



FREE eBook

LEARNING PowerShell

Free unaffiliated eBook created from
Stack Overflow contributors.

#powershell

Table of Contents

| | |
|--|-----------|
| About..... | 1 |
| Chapter 1: Getting started with PowerShell..... | 2 |
| Remarks..... | 2 |
| Versions..... | 2 |
| Examples..... | 2 |
| Installation or Setup..... | 2 |
| Windows..... | 2 |
| Other Platforms..... | 3 |
| Allow scripts stored on your machine to run un-signed..... | 3 |
| Aliases & Similar Functions..... | 4 |
| The Pipeline - Using Output from a PowerShell cmdlet..... | 5 |
| Commenting..... | 6 |
| Calling .Net Library Methods..... | 6 |
| Creating Objects..... | 7 |
| Chapter 2: ActiveDirectory module..... | 9 |
| Introduction..... | 9 |
| Remarks..... | 9 |
| Examples..... | 9 |
| Module..... | 9 |
| Users..... | 9 |
| Groups..... | 10 |
| Computers..... | 10 |
| Objects..... | 10 |
| Chapter 3: Aliases..... | 12 |
| Remarks..... | 12 |
| Examples..... | 13 |
| Get-Alias..... | 13 |
| Set-Alias..... | 13 |
| Chapter 4: Amazon Web Services (AWS) Rekognition..... | 15 |
| Introduction..... | 15 |

| | |
|--|-----------|
| Examples..... | 15 |
| Detect Image Labels with AWS Rekognition..... | 15 |
| Compare Facial Similarity with AWS Rekognition..... | 16 |
| Chapter 5: Amazon Web Services (AWS) Simple Storage Service (S3)..... | 17 |
| Introduction..... | 17 |
| Parameters..... | 17 |
| Examples..... | 17 |
| Create a new S3 Bucket..... | 17 |
| Upload a Local File Into an S3 Bucket..... | 17 |
| Delete a S3 Bucket..... | 18 |
| Chapter 6: Anonymize IP (v4 and v6) in text file with Powershell..... | 19 |
| Introduction..... | 19 |
| Examples..... | 19 |
| Anonymize IP address in text file..... | 19 |
| Chapter 7: Archive Module..... | 21 |
| Introduction..... | 21 |
| Syntax..... | 21 |
| Parameters..... | 21 |
| Remarks..... | 21 |
| Examples..... | 22 |
| Compress-Archive with wildcard..... | 22 |
| Update existing ZIP with Compress-Archive..... | 22 |
| Extract a Zip with Expand-Archive..... | 22 |
| Chapter 8: Automatic Variables..... | 23 |
| Introduction..... | 23 |
| Syntax..... | 23 |
| Examples..... | 23 |
| \$pid..... | 23 |
| Boolean values..... | 23 |
| \$null..... | 23 |
| \$OFS..... | 24 |
| \$_ / \$PSItem..... | 24 |

| | |
|--|-----------|
| \$?..... | 25 |
| \$error..... | 25 |
| Chapter 9: Automatic Variables - part 2..... | 26 |
| Introduction..... | 26 |
| Remarks..... | 26 |
| Examples..... | 26 |
| \$PSVersionTable..... | 26 |
| Chapter 10: Basic Set Operations..... | 27 |
| Introduction..... | 27 |
| Syntax..... | 27 |
| Examples..... | 27 |
| Filtering: Where-Object / where / ?..... | 27 |
| Ordering: Sort-Object / sort..... | 28 |
| Grouping: Group-Object / group..... | 29 |
| Projecting: Select-Object / select..... | 29 |
| Chapter 11: Built-in variables..... | 32 |
| Introduction..... | 32 |
| Examples..... | 32 |
| \$PSScriptRoot..... | 32 |
| \$Args..... | 32 |
| \$PSItem..... | 32 |
| \$?..... | 33 |
| \$error..... | 33 |
| Chapter 12: Calculated Properties..... | 34 |
| Introduction..... | 34 |
| Examples..... | 34 |
| Display file size in KB - Calculated Properties..... | 34 |
| Chapter 13: Cmdlet Naming..... | 35 |
| Introduction..... | 35 |
| Examples..... | 35 |
| Verbs..... | 35 |
| Nouns..... | 35 |

