguenza, come gli operatori aritmetici eseguono operazioni principalmente su numeri, ma influenzano anche le stringhe e altri tipi di dati. Insieme agli operatori di base, PowerShell dispone di un numero di operatori che risparmia tempo e impegno nella codifica (ad esempio: -like, -match, -replace, ecc.).

Examples

Operatori aritmetici

```
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
```

Operatori logici

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

Operatori di assegnazione

Aritmetica semplice:

```
$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
```

Incremento e decremento:

```
$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
```

Operatori di confronto

Gli operatori di confronto di PowerShell sono composti da un trattino iniziale (-) seguito da un nome (`eq` per `equal` , `gt` per `greater than` , ecc ...).

I nomi possono essere preceduti da caratteri speciali per modificare il comportamento dell'operatore:

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

Case-Insensitive è il valore predefinito se non specificato, ("`a`" `-eq` "`A`") uguale a ("`a`" `-ieq` "`A`").

Operatori di confronto semplici:

```
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 (<=)
```

Operatori di confronto delle stringhe:

```
"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
```

Operatori di confronto delle collezioni:

```
"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)
```

Operatori di reindirizzamento

Flusso di output di successo:

```
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
```

Errore flusso di output:

```
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
```

Flusso di output di avviso: (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
```

Flusso di output dettagliato: (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
```

Debug output stream: (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
```

Flusso di output delle informazioni: (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
```

Tutti i flussi di output:

```
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
```

Differenze rispetto all'operatore del tubo (|)

Gli operatori di reindirizzamento reindirizzano solo i flussi a file o flussi su flussi. L'operatore del tubo pompa un oggetto lungo la pipeline a un cmdlet o all'output. In che modo la pipeline funziona differisce in generale dal modo in cui il reindirizzamento funziona e può essere letto in [Lavorare con la pipeline di PowerShell](#)

Tipi di operando di messaggio: il tipo dell'operando sinistro determina il comportamento.

Per aggiunta

```
"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"
```

Per moltiplicazione

```
"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'impatto può avere conseguenze nascoste sugli operatori di confronto:

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

Operatori di manipolazione delle stringhe

Sostituisci operatore:

L'operatore `-replace` sostituisce un modello in un valore di input utilizzando un'espressione regolare. Questo operatore utilizza due argomenti (separati da una virgola): un modello di espressione regolare e il suo valore di sostituzione (che è facoltativo e una stringa vuota per impostazione predefinita).

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

Dividi e unisciti agli operatori:

L'operatore `-split` suddivide una stringa in una serie di sottostringhe.

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

L'operatore `-join` unisce una serie di stringhe in una singola stringa.

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

Leggi operatori online: <https://riptutorial.com/it/powershell/topic/1071/operatori>

Capitolo 46: Operatori speciali

Examples

Operatore di espressione di array

Restituisce l'espressione come una matrice.

```
@(Get-ChildItem $env:windir\System32\ntdll.dll)
```

Restituirà un array con un oggetto

```
@(Get-ChildItem $env:windir\System32)
```

Restituirà un array con tutti gli elementi nella cartella (che non è un cambiamento di comportamento dall'espressione interna).

Operazione di chiamata

```
$command = 'Get-ChildItem'  
& $Command
```

Eseguirà `Get-ChildItem`

Operatore di acquisizione punti

```
.. \MyScript.ps1
```

gira `.\myScript.ps1` nell'attuale scope rendendo qualsiasi funzione e variabile disponibile nell'ambito corrente.

Leggi Operatori speciali online: <https://riptutorial.com/it/powershell/topic/8981/operatori-speciali>

Capitolo 47: Operazioni di base

introduzione

Un set è una collezione di oggetti che possono essere qualsiasi cosa. Qualunque operatore di cui abbiamo bisogno per lavorare su questi set sono in breve gli *operatori impostati* e l'operazione è anche nota come *operazione set*. L'operazione di base del set include Unione, intersezione e aggiunta, sottrazione, ecc.

Sintassi

- Group-Object
- Group-Object -Property <propertyName>
- Oggetto di gruppo: proprietà <propertyName>, <propertyName2>
- Oggetto di gruppo: proprietà <propertyName> -CaseSensitive
- Oggetto-gruppo -Proprietà <propertyName> -Culture <cultura>
- Oggetto-gruppo -Proprietà <ScriptBlock>
- Sort-Object
- Sort-Object -Property <propertyName>
- Sort-Object -Property <ScriptBlock>
- Sort-Object -Property <propertyName>, <propertyName2>
- Sort-Object -Property <propertyObject> -CaseSensitive
- Sort-Object -Property <propertyObject> -Descending
- Sort-Object -Property <propertyObject> -Unico
- Sort-Object -Property <propertyObject> -Culture <cultura>

Examples

Filtro: dove-oggetto / dove /?

Filtra un'enumerazione utilizzando un'espressione condizionale

Sinonimi:

```
Where-Object
where
?
```

Esempio:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")

$names | Where-Object { $_ -like "A*" }
$names | where { $_ -like "A*" }
$names | ? { $_ -like "A*" }
```

Ritorna:

```
Aaron
Albert
Alphonse
```

Ordinamento: Sort-Object / sort

Ordina un'enumerazione in ordine crescente o decrescente

Sinonimi:

```
Sort-Object
sort
```

assumendo:

```
$names = @( "Aaron", "Aaron", "Bernie", "Charlie", "Danny" )
```

Ordinamento crescente è l'impostazione predefinita:

```
$names | Sort-Object
$names | sort
```

```
Aaron
Aaron
Bernie
Charlie
Danny
```

Per richiedere l'ordine decrescente:

```
$names | Sort-Object -Descending
$names | sort -Descending
```

```
Danny
Charlie
```

Bernie
Aaron
Aaron

Puoi ordinare usando un'espressione.

```
$names | Sort-Object { $_.length }
```

Aaron
Aaron
Danny
Bernie
Charlie

Raggruppamento: gruppo-oggetto / gruppo

È possibile raggruppare un'enumerazione basata su un'espressione.

Sinonimi:

```
Group-Object  
group
```

Esempi:

```
$names = @( "Aaron", "Albert", "Alphonse","Bernie", "Charlie", "Danny", "Ernie", "Frank")  
  
$names | Group-Object -Property Length  
$names | group -Property Length
```

Risposta:

| Contare | Nome | Gruppo |
|---------|------|------------------------------|
| 4 | 5 | {Aaron, Danny, Ernie, Frank} |
| 2 | 6 | {Albert, Bernie} |
| 1 | 8 | { } Alphonse |
| 1 | 7 | {Charlie} |

Proiezione: Seleziona-Oggetto / seleziona

La proiezione di un'enumerazione consente di estrarre membri specifici di ciascun oggetto, di estrarre tutti i dettagli o di calcolare i valori per ciascun oggetto

Sinonimi:

```
Select-Object  
select
```

Selezione di un sottoinsieme delle proprietà:

```
$dir = dir "C:\MyFolder"  
  
$dir | Select-Object Name, FullName, Attributes  
$dir | select Name, FullName, Attributes
```

| Nome | Nome e cognome | attributi |
|----------|----------------------|-----------|
| immagini | C:\MyFolder\Images | elenco |
| data.txt | C:\MyFolder\data.txt | Archivio |
| source.c | C:\MyFolder\source.c | Archivio |

Selezionando il primo elemento e mostrandone tutte le proprietà:

```
$d | select -first 1 *
```

| |
|----------------|
| PSPath |
| PSParentPath |
| PSChildName |
| PSDrive |
| PSProvider |
| PSIsContainer |
| BaseName |
| Modalità |
| Nome |
| Genitore |
| esiste |
| Radice |
| Nome e cognome |
| Estensione |

| |
|-------------------|
| CreationTime |
| CreationTimeUtc |
| LastAccessTime |
| LastAccessTimeUtc |
| LastWriteTime |
| LastWriteTimeUtc |
| attributi |

Leggi Operazioni di base online: <https://riptutorial.com/it/powershell/topic/1557/operazioni-di-base>

Capitolo 48: Parametri comuni

Osservazioni

I parametri comuni possono essere utilizzati con qualsiasi cmdlet (ciò significa che non appena si contrassegna la funzione come cmdlet [si veda `CmdletBinding()`], si ottengono tutti questi parametri gratuitamente).

Ecco l'elenco di tutti i parametri comuni (l'alias è tra parentesi dopo il parametro corrispondente):

```
-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)
```

Examples

Parametro ErrorAction

I valori possibili sono `Continue` | `Ignore` | `Inquire` | `SilentlyContinue` | `Stop` | `Suspend` .

Il valore di questo parametro determinerà il modo in cui il cmdlet gestirà gli errori non terminanti (ad esempio quelli generati da `Write-Error`; per ulteriori informazioni sulla gestione degli errori vedere [*argomento non ancora creato*]).

Il valore predefinito (se questo parametro è omissso) è `Continue` .

-ErrorAction Continua

Questa opzione produrrà un messaggio di errore e continuerà con l'esecuzione.

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second comma
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : te
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorEx
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorExcepti
Second command
```

-ErrorAction Ignora

Questa opzione non produrrà alcun messaggio di errore e continuerà con l'esecuzione. Inoltre, nessun errore verrà aggiunto alla variabile automatica `$Error`.

Questa opzione è stata introdotta in 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 Informarsi

Questa opzione produrrà un messaggio di errore e richiederà all'utente di scegliere un'azione da intraprendere.

```
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) Y
```

-ErrorAction SilentlyContinue

Questa opzione non produrrà un messaggio di errore e continuerà con l'esecuzione. Tutti gli errori verranno aggiunti alla variabile automatica `$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

Questa opzione produrrà un messaggio di errore e non proseguirà con l'esecuzione.

```
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

Disponibile solo nei flussi di lavoro di PowerShell. Se utilizzato, se il comando viene eseguito in un errore, il flusso di lavoro viene sospeso. Ciò consente di indagare su tale errore e offre la possibilità di riprendere il flusso di lavoro. Per ulteriori informazioni sul sistema del flusso di lavoro, vedere [argomento non ancora creato].

Leggi Parametri comuni online: <https://riptutorial.com/it/powershell/topic/5951/parametri-comuni>

Capitolo 49: Passare la dichiarazione

introduzione

Un'istruzione `switch` consente di verificare una variabile per l'uguaglianza rispetto a un elenco di valori. Ogni valore è chiamato *caso* e la variabile che viene *attivata* viene controllata per ogni caso di commutazione. Ti consente di scrivere uno script che può scegliere tra una serie di opzioni, ma senza richiedere di scrivere una lunga serie di istruzioni `if`.

Osservazioni

Questo argomento sta documentando l' **istruzione `switch`** usata per ramificare il flusso dello script. Non confonderlo con i **parametri `switch`** che vengono utilizzati in funzioni come flag booleani.

Examples

Interruttore semplice

Le istruzioni `switch` confrontano un singolo valore di test con più condizioni ed eseguono tutte le azioni associate per confronti riusciti. Può risultare in più partite / azioni.

Dato il seguente interruttore ...

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

'First Action' verrà emesso se `$myValue` è impostato come 'First Condition' .

'Section Action' verrà emesso se `$myValue` è impostato come 'Second Condition' .

Nulla verrà emesso se `$myValue` non corrisponde a nessuna delle due condizioni.

Switch Statement con Regex Parameter

Il parametro `-Regex` consente alle istruzioni `switch` di eseguire la corrispondenza delle espressioni regolari rispetto alle condizioni.

Esempio:

```
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'}
}
```

Produzione:

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

Interruttore semplice con pausa

La parola chiave `break` può essere utilizzata nelle istruzioni `switch` per uscire dall'istruzione prima di valutare tutte le condizioni.

Esempio:

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

Produzione:

```
First Action
Second Action
```

A causa della parola chiave di `break` nella seconda azione, la terza condizione non viene valutata.

Passa istruzione con parametro jolly

Il parametro `-Wildcard` consente alle istruzioni `switch` di eseguire la corrispondenza con caratteri jolly rispetto alle condizioni.

Esempio:

```
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'}
}

```

Produzione:

```

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

```

Passa istruzione con parametro esatto

Il parametro `-Exact` impone le istruzioni switch per eseguire una corrispondenza esatta, senza distinzione tra maiuscole e minuscole e condizioni stringa.

Esempio:

```

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

```

Produzione:

```

First Action
Second Action
Third Action

```

Le azioni dalla prima alla terza vengono eseguite perché le condizioni associate corrispondono all'ingresso. Le stringhe regex e jolly nella quarta e quinta condizione non corrispondono.

Si noti che la quarta condizione corrisponderebbe anche alla stringa di input se veniva eseguita la corrispondenza delle espressioni regolari, ma in questo caso è stata ignorata perché non lo è.

Switch Statement con parametro CaseSensitive

Il parametro `-CaseSensitive` applica le istruzioni switch per eseguire una corrispondenza esatta e sensibile al maiuscolo / minuscolo rispetto alle condizioni.

Esempio:

```

switch -CaseSensitive ('Condition')
{
'condition'   {'First Action'}
}

```

```
'Condition'  {'Second Action'}
'condition'  {'Third Action'}
}
```

Produzione:

```
Second Action
```

La seconda azione è l'unica azione eseguita perché è l'unica condizione che corrisponde esattamente alla stringa 'Condition' quando si tiene conto della distinzione tra maiuscole e minuscole.

Passa l'istruzione con il parametro File

Il parametro `-file` consente all'istruzione `switch` di ricevere input da un file. Ogni riga del file viene valutata dall'istruzione `switch`.

File di esempio `input.txt` :

```
condition
test
```

Esempio di istruzione `switch`:

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

Produzione:

```
First Action
Second Action
```

Interruttore semplice con condizione predefinita

La parola chiave `Default` viene utilizzata per eseguire un'azione quando nessun'altra condizione corrisponde al valore di input.

Esempio:

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

```
    'Second Action'
  }
  Default
  {
    'Default Action'
  }
}
```

Produzione:

```
Default Action
```

Cambia istruzione con le espressioni

Le condizioni possono anche essere espressioni:

```
$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
```

Leggi [Passare la dichiarazione online](https://riptutorial.com/it/powershell/topic/1174/passare-la-dichiarazione): <https://riptutorial.com/it/powershell/topic/1174/passare-la-dichiarazione>

Capitolo 50: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output e Information

Osservazioni

<https://technet.microsoft.com/en-us/library/hh849921.aspx>

Examples

Write-Output

`Write-Output` genera output. Questa uscita può passare al comando successivo dopo la pipeline o alla console, quindi viene semplicemente visualizzata.

Il cmdlet invia oggetti lungo la pipeline principale, noto anche come "flusso di output" o "pipeline di successo". Per inviare oggetti di errore lungo la pipeline dell'errore, utilizzare `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. Il cmdlet `Write-Output` invia l'oggetto specificato lungo la pipeline al comando successivo.
2. Se il comando è l'ultimo comando nella pipeline, l'oggetto viene visualizzato nella console.
3. L'interprete di PowerShell considera questo come un `Write-Output` implicito.

Poiché il comportamento predefinito di `Write-Output` è quello di visualizzare gli oggetti alla fine di una pipeline, in genere non è necessario utilizzare il cmdlet. Ad esempio, `Get-Process | Write-Output` è equivalente a `Get-Process`.

Scrivi Preferenze

I messaggi possono essere scritti con;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
Write-Debug "Debug Message"
```

```
Write-Progress "Progress Message"
Write-Warning "Warning Message"
```

Ognuno di questi ha una variabile di preferenza;

```
$VerbosePreference = "SilentlyContinue"
$InformationPreference = "SilentlyContinue"
$DebugPreference = "SilentlyContinue"
$ProgressPreference = "Continue"
$WarningPreference = "Continue"
```

La variabile di preferenza controlla come vengono gestiti il messaggio e la successiva esecuzione dello 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"
```

Il colore dei messaggi può essere controllato per `Write-Error` impostando;

```
$host.PrivateData.ErrorBackgroundColor = "Black"
$host.PrivateData.ErrorForegroundColor = "Red"
```

Impostazioni simili sono disponibili per `Write-Verbose` , `Write-Debug` e `Write-Warning` .

Leggi PowerShell "Streams"; Debug, Verbose, Warning, Error, Output e Information online:
<https://riptutorial.com/it/powershell/topic/3255/powershell--streams---debug--verbose--warning--error--output-e-information>

Capitolo 51: PowerShell Parametri dinamici

Examples

Parametro dinamico "semplice"

Questo esempio aggiunge un nuovo parametro a `MyTestFunction` se `$SomeUsefulNumber` è maggiore di 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)
    }
}
```

Uso:

```
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
```

Nel secondo esempio di utilizzo, è possibile vedere chiaramente che manca un parametro.

I parametri dinamici sono anche presi in considerazione con il completamento automatico. Ecco cosa succede se premi Ctrl + Spazio alla fine della riga:

```
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
```

Leggi PowerShell Parametri dinamici online:

<https://riptutorial.com/it/powershell/topic/6704/powershell-parametri-dinamici>

Capitolo 52: Profili PowerShell

Osservazioni

Il file profilo è uno script PowerShell che verrà eseguito mentre la console di PowerShell viene avviata. In questo modo possiamo avere il nostro ambiente preparato per noi ogni volta che iniziamo una nuova sessione di PowerShell.

Le cose tipiche che vogliamo fare all'avvio di PowerShell sono:

- importazione di moduli che usiamo spesso (ActiveDirectory, Exchange, alcune specifiche DLL)
- registrazione
- cambiando il prompt
- diagnostica

Esistono diversi file di profilo e posizioni che hanno usi diversi e anche gerarchia di ordini di avvio:

| Ospite | Utente | Sentiero | Ordine di partenza | Variabile |
|---------|---------|---|--------------------|-----------------------------------|
| Tutti | Tutti | % Windir% \ system32 \ WindowsPowerShell \ v1.0 \ Profile.ps1 | 1 | \$ profile.AllUsersAllHosts |
| Tutti | attuale | % USERPROFILE% \ Documenti \ WindowsPowerShell \ Profile.ps1 | 3 | \$ profile.CurrentUserAllHosts |
| console | Tutti | % Windir% \ system32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShell_profile.ps1 | 2 | \$ profile.AllUsersCurrentHost |
| console | attuale | % USERPROFILE% \ Documenti \ WindowsPowerShell \ Microsoft.PowerShell_profile.ps1 | 4 | \$ profile.CurrentUserCurrentHost |
| ISE | Tutti | % Windir% \ system32 \ WindowsPowerShell \ v1.0 \ Microsoft.PowerShellISE_profile.ps1 | 2 | \$ profile.AllUsersCurrentHost |
| ISE | attuale | % USERPROFILE% \ Documenti \ WindowsPowerShell \ Microsoft.PowerShellISE_profile.ps1 | 4 | \$ profile.CurrentUserCurrentHost |

Examples

Crea un profilo di base

Un profilo PowerShell viene utilizzato per caricare automaticamente variabili e funzioni definite dall'utente.

I profili PowerShell non vengono creati automaticamente per gli utenti.

Per creare un profilo PowerShell `C:>New-Item -ItemType File $profile .`

Se sei in ISE puoi usare l'editor integrato `C:>psEdit $profile`

Un modo semplice per iniziare con il tuo profilo personale per l'host corrente è di salvare del testo nel percorso memorizzato nella variabile `$profile`

```
"#Current host, current user" > $profile
```

Ulteriori modifiche al profilo possono essere eseguite utilizzando PowerShell ISE, Blocco note, Codice di Visual Studio o qualsiasi altro editor.

La variabile `$profile` restituisce il profilo utente corrente per l'host corrente per impostazione predefinita, ma è possibile accedere al percorso per il criterio macchina (tutti gli utenti) e / o il profilo per tutti gli host (console, ISE, di terze parti) utilizzando `Get-Profile`.

```
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
```

Leggi Profili PowerShell online: <https://riptutorial.com/it/powershell/topic/5636/profili-powershell>

Capitolo 53: Proprietà calcolate

introduzione

Le proprietà calcolate in PowerShell sono proprietà derivate personalizzate (calcolate). Permette all'utente di formattare una certa proprietà nel modo in cui vuole che sia. Il calcolo (espressione) può essere molto probabilmente qualsiasi cosa.

Examples

Visualizza le dimensioni del file in KB - Proprietà calcolate

Consideriamo il seguente frammento,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

Invia semplicemente il contenuto della cartella con le proprietà selezionate. Qualcosa di simile a,

```
Name                CreationTime          Length
----                -
AnotherFile.txt     1/26/2017  2:45:02 PM  546000
filetomove.txt      1/5/2017    2:36:01 PM    5
```

Cosa succede se voglio visualizzare la dimensione del file in KB? È qui che le proprietà calcolate vengono a portata di mano.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1Kb}}
```

Che produce,

```
Name                Size_In_KB
----                -
AnotherFile.txt     533.203125
Secondfile.txt      1066.4111328125
```

L' `Expression` è ciò che contiene il calcolo per la proprietà calcolata. E sì, può essere qualsiasi cosa!

Leggi Proprietà calcolate online: <https://riptutorial.com/it/powershell/topic/8913/proprietà-calcolate>

Capitolo 54: PSScriptAnalyzer - PowerShell Script Analyzer

introduzione

PSScriptAnalyzer, <https://github.com/PowerShell/PSScriptAnalyzer>, è un controllo di codice statico per i moduli e gli script di Windows PowerShell. PSScriptAnalyzer controlla la qualità del codice di Windows PowerShell eseguendo una serie di regole basate sulle migliori pratiche di PowerShell identificate dal team e dalla community di PowerShell. Genera DiagnosticResults (errori e avvertenze) per informare gli utenti sui potenziali difetti del codice e suggerisce possibili soluzioni per miglioramenti.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Sintassi

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>]`

Examples

Analizzare gli script con i set di regole predefiniti incorporati

ScriptAnalyzer viene fornito con set di regole predefinite incorporate che possono essere utilizzate per analizzare gli script. Questi includono: `PSGallery`, `DSC` e `CodeFormatting`. Possono essere eseguiti come segue:

Regole di PowerShell Gallery

Per eseguire le regole di PowerShell Gallery, utilizzare il seguente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

Regole DSC

Per eseguire le regole DSC usa il seguente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Regole di formattazione del codice

Per eseguire le regole di formattazione del codice, utilizzare il seguente comando:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Analizzare gli script contro ogni regola incorporata

Per eseguire l'analizzatore di script su un singolo file di script, eseguire:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

Questo analizzerà il tuo script contro ogni regola incorporata. Se lo script è sufficientemente grande che potrebbe causare molti avvisi e / o errori.