| | |
|--|-----------|
| Chapter 14: Comment-based help | 36 |
| Introduction..... | 36 |
| Examples..... | 36 |
| Function comment-based help..... | 36 |
| Script comment-based help..... | 38 |
| Chapter 15: Common parameters | 41 |
| Remarks..... | 41 |
| Examples..... | 41 |
| ErrorAction parameter..... | 41 |
| -ErrorAction Continue | 41 |
| -ErrorAction Ignore | 41 |
| -ErrorAction Inquire | 42 |
| -ErrorAction SilentlyContinue | 42 |
| -ErrorAction Stop | 42 |
| -ErrorAction Suspend | 43 |
| Chapter 16: Communicating with RESTful APIs | 44 |
| Introduction..... | 44 |
| Examples..... | 44 |
| Use Slack.com Incoming Webhooks..... | 44 |
| Post Message to hipChat..... | 44 |
| Using REST with PowerShell Objects to Get and Put individual data..... | 44 |
| Using REST with PowerShell Objects to GET and POST many items..... | 45 |
| Using REST with PowerShell to Delete items..... | 45 |
| Chapter 17: Conditional logic | 46 |
| Syntax..... | 46 |
| Remarks..... | 46 |
| Examples..... | 46 |
| if, else and else if..... | 46 |
| Negation..... | 47 |
| If conditional shorthand..... | 47 |
| Chapter 18: Creating DSC Class-Based Resources | 49 |

| | |
|--|-----------|
| Introduction..... | 49 |
| Remarks..... | 49 |
| Examples..... | 49 |
| Create a DSC Resource Skeleton Class..... | 49 |
| DSC Resource Skeleton with Key Property..... | 49 |
| DSC Resource with Mandatory Property..... | 50 |
| DSC Resource with Required Methods..... | 50 |
| Chapter 19: CSV parsing..... | 52 |
| Examples..... | 52 |
| Basic usage of Import-Csv..... | 52 |
| Import from CSV and cast properties to correct type..... | 52 |
| Chapter 20: Desired State Configuration..... | 54 |
| Examples..... | 54 |
| Simple example - Enabling WindowsFeature..... | 54 |
| Starting DSC (mof) on remote machine..... | 54 |
| Importing psd1 (data file) into local variable..... | 54 |
| List available DSC Resources..... | 55 |
| Importing resources for use in DSC..... | 55 |
| Chapter 21: Embedding Managed Code (C# VB)..... | 56 |
| Introduction..... | 56 |
| Parameters..... | 56 |
| Remarks..... | 56 |
| Removing Added types..... | 56 |
| CSharp and .NET syntax..... | 56 |
| Examples..... | 57 |
| C# Example..... | 57 |
| VB.NET Example..... | 57 |
| Chapter 22: Enforcing script prerequisites..... | 59 |
| Syntax..... | 59 |
| Remarks..... | 59 |
| Examples..... | 59 |
| Enforce minimum version of powershell host..... | 59 |