Per eseguire l'analizzatore di script su un'intera directory, specificare la cartella contenente lo script, il modulo e i file DSC che si desidera analizzare. Specificare il parametro Recurse se si desidera anche che le sottodirectory cerchino i file da analizzare.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

Elenca tutte le regole integrate

Per vedere tutte le regole integrate eseguite:

```
Get-ScriptAnalyzerRule
```

Leggi PSScriptAnalyzer - PowerShell Script Analyzer online:

<https://riptutorial.com/it/powershell/topic/9619/psscriptanalyzer---powershell-script-analyzer>

Capitolo 55: query sql PowerShell

introduzione

Passando attraverso questo documento è possibile conoscere come utilizzare le query SQL con powershell

Parametri

| Articolo | Descrizione |
|------------------------------|--|
| \$ ServerInstance | Qui dobbiamo menzionare l'istanza in cui è presente il database |
| \$ Database | Qui dobbiamo menzionare il database in cui è presente la tabella |
| \$ query | Qui abbiamo la query che vogliamo eseguire in SQ |
| \$ Nome utente e \$ Password | UserName e Password che hanno accesso nel database |

Osservazioni

È possibile utilizzare la funzione seguente se nel caso non si è in grado di importare il modulo 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
    #
    $xlFileFormats = @{
        # 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;         #
    }

    $xl = New-Object -ComObject Excel.Application;
    $xl.DisplayAlerts = $false;
    $xl.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 = $xl.Workbooks.Add($_);
        if ($?) {
            $csvTemp = GetTempFileName(".csv");
            $ws = $wb.Worksheets.Item($Worksheet);
            $ws.SaveAs($csvTemp, $xlFileFormats[".csv"]);
            $wb.Close($false);
            Remove-Variable -Name ('ws', 'wb') -Confirm:$false;
            Import-Csv $csvTemp;
            Remove-Item $csvTemp -Confirm:$false -Verbose:$VerbosePreference;
        }
    }
}
}
}
}

End
{
    $xl.Quit();
    Remove-Variable -name xl -Confirm:$false;
    [gc]::Collect();
}
}
}

```

Examples

SQLExample

Per interrogare tutti i dati dalla tabella *MachineName* possiamo usare il comando come sotto uno.

\$ Query = "Seleziona * da MachineName"

\$ Inst = "ServerInstance"

\$ dbName = "DatabaseName"

\$ UID = "ID utente"

\$ Password = "password"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $dbName -query $Query -Username $UID -Password $Password
```

SQLQuery

Per interrogare tutti i dati dalla tabella *MachineName* possiamo usare il comando come sotto uno.

\$ Query = "Seleziona * da MachineName"

\$ Inst = "ServerInstance"

\$ dbName = "DatabaseName"

\$ UID = "ID utente"

\$ Password = "password"

```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password $Password
```

Leggi query sql PowerShell online: <https://riptutorial.com/it/powershell/topic/8217/query-sql-powershell>

Capitolo 56: Remot PowerShell

Osservazioni

- [about_Remote](#)
- [about_RemoteFAQ](#)
- [about_RemoteTroubleshooting](#)

Examples

Abilitazione di Remot PowerShell

Il servizio remoto di PowerShell deve essere prima abilitato sul server a cui si desidera connettersi in remoto.

```
Enable-PSRemoting -Force
```

Questo comando esegue quanto segue:

- Esegue il cmdlet Set-WSManQuickConfig, che esegue le seguenti attività:
- Avvia il servizio WinRM.
- Imposta il tipo di avvio sul servizio WinRM su Automatico.
- Crea un listener per accettare richieste su qualsiasi indirizzo IP, se non ne esiste già uno.
- Abilita un'eccezione firewall per le comunicazioni WS-Management.
- Registra le configurazioni di sessione Microsoft.PowerShell e Microsoft.PowerShell.Workflow, se non sono già state registrate.
- Registra la configurazione della sessione Microsoft.PowerShell32 su computer a 64 bit, se non è già registrata.
- Abilita tutte le configurazioni di sessione.
- Cambia il descrittore di sicurezza di tutte le configurazioni di sessione per consentire l'accesso remoto.
- Riavvia il servizio WinRM per rendere effettive le modifiche precedenti.

Solo per ambienti non di dominio

Per i server in un dominio AD l'autenticazione remota PS avviene tramite Kerberos ('Default'), o NTLM ('Negoziare'). Se si desidera consentire la comunicazione remota a un server non di dominio, sono disponibili due opzioni.

Configurare la comunicazione WSMan su HTTPS (che richiede la generazione di certificati) o abilitare l'autenticazione di base che invia le credenziali attraverso la codifica base64-codificata (che è fondamentalmente la stessa di testo in chiaro, quindi fai attenzione a questo).

In entrambi i casi dovrai aggiungere i sistemi remoti all'elenco di host attendibili di WSMan.

Abilitazione dell'autenticazione di base

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

Poi sul computer che si desidera connettersi *da*, si deve dire che a fidarsi di computer che si sta collegando.

```
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 *
```

Importante : devi dire al tuo cliente di considerare affidabile il computer indirizzato nel modo in cui vuoi connetterti (ad esempio se ti connetti tramite IP, deve avere fiducia nell'IP e non nel nome host)

Connessione a un server remoto tramite PowerShell

Utilizzo delle credenziali dal tuo computer locale:

```
Enter-PSSession 192.168.1.1
```

Richiesta di credenziali sul computer remoto

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Esegui comandi su un computer remoto

Una volta abilitato il servizio remoto di Powershell (Enable-PSRemoting) È possibile eseguire comandi sul computer remoto come questo:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {  
    Write host "Remote Computer Name: $ENV:ComputerName"  
}
```

Il metodo precedente crea una sessione temporanea e la chiude subito dopo il termine del comando o del blocco di script.

Per lasciare la sessione aperta ed eseguire successivamente un altro comando, è necessario prima creare una sessione remota:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Quindi puoi usare questa sessione ogni volta che invochi comandi sul computer remoto:

```
Invoke-Command -Session $Session -ScriptBlock {
    Write host "Remote Computer Name: $ENV:ComputerName"
}

Invoke-Command -Session $Session -ScriptBlock {
    Get-Date
}
```

Se è necessario utilizzare credenziali diverse, è possibile aggiungerle con il parametro `-Credential` :

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Avviso di serializzazione remota

Nota:

È importante sapere che il servizio remoto serializza gli oggetti PowerShell sul sistema remoto e li deserializza al termine della sessione remota, ovvero vengono convertiti in XML durante il trasporto e perdono tutti i loro metodi.

```
$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...
```

Considerando che hai i metodi sull'oggetto PS regolare:

```
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)

```

Uso di argomenti

Per utilizzare gli argomenti come parametri per il blocco di script remoto, è possibile utilizzare il parametro `ArgumentList` di `Invoke-Command` oppure utilizzare la sintassi `$Using:`

Usando `ArgumentList` con parametri senza nome (cioè nell'ordine in cui sono passati allo scriptblock):

```

$servicesToShow = "servicel"
$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
}

```

Utilizzo di `ArgumentList` con parametri denominati:

```

$servicesToShow = "servicel"
$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
}

```

Utilizzo di `$Using:` sintassi:

```

$servicesToShow = "servicel"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}

```

Una best practice per la pulizia automatica delle PSSession

Quando viene creata una sessione remota tramite il cmdlet `New-PSSession`, la `PSSession` persiste

fino al termine della sessione corrente di PowerShell. Ciò significa che, per impostazione predefinita, la `PSSession` e tutte le risorse associate continueranno ad essere utilizzate fino al termine della sessione corrente di PowerShell.

Più `PSSessions` attive possono diventare un problema per le risorse, in particolare per gli script di lunga durata o interconnessi che creano centinaia di `PSSessions` in una singola sessione di PowerShell.

È consigliabile rimuovere esplicitamente ciascuna `PSSession` dopo averla utilizzata. [1]

Il seguente modello di codice utilizza `try-catch-finally` per ottenere quanto sopra, combinando la gestione degli errori con un metodo sicuro per garantire che tutte le `PSSessions` create vengano rimosse quando vengono utilizzate:

```
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
    }
}
```

Riferimenti: [1] <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession>

Leggi Remot PowerShell online: <https://riptutorial.com/it/powershell/topic/3087/remot-powershell>

Capitolo 57: Riconoscimento di Amazon Web Services (AWS)

introduzione

Amazon Rekognition è un servizio che semplifica l'aggiunta di analisi delle immagini alle tue applicazioni. Con Riconoscimento, è possibile rilevare oggetti, scene e volti nelle immagini. Puoi anche cercare e confrontare facce. L'API di Rekognition consente di aggiungere rapidamente sofisticate ricerche visive basate sull'apprendimento e la classificazione delle immagini alle applicazioni.

Examples

Rileva etichette immagine con Rekognition AWS

```
$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
```

Dopo aver eseguito lo script in alto, i risultati dovrebbero essere stampati nell'host PowerShell che assomigli a quanto segue:

```
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
```

Utilizzando il modulo AWS PowerShell in combinazione con il servizio Rekognition di AWS, è possibile rilevare le etichette in un'immagine, ad esempio l'identificazione di oggetti in una stanza, gli attributi delle foto scattate e il livello di confidenza corrispondente rilevato da AWS Rekognition per ciascuno di questi attributi.

Il comando `Find-REKLabel` è quello che consente di richiamare una ricerca per questi attributi / etichette. Mentre è possibile fornire contenuto di immagine come array di byte durante la chiamata API, un metodo migliore è caricare i file di immagine su un bucket di AWS S3 e quindi indirizzare il

servizio di riconoscimento all'oggetto S3 che si desidera analizzare. L'esempio sopra mostra come ottenere ciò.

Confrontare la somiglianza facciale con il rekognition AWS

```
$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
```

Lo script di esempio fornito sopra dovrebbe fornire risultati simili ai seguenti:

```
Face                                     Similarity  
----                                     -  
Amazon.Rekognition.Model.ComparedFace 90
```

Il servizio Rekognition di AWS consente di eseguire un confronto facciale tra due foto. L'utilizzo di questo servizio è abbastanza semplice. Carica semplicemente due file immagine, che desideri confrontare, con un bucket AWS S3. Quindi, richiamare il comando `Compare-REKFace`, simile all'esempio sopra riportato. Ovviamente, dovrai fornire il tuo nome S3 Bucket e nomi di file univoci a livello globale.

Leggi Riconoscimento di Amazon Web Services (AWS) online:

<https://riptutorial.com/it/powershell/topic/9581/riconoscimento-di-amazon-web-services--aws->

Capitolo 58: Riga di comando PowerShell.exe

Parametri

| Parametro | Descrizione |
|--|--|
| -Aiuta -? /? | Mostra l'aiuto |
| -File <FilePath> [<Args>] | Percorso per file di script che deve essere eseguito e argomenti (facoltativo) |
| -Command {- <script-block> [-args <arg-array>] <string> [<CommandParameters>]} | Comandi da eseguire seguiti da argomenti |
| -EncodedCommand <Base64EncodedCommand> | Comandi codificati Base64 |
| -ExecutionPolicy <ExecutionPolicy> | Imposta la politica di esecuzione solo per questo processo |
| -InputFormat {Testo XML} | Imposta il formato di input per i dati inviati al processo. Testo (stringhe) o XML (CLIXML serializzato) |
| -Mta | PowerShell 3.0+: esegue PowerShell nell'appartamento a più thread (STA è l'impostazione predefinita) |
| -sta | PowerShell 2.0: esegue PowerShell in un apartment a thread singolo (MTA è l'impostazione predefinita) |
| -Uscita Vietata | Lascia la console PowerShell in esecuzione dopo aver eseguito lo script / comando |
| -nologo | Nasconde il banner di copyright al momento del lancio |
| -NonInteractive | Nasconde la console all'utente |
| -noprofile | Evitare il caricamento di profili PowerShell per macchina o utente |
| -OutputFormat {Testo XML} | Imposta il formato di output per i dati restituiti da PowerShell. Testo (stringhe) o XML (CLIXML serializzato) |

| Parametro | Descrizione |
|--|---|
| -PSConsoleFile <FilePath> | Carica un file di console pre-creato che configura l'ambiente (creato utilizzando <code>Export-Console</code>) |
| -Versione <versione di Windows PowerShell> | Specificare una versione di PowerShell da eseguire. Utilizzato principalmente con 2.0 |
| -WindowStyle <stile> | Specifica se avviare il processo PowerShell come una finestra <code>normal</code> , <code>hidden</code> , <code>minimized</code> o <code>maximized</code> . |

Examples

Esecuzione di un comando

Il parametro `-Command` viene utilizzato per specificare i comandi da eseguire all'avvio. Supporta più input di dati.

-Command <stringa>

È possibile specificare i comandi da eseguire all'avvio sotto forma di stringa. Punto ; virgola multiplo ; -Le dichiarazioni separate possono essere eseguite.

```
>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}

Il `-Command` parametro supporta anche un ingresso scriptblock (uno o più istruzioni avvolti tra parentesi graffe `{ #code }`). Questo funziona solo quando si chiamano `PowerShell.exe` da un altro Windows PowerShell-sessione.

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

-Command - (input standard)

Puoi passare i comandi dallo standard input usando `-Command -`. L'input standard può venire `echo`, dalla lettura di un file, da un'applicazione legacy della console, ecc.

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

Esecuzione di un file di script

È possibile specificare un file in `ps1` script per eseguirne il contenuto all'avvio usando il parametro `-File`.

Script di base

MyScript.ps1

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

Produzione:

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

Usando parametri e argomenti

È possibile aggiungere parametri e / o argomenti dopo il percorso file per utilizzarli nello script. Gli argomenti saranno usati come valori per parametri di script non definiti / disponibili, il resto sarà disponibile nel `$args` -array

MyScript.ps1

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

Produzione:

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

Leggi Riga di comando PowerShell.exe online: <https://riptutorial.com/it/powershell/topic/5839/riga-di-comando-powershell-exe>

Capitolo 59: Script di firma

Osservazioni

La firma di uno script renderà gli script conformi a tutte le politiche di esecuzione in PowerShell e garantirà l'integrità di uno script. Gli script firmati non funzioneranno se sono stati modificati dopo la firma.

La firma degli script richiede un certificato di firma del codice. raccomandazioni:

- Script / test personali (non condivisi): certificato dell'autorità certificata fidata (interna o di terze parti) **O** certificato autofirmato.
- Organizzazione interna condivisa: certificato dell'autorità certificata fidata (interna o di terze parti)
- Organizzazione esterna condivisa: certificato dell'autorità certificata di terze parti attendibili

Maggiori informazioni su [about_Signing @ TechNet](#)

Politiche di esecuzione

PowerShell ha criteri di esecuzione configurabili che controllano quali condizioni sono richieste per lo script o la configurazione da eseguire. È possibile impostare una politica di esecuzione per più ambiti; computer, utente corrente e processo corrente. **Le politiche di esecuzione possono essere facilmente aggirate e non sono progettate per limitare gli utenti, ma piuttosto per proteggerli dalla violazione involontaria delle politiche di firma.**

Le politiche disponibili sono:

| Ambientazione | Descrizione |
|------------------|---|
| Limitato | Non sono consentiti script |
| AllSigned | Tutti gli script devono essere firmati |
| RemoteSigned | Sono consentiti tutti gli script locali; solo script remoti firmati |
| illimitato | Nessun requisito. Tutti gli script sono consentiti, ma avviseranno prima di eseguire script scaricati da Internet |
| Circonvallazione | Tutti gli script sono consentiti e non vengono visualizzati avvisi |
| Non definito | Rimuovere la politica di esecuzione corrente per l'ambito corrente. Utilizza la politica padre. Se tutte le politiche non sono definite, verranno utilizzate restrizioni. |

È possibile modificare le politiche di esecuzione correnti utilizzando `Set-ExecutionPolicy -cmdlet`,

Criteria di gruppo o il parametro `-ExecutionPolicy` quando si avvia un processo `powershell.exe`.

Maggiori informazioni su [about_Execution_Policies @ TechNet](#)

Examples

Firma di uno script

La firma di uno script viene eseguita utilizzando `Set-AuthenticodeSignature -cmdlet` e un certificato di firma del codice.

```
#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
```

È anche possibile leggere un certificato da un `.pfx` -file utilizzando:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

Lo script sarà valido fino alla scadenza del certificato. Se si utilizza un timestamp-server durante la firma, lo script continuerà ad essere valido dopo la scadenza del certificato. È anche utile aggiungere la catena di fiducia per il certificato (inclusa l'autorità di root) per aiutare la maggior parte dei computer a fidarsi del certificato utilizzato per firmare lo script.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timestamp.dll"
```

Si consiglia di utilizzare un timestamp-server di un fornitore di certificati attendibili come Verisign, Comodo, Thawte, ecc.

Modifica della politica di esecuzione tramite `Set-ExecutionPolicy`

Per modificare la politica di esecuzione per l'ambito predefinito (`LocalMachine`), utilizzare:

```
Set-ExecutionPolicy AllSigned
```

Per modificare la politica per un ambito specifico, utilizzare:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

È possibile sopprimere i prompt aggiungendo l' `-Force`.

Bypassare i criteri di esecuzione per un singolo script

Spesso potrebbe essere necessario eseguire uno script non firmato che non è conforme al criterio

di esecuzione corrente. Un modo semplice per farlo è aggirando la politica di esecuzione per quel singolo processo. Esempio:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

Oppure puoi usare la stenografia:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Altre politiche di esecuzione:

| Politica | Descrizione |
|---------------|---|
| AllSigned | È possibile eseguire solo script firmati da un editore attendibile. |
| Bypass | Senza restrizioni; tutti gli script di Windows PowerShell possono essere eseguiti. |
| Default | Normalmente <code>RemoteSigned</code> , ma è controllato tramite ActiveDirectory |
| RemoteSigned | Gli script scaricati devono essere firmati da un editore fidato prima che possano essere eseguiti. |
| Restricted | Non è possibile eseguire script. Windows PowerShell può essere utilizzato solo in modalità interattiva. |
| Undefined | N / A |
| Unrestricted* | Simile al <code>bypass</code> |

Unrestricted : se si esegue uno script senza firma scaricato da Internet, viene richiesta l'autorizzazione prima dell'esecuzione.*

Ulteriori informazioni disponibili [qui](#) .

Ottieni la politica di esecuzione corrente

Ottenere la politica di esecuzione effettiva per la sessione corrente:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

Elencare tutte le politiche di esecuzione efficaci per la sessione corrente:

```
PS> Get-ExecutionPolicy -List
```

```
Scope ExecutionPolicy
-----
MachinePolicy          Undefined
UserPolicy             Undefined
Process                Undefined
CurrentUser            Undefined
LocalMachine           RemoteSigned
```

Elencare la politica di esecuzione per un ambito specifico, ad es. processi:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Ottenere la firma da uno script firmato

Ottieni informazioni sulla firma Authenticode da uno script firmato utilizzando `Get-AuthenticodeSignature` cmdlet:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Creazione di un certificato di firma del codice autofirmato per il test

Quando si firmano script personali o si esegue il test della firma del codice, può essere utile creare un certificato di firma del codice autofirmato.

5.0

A partire da PowerShell 5.0 è possibile generare un certificato di firma del codice autofirmato utilizzando `New-SelfSignedCertificate` cmdlet:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -
CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

Nelle versioni precedenti, è possibile creare un certificato autofirmato utilizzando lo strumento `makecert.exe` disponibile in .NET Framework SDK e Windows SDK.

Un certificato autofirmato sarà considerato affidabile solo dai computer che hanno installato il certificato. Per gli script che verranno condivisi, si consiglia un certificato da un'autorità di certificazione attendibile (interna o fidata di terze parti).

Leggi Script di firma online: <https://riptutorial.com/it/powershell/topic/5670/script-di-firma>

Capitolo 60: Servizio di archiviazione semplice Amazon Web Services (AWS) (S3)

introduzione

Questa sezione di documentazione si concentra sullo sviluppo contro il servizio di archiviazione semplice (S3) Amazon Web Services (AWS). S3 è davvero un servizio semplice con cui interagire. Crei "bucket" S3 che possono contenere zero o più "oggetti". Una volta creato un bucket, puoi caricare file o dati arbitrari nel bucket S3 come "oggetto". Si fa riferimento a oggetti S3, all'interno di un bucket, dalla "chiave" dell'oggetto (nome).

Parametri

| Parametro | Dettagli |
|---------------|---|
| BucketName | Il nome del bucket AWS S3 su cui stai operando. |
| CannedACLName | Il nome dell'elenco di controllo di accesso (ACL) predefinito (predefinito) che verrà associato al bucket S3. |
| File | Il nome di un file sul filesystem locale che verrà caricato su un bucket di AWS S3. |

Examples

Crea un nuovo S3 Bucket

```
New-S3Bucket -BucketName trevor
```

Il nome del bucket del servizio di archiviazione semplice (S3) deve essere globalmente univoco. Ciò significa che se qualcun altro ha già utilizzato il nome del bucket che si desidera utilizzare, è necessario decidere un nuovo nome.

Carica un file locale in un bucket S3

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'  
Write-S3Object -BucketName powershell -File myfile.txt
```

Il caricamento di file dal tuo filesystem locale in AWS S3 è facile, usando il comando `Write-S3Object`. Nella sua forma più semplice, è sufficiente specificare il parametro `-BucketName`, per indicare quale bucket S3 si desidera caricare un file e il parametro `-File`, che indica il percorso relativo o assoluto del file locale che si desidera caricare nel secchio S3.

Elimina un bucket S3

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force  
Remove-S3Bucket -BucketName powershell -Force
```

Per rimuovere un bucket S3, è prima necessario rimuovere tutti gli oggetti S3 che sono memorizzati all'interno del bucket, a condizione che si abbia il permesso di farlo. Nell'esempio precedente, stiamo recuperando un elenco di tutti gli oggetti all'interno di un bucket e quindi li `Remove-S3Object` comando `Remove-S3Object` per eliminarli. Una volta che tutti gli oggetti sono stati rimossi, possiamo usare il comando `Remove-S3Bucket` per cancellare il bucket.

Leggi [Servizio di archiviazione semplice Amazon Web Services \(AWS\) \(S3\) online](https://riptutorial.com/it/powershell/topic/9579/servizio-di-archiviazione-semplce-amazon-web-services--aws---s3-):
<https://riptutorial.com/it/powershell/topic/9579/servizio-di-archiviazione-semplce-amazon-web-services--aws---s3->

Capitolo 61: Set di parametri

introduzione

I **set di parametri** vengono utilizzati per limitare la possibile combinazione di parametri o per imporre l'utilizzo dei parametri quando uno o più parametri sono selezionati.

Gli esempi spiegheranno l'uso e la ragione di un set di parametri.

Examples

Set di parametri semplici

```
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 per forzare l'uso di un parametro quando viene selezionato un altro.