| | |
|--|-----------|
| Enforce running the script as administrator..... | 59 |
| Chapter 23: Environment Variables..... | 61 |
| Examples..... | 61 |
| Windows environment variables are visible as a PS drive called Env:..... | 61 |
| Instant call of Environment Variables with \$env:..... | 61 |
| Chapter 24: Error handling..... | 62 |
| Introduction..... | 62 |
| Examples..... | 62 |
| Error Types..... | 62 |
| Chapter 25: GUI in Powershell..... | 64 |
| Examples..... | 64 |
| WPF GUI for Get-Service cmdlet..... | 64 |
| Chapter 26: Handling Secrets and Credentials..... | 66 |
| Introduction..... | 66 |
| Examples..... | 66 |
| Prompting for Credentials..... | 66 |
| Accessing the Plaintext Password..... | 66 |
| Working with Stored Credentials..... | 66 |
| Encrypter..... | 67 |
| The code that uses the stored credentials:..... | 67 |
| Storing the credentials in Encrypted form and Passing it as parameter when Required..... | 67 |
| Chapter 27: HashTables..... | 69 |
| Introduction..... | 69 |
| Remarks..... | 69 |
| Examples..... | 69 |
| Creating a Hash Table..... | 69 |
| Access a hash table value by key..... | 69 |
| Looping over a hash table..... | 70 |
| Add a key value pair to an existing hash table..... | 70 |
| Enumerating through keys and Key-Value Pairs..... | 70 |
| Remove a key value pair from an existing hash table..... | 71 |

| | |
|--|-----------|
| Chapter 28: How to download latest artifact from Artifactory using Powershell script (v2.0) | 72 |
| Introduction | 72 |
| Examples | 72 |
| Powershell Script for downloading the latest artifcat | 72 |
| Chapter 29: Infrastructure Automation | 73 |
| Introduction | 73 |
| Examples | 73 |
| Simple script for black-box integration test of console applications | 73 |
| Chapter 30: Introduction to Pester | 74 |
| Remarks | 74 |
| Examples | 74 |
| Getting Started with Pester | 74 |
| Chapter 31: Introduction to Psake | 76 |
| Syntax | 76 |
| Remarks | 76 |
| Examples | 76 |
| Basic outline | 76 |
| FormatTaskName example | 76 |
| Run Task conditionally | 77 |
| ContinueOnError | 77 |
| Chapter 32: ISE module | 78 |
| Introduction | 78 |
| Examples | 78 |
| Test Scripts | 78 |
| Chapter 33: Loops | 79 |
| Introduction | 79 |
| Syntax | 79 |
| Remarks | 79 |
| Foreach | 79 |
| Performance | 80 |
| Examples | 80 |

| | |
|--|-----------|
| For..... | 80 |
| Foreach..... | 80 |
| While..... | 81 |
| ForEach-Object..... | 81 |
| Basic usage..... | 82 |
| Advanced usage..... | 82 |
| Do..... | 83 |
| ForEach() Method..... | 83 |
| Continue..... | 84 |
| Break..... | 84 |
| Chapter 34: Modules, Scripts and Functions..... | 86 |
| Introduction..... | 86 |
| Examples..... | 86 |
| Function..... | 86 |
| Demo..... | 86 |
| Script..... | 87 |
| Demo..... | 87 |
| Module..... | 88 |
| Demo..... | 88 |
| Advanced Functions..... | 88 |
| Chapter 35: MongoDB..... | 92 |
| Remarks..... | 92 |
| Examples..... | 92 |
| MongoDB with C# driver 1.7 using PowerShell..... | 92 |
| I have 3 sets of array in Powershell..... | 92 |
| Chapter 36: Naming Conventions..... | 94 |
| Examples..... | 94 |
| Functions..... | 94 |
| Chapter 37: Operators..... | 95 |
| Introduction..... | 95 |
| Examples..... | 95 |
| Arithmetic Operators..... | 95 |