Quando si desidera, ad esempio, imporre l'uso del parametro Password se viene fornito il

parametro Utente. (e 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
```

Parametro impostato per limitare la combinazione di parametri

```
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
```

Leggi Set di parametri online: <https://riptutorial.com/it/powershell/topic/6598/set-di-parametri>

Capitolo 62: Sicurezza e crittografia

Examples

Calcolo dei codici hash di una stringa tramite .Net Cryptography

Utilizzo dello spazio dei nomi .Net `System.Security.Cryptography.HashAlgorithm` per generare il codice hash del messaggio con gli algoritmi supportati.

```
$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 parte "sha256" era l'algoritmo di hash utilizzato.

il - può essere rimosso o modificato in minuscolo

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-", "").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbc649790c0e589e462d3d
```

Se il formato base64 è stato preferito, utilizzare il convertitore base64 per l'output

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Leggi Sicurezza e crittografia online: <https://riptutorial.com/it/powershell/topic/5683/sicurezza-e-crittografia>

Capitolo 63: Splatting

introduzione

Lo splatting è un metodo per passare più parametri a un comando come una singola unità. Questo viene fatto memorizzando i parametri e i loro valori come coppie chiave-valore in una [tabella hash](#) e suddividendola in un cmdlet usando l'operatore splatting @ .

Lo splatting può rendere più leggibile un comando e consente di riutilizzare i parametri in più comandi.

Osservazioni

Nota: l' [operatore di espressione Array o @\(\)](#) hanno un comportamento molto diverso rispetto all'operatore Splatting @ .

Maggiori informazioni su [about_Splatting @ TechNet](#)

Examples

Splattare i parametri

Lo splatting viene effettuato sostituendo il simbolo \$ con l'operatore splatting @ quando si utilizza una variabile contenente un [HashTable](#) di parametri e valori in una chiamata di comando.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Senza splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

È possibile combinare parametri normali con parametri splattati per aggiungere facilmente parametri comuni alle chiamate.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters

Invoke-Command -ScriptBlock { "Something to excute remotely" } @MyParameters
```

Passare un parametro Switch usando Splatting

Per utilizzare Splatting per chiamare `Get-Process` con l' `-FileVersionInfo` simile a questo:

```
Get-Process -FileVersionInfo
```

Questa è la chiamata che utilizza lo splatting:

```
$MyParameters = @{  
    FileVersionInfo = $true  
}  
  
Get-Process @MyParameters
```

Nota: questo è utile perché puoi creare un set predefinito di parametri e fare la chiamata più volte in questo modo

```
$MyParameters = @{  
    FileVersionInfo = $true  
}  
  
Get-Process @MyParameters -Name WmiPrvSE  
Get-Process @MyParameters -Name explorer
```

Piping e Splatting

La dichiarazione dello splat è utile per riutilizzare insiemi di parametri più volte o con leggere variazioni:

```
$splat = @{  
    Class = "Win32_SystemEnclosure"  
    Property = "Manufacturer"  
    ErrorAction = "Stop"  
}  
  
Get-WmiObject -ComputerName $env:COMPUTERNAME @splat  
Get-WmiObject -ComputerName "Computer2" @splat  
Get-WmiObject -ComputerName "Computer3" @splat
```

Tuttavia, se lo splat non è rientrato per il riutilizzo, non si può desiderare di dichiararlo. Può essere convogliato invece:

```
@{  
    ComputerName = $env:COMPUTERNAME  
    Class = "Win32_SystemEnclosure"  
    Property = "Manufacturer"  
    ErrorAction = "Stop"  
} | % { Get-WmiObject @_ }
```

Splatting dalla funzione di primo livello a una serie di funzioni interne

Senza lo splatting è molto complicato cercare di passare i valori attraverso lo stack delle chiamate. Ma se si combina lo splatting con la potenza di **@PSBoundParameters**, è possibile passare la raccolta dei parametri di livello superiore attraverso i livelli.

```
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
```

Leggi Splatting online: <https://riptutorial.com/it/powershell/topic/5647/splatting>

Capitolo 64: stringhe

Sintassi

- Stringa "(quotata doppia)"
- 'Stringa letterale'
- @"
Qui-string
"@
- @'
Stringa qui letterale
'@

Osservazioni

Le stringhe sono oggetti che rappresentano il testo.

Examples

Creare una stringa di base

Stringa

Le stringhe vengono create avvolgendo il testo con virgolette doppie. Le stringhe con doppi apici possono valutare variabili e caratteri speciali.

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

Per utilizzare una virgoletta doppia all'interno di una stringa, è necessario eseguire l'escape usando il carattere di escape, backtick (`). Le virgolette singole possono essere utilizzate all'interno di una stringa doppia citazione.

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

Stringa letterale

Le stringhe letterali sono stringhe che non valutano variabili e caratteri speciali. Viene creato utilizzando virgolette singole.

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

Per utilizzare le virgolette singole all'interno di una stringa letterale, utilizzare doppie virgolette singole o una stringa di testo letterale. I doppi quto possono essere usati tranquillamente all'interno di una stringa letterale

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

Formato stringa

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

Le stringhe di formato possono essere utilizzate con l'operatore `-f` o il metodo statico `[String]::Format(string format, args)` **NET**.

Stringa multilinea

Esistono diversi modi per creare una stringa multilinea in PowerShell:

- È possibile utilizzare i caratteri speciali per il ritorno a capo e / o newline manualmente o utilizzare la variabile `NewLine` -environment per inserire il valore "newline" dei sistemi)

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

- Creare un interruzione di riga durante la definizione di una stringa (prima di chiudere la citazione)

```
"Hello  
World"
```

- Usando una stringa qui. *Questa è la tecnica più comune.*

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

Qui-string

Le stringhe Here sono molto utili quando si creano stringhe multilinea. Uno dei maggiori vantaggi rispetto ad altre stringhe multilinea è che è possibile utilizzare le virgolette senza doverle sfuggire utilizzando un backtick.

Qui-string

Le stringhe qui iniziano con `@` e una interruzione di riga e terminano con `"@` sulla propria riga (`"@` devono essere i primi caratteri sulla linea, nemmeno lo spazio / tabulazione).

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

Stringa qui letterale

È anche possibile creare una stringa qui letterale utilizzando le virgolette singole, quando non si desidera che qualsiasi espressione venga espansa come una normale stringa letterale.

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

Concatenazione di stringhe

Uso delle variabili in una stringa

È possibile concatenare stringhe utilizzando le variabili all'interno di una stringa con doppi apici. Questo non funziona con le proprietà.

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

Usando l'operatore +

Puoi anche unire le stringhe usando l'operatore `+`.

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

Questo funziona anche con le proprietà degli oggetti.

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

Utilizzo di sottoespressioni

L'output / risultato di una sottoespressione `$()` può essere usato in una stringa. Questo è utile quando si accede alle proprietà di un oggetto o si esegue un'espressione complessa. Le sottoespressioni possono contenere più istruzioni separate da punto ; virgola ;

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

Personaggi speciali

Quando viene usato all'interno di una stringa a virgolette doppie, il carattere di escape (backtick `) recupera un carattere speciale.

```
`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)
```

Esempio:

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

Puoi anche sfuggire a caratteri speciali con significati speciali:

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

Leggi stringhe online: <https://riptutorial.com/it/powershell/topic/5124/stringhe>

Capitolo 65: Usando la barra di avanzamento

introduzione

Una barra di avanzamento può essere utilizzata per mostrare che qualcosa si trova in un processo. È una funzionalità che consente di risparmiare tempo e di essere chiari. Le barre di avanzamento sono incredibilmente utili durante il debug per capire quale parte dello script è in esecuzione, e sono soddisfacenti per le persone che eseguono script per tenere traccia di ciò che sta accadendo. È comune visualizzare alcuni tipi di progresso quando uno script richiede molto tempo per essere completato. Quando un utente lancia lo script e non accade nulla, uno inizia a chiedersi se lo script sia stato lanciato correttamente.

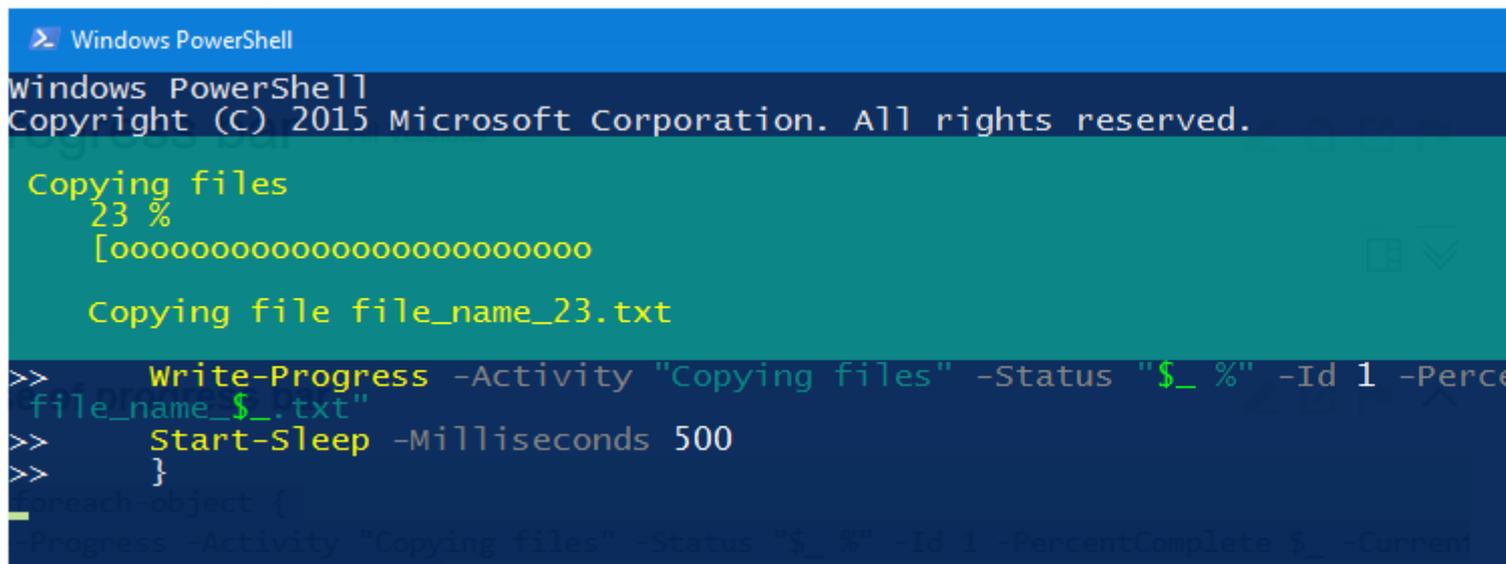
Examples

Semplice utilizzo della barra di avanzamento

```
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)
}
```

Si prega di notare che per brevità questo esempio non contiene alcun codice esecutivo (simulato con `Start-Sleep`). Tuttavia è possibile eseguirlo direttamente così com'è e modificarlo e giocarci.