| | |
|--|------------|
| Logical Operators | 95 |
| Assignment Operators | 95 |
| Comparison Operators | 96 |
| Redirection Operators | 96 |
| Mixing operand types : the type of the left operand dictates the behavior..... | 97 |
| String Manipulation Operators | 98 |
| Chapter 38: Package management | 99 |
| Introduction | 99 |
| Examples | 99 |
| Find a PowerShell module using a pattern | 99 |
| Create the default PowerShell Module Repository | 99 |
| Find a module by name | 99 |
| Install a Module by name | 99 |
| Uninstall a module my name and version | 99 |
| Update a module by name | 99 |
| Chapter 39: Parameter sets | 101 |
| Introduction | 101 |
| Examples | 101 |
| Simple parameter sets | 101 |
| Parameterset to enforce the use of a parmeter when a other is selected..... | 101 |
| Parameter set to limit the combination of parmeters | 102 |
| Chapter 40: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output and Information .. | 103 |
| Remarks | 103 |
| Examples | 103 |
| Write-Output | 103 |
| Write Preferences | 103 |
| Chapter 41: PowerShell Background Jobs | 105 |
| Introduction | 105 |
| Remarks | 105 |
| Examples | 105 |
| Basic job creation | 105 |
| Basic job management | 106 |

| | |
|--|------------|
| Chapter 42: PowerShell Classes | 108 |
| Introduction..... | 108 |
| Examples..... | 108 |
| Methods and properties..... | 108 |
| Listing available constructors for a class..... | 108 |
| Constructor overloading..... | 110 |
| Get All Members of an Instance..... | 110 |
| Basic Class Template..... | 110 |
| Inheritance from Parent Class to Child Class..... | 111 |
| Chapter 43: PowerShell Dynamic Parameters | 112 |
| Examples..... | 112 |
| "Simple" dynamic parameter..... | 112 |
| Chapter 44: PowerShell Functions | 114 |
| Introduction..... | 114 |
| Examples..... | 114 |
| Simple Function with No Parameters..... | 114 |
| Basic Parameters..... | 114 |
| Mandatory Parameters..... | 115 |
| Advanced Function..... | 116 |
| Parameter Validation..... | 117 |
| ValidateSet..... | 117 |
| ValidateRange..... | 118 |
| ValidatePattern..... | 118 |
| ValidateLength..... | 118 |
| ValidateCount..... | 118 |
| ValidateScript..... | 118 |
| Chapter 45: Powershell Modules | 120 |
| Introduction..... | 120 |
| Examples..... | 120 |
| Create a Module Manifest..... | 120 |
| Simple Module Example..... | 120 |
| Exporting a Variable from a Module..... | 121 |

| | |
|---|------------|
| Structuring PowerShell Modules..... | 121 |
| Location of Modules..... | 122 |
| Module Member Visibility..... | 122 |
| Chapter 46: Powershell profiles..... | 123 |
| Remarks..... | 123 |
| Examples..... | 123 |
| Create an basic profile..... | 123 |
| Chapter 47: Powershell Remoting..... | 125 |
| Remarks..... | 125 |
| Examples..... | 125 |
| Enabling PowerShell Remoting..... | 125 |
| Only for non-domain environments..... | 125 |
| Enabling Basic Authentication..... | 125 |
| Connecting to a Remote Server via PowerShell..... | 126 |
| Run commands on a Remote Computer..... | 126 |
| Remoting serialization warning..... | 127 |
| Argument Usage..... | 128 |
| A best practise for automatically cleaning-up PSSessions..... | 128 |
| Chapter 48: powershell sql queries..... | 130 |
| Introduction..... | 130 |
| Parameters..... | 130 |
| Remarks..... | 130 |
| Examples..... | 132 |
| SQLExample..... | 132 |
| SQLQuery..... | 132 |
| Chapter 49: PowerShell Workflows..... | 134 |
| Introduction..... | 134 |
| Remarks..... | 134 |
| Examples..... | 134 |
| Simple Workflow Example..... | 134 |
| Workflow with Input Parameters..... | 134 |