Ecco come appaiono i risultati nella console PS:



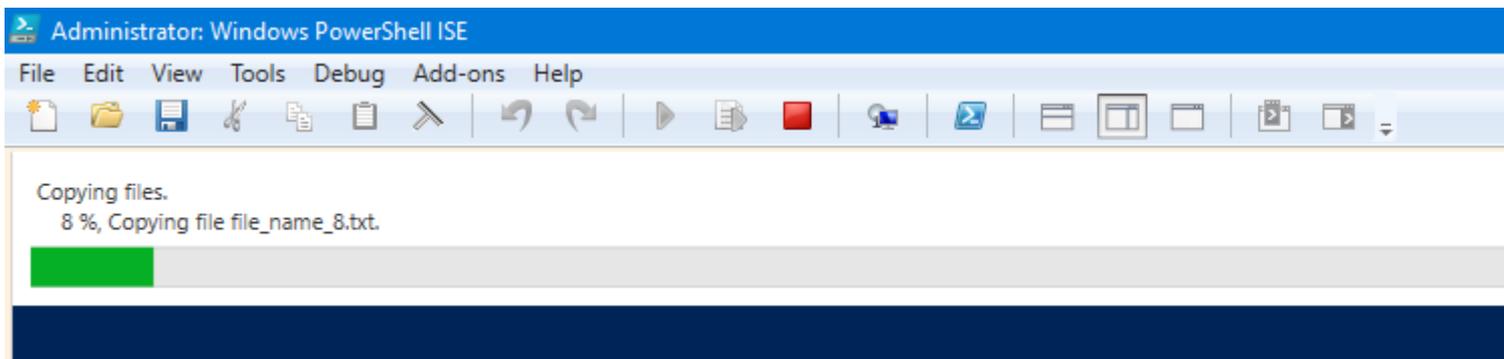
```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

Copying files
23 %
[ooooooooooooooooooooooooooooo

Copying file file_name_23.txt

>> Write-Progress -Activity "Copying files" -Status "$_ %" -Id 1 -Perce
file_name_$_.txt"
>> Start-Sleep -Milliseconds 500
>> }
```

Ecco come appare il risultato in PS ISE:



Utilizzo della barra di avanzamento interna

```
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)
}
```

Si prega di notare che per brevità questo esempio non contiene alcun codice esecutivo (simulato con `Start-Sleep`). Tuttavia è possibile eseguirlo direttamente così com'è e modificarlo e giocarci.

Ecco come appaiono i risultati nella console PS:

```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

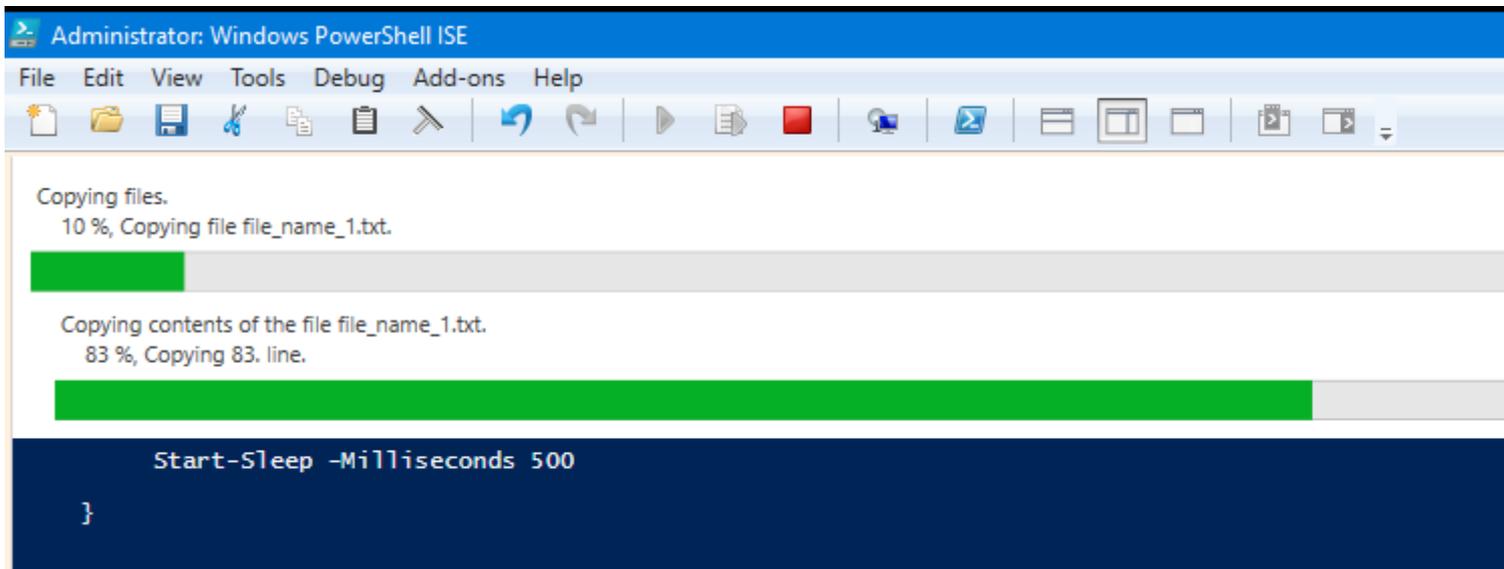
Copying files
30 %
[ooooooooooooooooooooooooooooooooooooo

Copying file file_name_3.txt
Copying contents of the file file_name_3.txt
46 %
[ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo

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
>>> }
```

Ecco come appare il risultato in PS ISE:



The screenshot shows the Windows PowerShell ISE interface. The console output displays two progress bars. The first bar is for 'Copying files' at 10% completion, with the text 'Copying file file_name_1.txt.' The second bar is for 'Copying contents of the file file_name_1.txt' at 83% completion, with the text 'Copying 83. line.' Below the progress bars, the PowerShell script code is visible, including the 'Start-Sleep -Milliseconds 500' command.

Leggi Usando la barra di avanzamento online:

<https://riptutorial.com/it/powershell/topic/5020/usando-la-barra-di-avanzamento>

Capitolo 66: Utilizzando le classi statiche esistenti

introduzione

Queste classi sono librerie di riferimento di metodi e proprietà che non cambiano stato, in una parola, immutabili. Non è necessario crearli, basta semplicemente usarli. Classi e metodi come questi sono chiamati classi statiche perché non vengono creati, distrutti o modificati. Puoi fare riferimento a una classe statica circondando il nome della classe con parentesi quadre.

Examples

Creazione immediata di nuovo GUID

Utilizza immediatamente le classi .NET esistenti con PowerShell usando `[class] :: Method (args)`:

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Allo stesso modo, in PowerShell 5+ è possibile utilizzare il cmdlet `New-Guid`:

```
PS C:\> New-Guid

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Per ottenere il GUID solo come `[String]`, fare riferimento alla proprietà `.Guid`:

```
[guid]::NewGuid().Guid
```

Utilizzo della classe `Math .Net`

È possibile utilizzare la classe `Math .Net` per eseguire calcoli (`[System.Math]`)

Se vuoi sapere quali metodi sono disponibili puoi usare:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Ecco alcuni esempi su come utilizzare 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
```

Aggiungere tipi

Per nome assembly, aggiungi libreria

```
Add-Type -AssemblyName "System.Math"
```

o dal percorso del file:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

Per utilizzare il tipo aggiunto:

```
[CustomMath.Namespace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Leggi Utilizzando le classi statiche esistenti online:

<https://riptutorial.com/it/powershell/topic/1522/utilizzando-le-classi-statiche-esistenti>

Capitolo 67: Utilizzando ShouldProcess

Sintassi

- `$ PSCmdlet.ShouldProcess ("Target")`
- `$ PSCmdlet.ShouldProcess ("Target", "Azione")`

Parametri

| Parametro | Dettagli |
|-----------|--|
| Bersaglio | La risorsa viene cambiata. |
| Azione | L'operazione che viene eseguita. Il valore predefinito è il nome del cmdlet. |

Osservazioni

`$PSCmdlet.ShouldProcess()` scriverà automaticamente un messaggio sull'output dettagliato.

```
PS> Invoke-MyCmdlet -Verbose  
VERBOSE: Performing the operation "Invoke-MyCmdlet" on target "Target of action"
```

Examples

Aggiunta di -WhatIf e -Confermare il supporto per il cmdlet

```
function Invoke-MyCmdlet {  
    [CmdletBinding(SupportsShouldProcess = $true)]  
    param()  
    # ...  
}
```

Utilizzare ShouldProcess () con un argomento

```
if ($PSCmdlet.ShouldProcess("Target of action")) {  
    # Do the thing  
}
```

Quando si utilizza `-WhatIf` :

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

Quando si utilizza `-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"):
```

Esempio di utilizzo completo

Altri esempi potrebbero non spiegarmi chiaramente come attivare la logica condizionale.

Questo esempio mostra anche che i comandi sottostanti ascolteranno anche il flag -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 binding
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
```

Leggi Utilizzando ShouldProcess online: <https://riptutorial.com/it/powershell/topic/1145/utilizzando-shouldprocess>

Capitolo 68: Utilizzo del sistema di guida

Osservazioni

`Get-Help` è un cmdlet per la lettura degli argomenti della Guida in PowerShell.

Leggi di più un [TechNet](#)

Examples

Aggiornamento del sistema di guida

3.0

A partire da PowerShell 3.0, è possibile scaricare e aggiornare la documentazione della guida offline utilizzando un singolo cmdlet.

```
Update-Help
```

Per aggiornare la guida su più computer (o computer non connessi a Internet).

Eseguire quanto segue su un computer con i file della guida

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

Per eseguire su molti computer da remoto

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath \\Server01\Share\Help -Credential $cred}
```

Utilizzando Get-Help

È possibile utilizzare `Get-Help` per visualizzare la guida in PowerShell. È possibile cercare cmdlet, funzioni, provider o altri argomenti.

Per visualizzare la documentazione della guida relativa ai lavori, utilizzare:

```
Get-Help about_Jobs
```

Puoi cercare argomenti usando i caratteri jolly. Se desideri elencare gli argomenti della guida disponibili con un titolo che inizia con `about_`, prova:

```
Get-Help about_*
```

Se volevi aiuto su `Select-Object`, dovresti usare:

```
Get-Help Select-Object
```

Puoi anche usare l' `help` degli alias o `man` .

Visualizzazione della versione online di un argomento della guida

È possibile accedere alla documentazione della guida in linea utilizzando:

```
Get-Help Get-Command -Online
```

Esempi di visualizzazione

Mostra esempi di utilizzo per un cmdlet specifico.

```
Get-Help Get-Command -Examples
```

Visualizzazione della pagina della Guida completa

Visualizza la documentazione completa per l'argomento.

```
Get-Help Get-Command -Full
```

Visualizzazione della guida per un parametro specifico

È possibile visualizzare la guida per un parametro specifico utilizzando:

```
Get-Help Get-Content -Parameter Path
```

Leggi **Utilizzo del sistema di guida online**: <https://riptutorial.com/it/powershell/topic/5644/utilizzo-del-sistema-di-guida>

Capitolo 69: variabili ambientali

Examples

Le variabili di ambiente di Windows sono visibili come un drive PS chiamato Env:

Puoi vedere la lista con tutte le variabili d'ambiente con:
Get-Childitem env:

Chiamata istantanea di variabili d'ambiente con \$ env:

```
$env:COMPUTERNAME
```

Leggi variabili ambientali online: <https://riptutorial.com/it/powershell/topic/5635/variabili-ambientali>

Capitolo 70: Variabili automatiche

introduzione

Le variabili automatiche sono create e gestite da Windows PowerShell. Si ha la possibilità di chiamare una variabile praticamente qualsiasi nome nel libro; Le uniche eccezioni a questo sono le variabili che sono già state gestite da PowerShell. Queste variabili, senza dubbio, saranno gli oggetti più ripetitivi che si usano in PowerShell accanto alle funzioni (come `$?` - indica lo stato Success / Failure dell'ultima operazione)

Sintassi

- `$$` - Contiene l'ultimo token nell'ultima riga ricevuta dalla sessione.
- `$^` - Contiene il primo token nell'ultima riga ricevuta dalla sessione.
- `$?` - Contiene lo stato di esecuzione dell'ultima operazione.
- `$_` - Contiene l'oggetto corrente nella pipeline

Examples

`$ pid`

Contiene l'ID di processo del processo di hosting corrente.

```
PS C:\> $pid
26080
```

Valori booleani

`$true` e `$false` sono due variabili che rappresentano logico TRUE e FALSE.