| | |
|---|------------|
| Run Workflow as a Background Job..... | 135 |
| Add a Parallel Block to a Workflow..... | 135 |
| Chapter 50: PowerShell.exe Command-Line..... | 136 |
| Parameters..... | 136 |
| Examples..... | 137 |
| Executing a command..... | 137 |
| -Command <string>..... | 137 |
| -Command { scriptblock }..... | 137 |
| -Command - (standard input)..... | 137 |
| Executing a script file..... | 138 |
| Basic script..... | 138 |
| Using parameters and arguments..... | 138 |
| Chapter 51: PSScriptAnalyzer - PowerShell Script Analyzer..... | 139 |
| Introduction..... | 139 |
| Syntax..... | 139 |
| Examples..... | 139 |
| Analyzing scripts with the built-in preset rulesets..... | 139 |
| Analyzing scripts against every built-in rule..... | 140 |
| List all built-in rules..... | 140 |
| Chapter 52: Regular Expressions..... | 141 |
| Syntax..... | 141 |
| Examples..... | 141 |
| Single match..... | 141 |
| Using the -Match operator..... | 141 |
| Using Select-String..... | 142 |
| Using [Regex]::Match()..... | 143 |
| Replace..... | 143 |
| Using -Replace operator..... | 143 |
| Using [Regex]::Replace() method..... | 144 |
| Replace text with dynamic value using a MatchEvaluator..... | 144 |
| Escape special characters..... | 145 |

| | |
|--|------------|
| Multiple matches..... | 145 |
| Using Select-String..... | 146 |
| Using [RegEx]::Matches()..... | 146 |
| Chapter 53: Return behavior in PowerShell..... | 148 |
| Introduction..... | 148 |
| Remarks..... | 148 |
| Examples..... | 148 |
| Early exit..... | 148 |
| Gotcha! Return in the pipeline..... | 148 |
| Gotcha! Ignoring unwanted output..... | 149 |
| Return with a value..... | 149 |
| How to work with functions returns..... | 150 |
| Chapter 54: Running Executables..... | 152 |
| Examples..... | 152 |
| Console Applications..... | 152 |
| GUI Applications..... | 152 |
| Console Streams..... | 152 |
| Exit Codes..... | 153 |
| Chapter 55: Scheduled tasks module..... | 154 |
| Introduction..... | 154 |
| Examples..... | 154 |
| Run PowerShell Script in Scheduled Task..... | 154 |
| Chapter 56: Security and Cryptography..... | 155 |
| Examples..... | 155 |
| Calculating a string's hash codes via .Net Cryptography..... | 155 |
| Chapter 57: Sending Email..... | 156 |
| Introduction..... | 156 |
| Parameters..... | 156 |
| Examples..... | 157 |
| Simple Send-MailMessage..... | 157 |
| Send-MailMessage with predefined parameters..... | 157 |

| | |
|--|------------|
| SMTPClient - Mail with .txt file in body message..... | 157 |
| Chapter 58: SharePoint Module..... | 159 |
| Examples..... | 159 |
| Loading SharePoint Snap-In..... | 159 |
| Iterating over all lists of a site collection..... | 159 |
| Get all installed features on a site collection..... | 159 |
| Chapter 59: Signing Scripts..... | 161 |
| Remarks..... | 161 |
| Execution policies..... | 161 |
| Examples..... | 162 |
| Signing a script..... | 162 |
| Changing the execution policy using Set-ExecutionPolicy..... | 162 |
| Bypassing execution policy for a single script..... | 162 |
| Other Execution Policies:..... | 163 |
| Get the current execution policy..... | 163 |
| Getting the signature from a signed script..... | 164 |
| Creating a self-signed code signing certificate for testing..... | 164 |
| Chapter 60: Special Operators..... | 165 |
| Examples..... | 165 |
| Array Expression Operator..... | 165 |
| Call Operation..... | 165 |
| Dot sourcing operator..... | 165 |
| Chapter 61: Splatting..... | 166 |
| Introduction..... | 166 |
| Remarks..... | 166 |
| Examples..... | 166 |
| Splatting parameters..... | 166 |
| Passing a Switch parameter using Splatting..... | 167 |
| Piping and Splatting..... | 167 |
| Splatting From Top Level Function to a Series of Inner Function..... | 167 |
| Chapter 62: Strings..... | 169 |
| Syntax..... | 169 |