Si noti che è necessario specificare il simbolo del dollaro come primo carattere (che è diverso da 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
```

Si noti che quando si utilizza il valore booleano vero / falso nel codice si scrive `$true` o `$false` , ma quando Powershell restituisce un valore booleano, appare come `True` o `False`

`$ null`

`$null` viene utilizzato per rappresentare il valore assente o non definito.

`$null` può essere usato come segnaposto vuoto per il valore vuoto negli array:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

Quando utilizziamo la stessa matrice dell'origine per `ForEach-Object`, elaborerà tutti e tre gli elementi (incluso `$ null`):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Stai attento! Ciò significa che `ForEach-Object` **elaborerà** anche `$null` tutto da solo:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Il che è un risultato molto inaspettato se lo si confronta con il ciclo `foreach` classico:

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

\$ OFS

La variabile chiamata Output Field Separator contiene il valore di stringa che viene utilizzato durante la conversione di una matrice in una stringa. Di default `$OFS = " "` (*uno spazio*), ma può essere modificato:

```
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

Contiene l'oggetto / oggetto attualmente in fase di elaborazione dalla 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
```

`$PSItem` e `$_` sono identici e possono essere usati in modo intercambiabile, ma `$_` è di gran lunga il

più usato.

\$?

Contiene lo stato dell'ultima operazione. Quando non ci sono errori, è impostato su `True` :

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

Se c'è qualche errore, è impostato su `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
```

\$error

Matrice degli oggetti di errore più recenti. Il primo nell'array è il più recente:

```
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
```

Suggerimenti per l'utilizzo: quando si utilizza la variabile `$error` in un cmdlet di formato (ad esempio `format-list`), prestare attenzione all'utilizzo dell'interruttore `-Force` . In caso contrario, il cmdlet di formato produrrà i contenuti `$error` in modo sopra illustrato.

Le voci di errore possono essere rimosse tramite ad esempio `$Error.Remove($Error[0])` .

Leggi Variabili automatiche online: <https://riptutorial.com/it/powershell/topic/5353/variabili->

automatiche

Capitolo 71: Variabili automatiche - parte 2

introduzione

L'argomento "Variabili automatiche" contiene già 7 esempi elencati e non è possibile aggiungerne altri. Questo argomento avrà una continuazione di variabili automatiche.

Le variabili automatiche sono variabili che memorizzano le informazioni di stato per PowerShell. Queste variabili sono create e gestite da Windows PowerShell.

Osservazioni

Non sono sicuro se questo è il modo migliore per gestire la documentazione delle variabili automatiche, ma è meglio di niente. Si prega di commentare se trovate un modo migliore :)

Examples

\$ PSVersionTable

Contiene una tabella hash di sola lettura (costante, AllScope) che visualizza i dettagli sulla versione di PowerShell in esecuzione nella sessione corrente.

```
$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
```

Il modo più veloce per ottenere una versione di PowerShell in esecuzione:

```
$PSVersionTable.PSVersion
# result :
Major  Minor  Build  Revision
-----
5      0      10586  117
```

Leggi Variabili automatiche - parte 2 online: <https://riptutorial.com/it/powershell/topic/8639/variabili-automatiche---parte-2>

Capitolo 72: Variabili in PowerShell

introduzione

Le variabili sono utilizzate per la memorizzazione dei valori. Lascia che il valore sia di qualsiasi tipo, abbiamo bisogno di memorizzarlo da qualche parte in modo che possiamo usarlo attraverso la console / script. I nomi delle variabili in PowerShell iniziano con \$, come in \$ *Variable1* , e i valori sono assegnati usando = , come \$ **Variable1** = "**Valore 1**". PowerShell supporta un numero enorme di tipi di variabile; come stringhe di testo, numeri interi, decimali, matrici e persino tipi avanzati come numeri di versione o indirizzi IP.

Examples

Variabile semplice

Tutte le variabili in PowerShell iniziano con un segno di dollaro statunitense (\$). L'esempio più semplice di questo è:

```
$foo = "bar"
```

Questa affermazione alloca una variabile chiamata `foo` con un valore stringa di "bar".

Rimozione di una variabile

Per rimuovere una variabile dalla memoria, si può usare il cmdlet `Remove-Item` . Nota: il nome della variabile NON include \$.

```
Remove-Item Variable:\foo
```

`Variable` ha un provider che consente alla maggior parte dei cmdlet `*-item` di funzionare in modo molto simile ai file system.

Un altro metodo per rimuovere la variabile è usare il cmdlet `Remove-Variable` e il suo 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
```

Scopo

L' [ambito](#) predefinito per una variabile è il contenitore che lo contiene. Se all'esterno di uno script o

di un altro contenitore, l'ambito è `Global` . Per specificare un **ambito** , è preceduto dal nome della variabile `$scope:varname` modo:

```
$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
```

Produzione:

```
Global Scope
Function (local) scope
Function (local) scope
Global Scope
Global Scope
```

Lettura di un output CmdLet

Per impostazione predefinita, powershell restituisce l'output all'entità chiamante. Considera sotto l'esempio,

```
Get-Process -Name excel
```

Ciò semplicemente restituirebbe all'entità chiamante il processo in esecuzione che corrisponde al nome excel. In questo caso, l'host PowerShell. Stampa qualcosa come,

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | SI | ProcessName |
|---------|--------|-------|-------|-------|--------|------|----|-------------|
| ----- | ----- | ----- | ----- | ----- | ----- | -- | -- | ----- |
| 1037 | 54 | 67632 | 62544 | 617 | 5.23 | 4544 | 1 | EXCEL |

Ora se assegni l'output a una variabile, semplicemente non stamperà nulla. E naturalmente la variabile mantiene l'output. (Sia una stringa, Oggetto - Qualsiasi tipo per quella materia)

```
$allExcel = Get-Process -Name excel
```

Quindi, diciamo che hai uno scenario in cui vuoi assegnare una variabile con un nome dinamico, puoi usare il parametro `-OutVariable`

```
Get-Process -Name excel -OutVariable AllRunningExcel
```

Si noti che il '\$' manca qui. Una grande differenza tra questi due compiti è che, inoltre, stampa l'output oltre a assegnarlo alla variabile `AllRunningExcel`. Puoi anche scegliere di assegnarlo a

un'altra variabile.

```
$VarOne = Get-Process -Name excel -OutVariable VarTwo
```

Anche se lo scenario sopra è molto raro, entrambe le variabili \$ VarOne e \$ VarTwo avranno lo stesso valore.

Ora considera questo,

```
Get-Process -Name EXCEL -OutVariable MSOFFICE  
Get-Process -Name WINWORD -OutVariable +MSOFFICE
```

La prima istruzione otterrebbe semplicemente il processo excel e lo assegnerà alla variabile MSOFFICE, e in seguito otterrebbe i processi di parola ms in esecuzione e "Accoda" al valore esistente di MSOFFICE. Sembrerebbe qualcosa del genere,

| 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 |

Assegna lista di variabili multiple

Powershell consente l'assegnazione multipla di variabili e tratta quasi tutto come un array o un elenco. Ciò significa che invece di fare qualcosa del genere:

```
$input = "foo.bar.baz"  
$parts = $input.Split(".")  
$foo = $parts[0]  
$bar = $parts[1]  
$baz = $parts[2]
```

Puoi semplicemente fare questo:

```
$foo, $bar, $baz = $input.Split(".")
```

Poiché Powershell tratta gli incarichi in questo modo come gli elenchi, se nell'elenco sono presenti più valori rispetto agli elementi dell'elenco di variabili a cui assegnarli, l'ultima variabile diventa una matrice dei valori rimanenti. Ciò significa che puoi anche fare cose del genere:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]  
$bar = $leftover[0] # $bar = "bar"  
$baz = $leftover[1] # $baz = "baz"
```

Array

La dichiarazione di array in Powershell equivale quasi a creare un'istanza di qualsiasi altra variabile, ovvero si utilizza una sintassi `$name =` . Gli elementi dell'array vengono dichiarati separandoli con le virgole (,):

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Aggiungendo a un array

L'aggiunta a un array è semplice come utilizzare l'operatore + :

```
$myArrayOfInts = $myArrayOfInts + 5
//now contains 1,2,3,4 & 5!
```

Combinare insieme gli array

Ancora una volta questo è semplice come usare l'operatore +

```
$myArrayOfInts = 1,2,3,4
$myOtherArrayOfInts = 5,6,7
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts
//now 1,2,3,4,5,6,7
```

Leggi Variabili in PowerShell online: <https://riptutorial.com/it/powershell/topic/3457/variabili-in-powershell>

Capitolo 73: Variabili integrate

introduzione

PowerShell offre una varietà di utili variabili "automatiche" (integrate). Alcune variabili automatiche vengono popolate solo in circostanze particolari, mentre altre sono disponibili a livello globale.

Examples

\$ PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

Questo esempio recupera l'elenco di elementi secondari (directory e file) dalla cartella in cui si trova il file di script.

La variabile automatica `$PSScriptRoot` è `$null` se utilizzata dall'esterno di un file di codice PowerShell. Se utilizzato *all'interno di* uno script PowerShell, ha automaticamente definito il percorso del file system completo alla directory che contiene il file di script.

In Windows PowerShell 2.0, questa variabile è valida solo nei moduli di script (.psm1). A partire da Windows PowerShell 3.0, è valido in tutti gli script.

\$ Args

```
$Args
```

Contiene una matrice di parametri non dichiarati e / o valori di parametro passati a una funzione, script o blocco di script. Quando si crea una funzione, è possibile dichiarare i parametri utilizzando la parola chiave `param` o aggiungendo un elenco di parametri separati da virgole tra parentesi dopo il nome della funzione.

In un'azione evento, la variabile `$ Args` contiene oggetti che rappresentano gli argomenti evento dell'evento che viene elaborato. Questa variabile viene popolata solo all'interno del blocco Azione di un comando di registrazione eventi. Il valore di questa variabile può essere trovato anche nella proprietà `SourceArgs` dell'oggetto `PSEventArgs` (`System.Management.Automation.PSEventArgs`) restituito da `Get-Event`.

\$ PSItem

```
Get-Process | ForEach-Object -Process {  
    $PSItem.Name  
}
```

Come `$_`. Contiene l'oggetto corrente nell'oggetto pipeline. È possibile utilizzare questa variabile

nei comandi che eseguono un'azione su ogni oggetto o su oggetti selezionati in una pipeline.

\$?

```
Get-Process -Name doesnotexist  
Write-Host -Object "Was the last operation successful? $?"
```

Contiene lo stato di esecuzione dell'ultima operazione. Contiene TRUE se l'ultima operazione è riuscita e FALSE se fallisce.

\$ error

```
Get-Process -Name doesnotexist  
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Contiene una matrice di oggetti di errore che rappresentano gli errori più recenti. L'errore più recente è il primo oggetto di errore nella matrice (\$ Error [0]).

Per evitare che venga aggiunto un errore all'array \$ Error, utilizzare il parametro comune `ErrorAction` con il valore `Ignore`. Per ulteriori informazioni, vedere `about_CommonParameters` (<http://go.microsoft.com/fwlink/?LinkID=113216>) .

Leggi Variabili integrate online: <https://riptutorial.com/it/powershell/topic/8732/variabili-integrate>

Capitolo 74: WMI e CIM

Osservazioni

CIM vs WMI

A partire da PowerShell 3.0, esistono due modi per lavorare con le classi di gestione in PowerShell, WMI e CIM. PowerShell 1.0 e 2.0 supportavano solo il modulo WMI, che ora è sostituito dal nuovo e migliorato modulo CIM. In una versione successiva di PowerShell, i cmdlet WMI verranno rimossi.

Confronto tra moduli CIM e WMI:

| CIM-cmdlet | WMI-cmdlet | Cosa fa |
|-------------------------------|---------------------|--|
| Get-CimInstance | Get-WmiObject | Ottiene oggetti CIM / WMI per una classe |
| Invoke-CimMethod | Invoke-WmiMethod | Richiama un metodo di classe CIM / WMI |
| Registrati-CimIndicationEvent | Registrati-WmiEvent | Registra l'evento per una classe CIM / WMI |
| Rimuovere-CimInstance | Rimuovere-WmiObject | Rimuovi l'oggetto CIM / WMI |
| Set-CimInstance | Set-WmiInstance | Aggiorna / salva oggetto CIM / WMI |
| Get-CimAssociatedInstance | N / A | Ottieni istanze associate (oggetto / classi collegati) |
| Get-CIMClass | Get-WmiObject-List | Elenca le classi CIM / WMI |
| New-CimInstance | N / A | Crea un nuovo oggetto CIM |
| Get-CimSession | N / A | Elenca le sessioni CIM |
| New-CimSession | N / A | Crea una nuova sessione CIM |
| New-CimSessionOption | N / A | Crea oggetto con opzioni di sessione; protocollo, codifica, disabilitazione della crittografia ecc. (da utilizzare con <code>New-CimSession</code>) |

| CIM-cmdlet | WMI-cmdlet | Cosa fa |
|----------------------|------------|--------------------------------------|
| Rimuovere-CimSession | N / A | Rimuove / interrompe la sessione CIM |

Risorse aggiuntive

Dovrei usare CIM o WMI con Windows PowerShell? @ Ehi, Scripting Guy! blog

Examples

Interrogare oggetti

CIM / WMI è più comunemente usato per interrogare informazioni o configurazione su un dispositivo. Una classe che rappresenta una configurazione, un processo, un utente, ecc. In PowerShell esistono diversi modi per accedere a queste classi e istanze, ma i metodi più comuni sono l'utilizzo dei `Get-CimInstance` (CIM) o `Get-WmiObject` (WMI).

Elenca tutti gli oggetti per la classe CIM

Puoi elencare tutte le istanze di una 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
```

Utilizzando un filtro

È possibile applicare un filtro per ottenere solo istanze specifiche di una classe CIM / WMI. I filtri sono scritti usando `WQL` (predefinito) o `CQL` (add `-QueryDialect CQL`). `-Filter` utilizza la parte `WHERE` di una query `WQL` / `CQL` completa.

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
....
```

Utilizzando una query WQL:

È inoltre possibile utilizzare una query WQL / CQL per interrogare e filtrare le istanze.

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 |

Interrogazione di oggetti in un diverso spazio dei nomi:

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
```

Classi e spazi dei nomi

Esistono molte classi disponibili in CIM e WMI, separate in più spazi dei nomi. Lo spazio dei nomi più comune (e predefinito) in Windows è `root/cimv2`. Per trovare la classe `Right`, può essere utile elencare tutto o cercare.

Elenca le classi disponibili

È possibile elencare tutte le classi disponibili nello spazio dei nomi predefinito (`root/cimv2`) su un computer.

3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Cerca una classe

Puoi cercare classi specifiche usando i caratteri jolly. Es .: Trova le classi che contengono il process verbale.

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*"
```

Elenca le classi in un diverso spazio dei nomi

Lo spazio dei nomi di root è semplicemente chiamato `root` . È possibile elencare le classi in un altro spazio dei nomi utilizzando il parametro `-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"
```

Elenca gli spazi dei nomi disponibili

Per trovare gli spazi dei nomi figlio di `root` (o di un altro spazio dei nomi), interrogare gli oggetti nella classe `__NAMESPACE` per quello spazio dei nomi.

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"
```

Leggi WMI e CIM online: <https://riptutorial.com/it/powershell/topic/6808/wmi-e-cim>

Titoli di coda

| S. No | Capitoli | Contributors |
|-------|--|--|
| 1 | Introduzione a 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 , ошәә , Poorkenny , Sam Martin , th1rdey3 , TheIncorrigible1 , Tim , tjrobinson , TravisEz13 , vonPryz , Xalorous |
| 2 | alias | jumbo |
| 3 | Analisi CSV | Andrei Epure , Frode F. |
| 4 | Anonimizza IP (v4 e v6) nel file di testo con PowerShell | NooJ |
| 5 | Automazione dell'infrastruttura | Giulio Caccin , Ranadip Dutta |
| 6 | Classi di PowerShell | boeproX , Brant Bobby , Frode F. , Jaqueline Vanek , Mert Gülsoy , Ranadip Dutta , xvorsx |
| 7 | Codifica / decodifica URL | VertigoRay |
| 8 | Come scaricare l'ultimo artefatto da Artifactory usando lo script Powershell (v2.0 o inferiore)? | ANIL |
| 9 | Comportamento di restituzione in PowerShell | Bert Levräu , camilohe , Eris , jumbo , Ranadip Dutta , Thomas Gerot |
| 10 | Comunicazione con API RESTful | autosvet , Clijsters , HAL9256 , kdtong , RamenChef , Ranadip Dutta , Sam Martin , YChi Lu |
| 11 | Comunicazione TCP con PowerShell | autosvet , RamenChef , Richard |
| 12 | Configurazione dello stato desiderata | autosvet , CmdrTchort , Frode F. , RamenChef |

| | | |
|----|---|--|
| 13 | Convenzioni di denominazione | niksofteng |
| 14 | Creazione di risorse basate su classi DSC | Trevor Sullivan |
| 15 | Esecuzione di eseguibili | RamenChef , W1M0R |
| 16 | Espressioni regolari | Frode F. |
| 17 | Flussi di lavoro di PowerShell | Trevor Sullivan |
| 18 | Funzioni di PowerShell | Bert Levrau , Eris , James Ruskin , Luke Ryan , niksofteng , Ranadip Dutta , Richard , TessellatingHeckler , TravisEz13 , Xalorous |
| 19 | Gestione degli errori | Prageeth Saravanan |
| 20 | Gestione dei pacchetti | TravisEz13 |
| 21 | Gestione di segreti e credenziali | 4444 , briantist , Ranadip Dutta , TravisEz13 |
| 22 | GUI in Powershell | Sam Martin |
| 23 | Guida basata sui commenti | Christophe |
| 24 | hashtables | Florian Meyer , Ranadip Dutta , TravisEz13 |
| 25 | Imporre i prerequisiti di script | autosvet , Frode F. , jumbo , RamenChef |
| 26 | Incorporare codice gestito (C # VB) | ajb101 |
| 27 | Introduzione a Pester | Frode F. , Sam Martin |
| 28 | Introduzione a Psake | Roman |
| 29 | Invio di email | Adam M. , jimmyb , megamorf , NooJ , Ranadip Dutta , void , Yusuke Arakawa |
| 30 | Lavorare con gli oggetti | Chris N , djwork , Mathieu Buisson , megamorf |

| | | |
|----|--|---|
| 31 | Lavorare con i file XML | autosvet , Frode F. , Giorgio Gambino , Lieven Keersmaekers , RamenChef , Richard , Rowshi |
| 32 | Lavorare con la pipeline di PowerShell | Alban , Atsch , Clijsters , Deptor , James Ruskin , Keith , οοωα , Sam Martin |
| 33 | Lavori in background di PowerShell | Clijsters , mattnicola , Ranadip Dutta , Richard , TravisEz13 |
| 34 | Logica condizionale | Liam , lloyd , miken32 , TravisEz13 |
| 35 | Loops | 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 |
| 36 | Moduli PowerShell | autosvet , Mike Shepard , TravisEz13 , Trevor Sullivan |
| 37 | Moduli, script e funzioni | Frode F. , Ranadip Dutta , Xalorous |
| 38 | Modulo ActiveDirectory | Lachie White |
| 39 | Modulo delle attività pianificate | Sam Martin |
| 40 | Modulo di archiviazione | James Ruskin , RapidCoder |
| 41 | Modulo di SharePoint | Raziel |
| 42 | Modulo ISE | Florian Meyer |
| 43 | MongoDB | Thomas Gerot , Zteffer |
| 44 | Nome del cmdlet | TravisEz13 |
| 45 | operatori | Anthony Neace , Bevo , Clijsters , Gordon Bell , JPBlanc , Mark Wragg , Ranadip Dutta |
| 46 | Operatori speciali | TravisEz13 |
| 47 | Operazioni di base | Euro Micelli , Ranadip Dutta , TravisEz13 |
| 48 | Parametri comuni | autosvet , jumbo , RamenChef |

| | | |
|----|--|--|
| 49 | Passare la dichiarazione | Anthony Neace , Frode F. , jumbo , oowae , Ranadip Dutta , TravisEz13 |
| 50 | PowerShell "Streams"; Debug, Verbose, Warning, Error, Output e Information | DarkLite1 , Dave Anderson , megamorf |
| 51 | PowerShell Parametri dinamici | Poorkenny |
| 52 | Profili PowerShell | Frode F. , Kolob Canyon |
| 53 | Proprietà calcolate | Prageeth Saravanan |
| 54 | PSScriptAnalyzer - PowerShell Script Analyzer | Mark Wragg , mattnicola |
| 55 | query sql PowerShell | Venkatakrisnan |
| 56 | Remot PowerShell | Avshalom , megamorf , Moerwald , Sam Martin , ShaneC |
| 57 | Riconoscimento di Amazon Web Services (AWS) | Trevor Sullivan |
| 58 | Riga di comando PowerShell.exe | Frode F. |
| 59 | Script di firma | AP. , Frode F. |
| 60 | Servizio di archiviazione semplice Amazon Web Services (AWS) (S3) | Trevor Sullivan |
| 61 | Set di parametri | Bert Levrau , Poorkenny |
| 62 | Sicurezza e crittografia | YChi Lu |
| 63 | Splatting | autosvet , Frode F. , Moerwald , Petru Zaharia , Poorkenny , RamenChef , Ranadip Dutta , TravisEz13 , xXhRQ8sD2L7Z |
| 64 | stringhe | Frode F. , restless1987 , void |
| 65 | Usando la barra di | Clijsters , jumbo , Ranadip Dutta |

| | | |
|----|--|--|
| | avanzamento | |
| 66 | Utilizzando le classi statiche esistenti | Austin T French , briantist , motcke , Ranadip Dutta , Xenophane |
| 67 | Utilizzando ShouldProcess | Brant Bobby , Charlie Joynt , Schwarzie2478 |
| 68 | Utilizzo del sistema di guida | Frode F. , Madniz , mattnicola , RamenChef |
| 69 | variabili ambientali | autosvet |
| 70 | Variabili automatiche | Brant Bobby , jumbo , Mateusz Piotrowski , Moerwald , Ranadip Dutta , Roman |
| 71 | Variabili automatiche - parte 2 | Roman |
| 72 | Variabili in PowerShell | autosvet , Eris , Liam , Prageeth Saravanan , Ranadip Dutta , restless1987 , Steve K |
| 73 | Variabili integrate | Trevor Sullivan |
| 74 | WMI e CIM | Frode F. |