| | |
|---|------------|
| Remarks..... | 169 |
| Examples..... | 169 |
| Creating a basic string..... | 169 |
| String..... | 169 |
| Literal string..... | 169 |
| Format string..... | 170 |
| Multiline string..... | 170 |
| Here-string..... | 170 |
| Here-string..... | 170 |
| Literal here-string..... | 171 |
| Concatenating strings..... | 171 |
| Using variables in a string..... | 171 |
| Using the + operator..... | 171 |
| Using subexpressions..... | 171 |
| Special characters..... | 172 |
| Chapter 63: Switch statement..... | 173 |
| Introduction..... | 173 |
| Remarks..... | 173 |
| Examples..... | 173 |
| Simple Switch..... | 173 |
| Switch Statement with Regex Parameter..... | 173 |
| Simple Switch With Break..... | 174 |
| Switch Statement with Wildcard Parameter..... | 174 |
| Switch Statement with Exact Parameter..... | 175 |
| Switch Statement with CaseSensitive Parameter..... | 175 |
| Switch Statement with File Parameter..... | 176 |
| Simple Switch with Default Condition..... | 176 |
| Switch Statement with Expressions..... | 177 |
| Chapter 64: TCP Communication with PowerShell..... | 178 |
| Examples..... | 178 |
| TCP listener..... | 178 |

| | |
|--|------------|
| TCP Sender | 178 |
| Chapter 65: URL Encode/Decode | 180 |
| Remarks | 180 |
| Examples | 180 |
| Quick Start: Encoding | 180 |
| Quick Start: Decoding | 180 |
| Encode Query String with `[uri]::EscapeDataString()` | 181 |
| Encode Query String with `[System.Web.HttpUtility]::UrlEncode()` | 182 |
| Decode URL with `[uri]::UnescapeDataString()` | 182 |
| Decode URL with `[System.Web.HttpUtility]::UrlDecode()` | 184 |
| Chapter 66: Using existing static classes | 187 |
| Introduction | 187 |
| Examples | 187 |
| Creating new GUID instantly | 187 |
| Using the .Net Math Class | 187 |
| Adding types | 188 |
| Chapter 67: Using ShouldProcess | 189 |
| Syntax | 189 |
| Parameters | 189 |
| Remarks | 189 |
| Examples | 189 |
| Adding -WhatIf and -Confirm support to your cmdlet | 189 |
| Using ShouldProcess() with one argument | 189 |
| Full Usage Example | 190 |
| Chapter 68: Using the Help System | 192 |
| Remarks | 192 |
| Examples | 192 |
| Updating the Help System | 192 |
| Using Get-Help | 192 |
| Viewing online version of a help topic | 193 |
| Viewing Examples | 193 |
| Viewing the Full Help Page | 193 |

| | |
|---|------------|
| Viewing help for a specific parameter..... | 193 |
| Chapter 69: Using the progress bar..... | 194 |
| Introduction..... | 194 |
| Examples..... | 194 |
| Simple use of progress bar..... | 194 |
| Usage of inner progress bar..... | 195 |
| Chapter 70: Variables in PowerShell..... | 197 |
| Introduction..... | 197 |
| Examples..... | 197 |
| Simple variable..... | 197 |
| Removing a variable..... | 197 |
| Scope..... | 197 |
| Reading a CmdLet Output..... | 198 |
| List Assignment of Multiple Variables..... | 199 |
| Arrays..... | 199 |
| Adding to an array..... | 200 |
| Combining arrays together..... | 200 |
| Chapter 71: WMI and CIM..... | 201 |
| Remarks..... | 201 |
| CIM vs WMI..... | 201 |
| Additional resources..... | 201 |
| Examples..... | 202 |
| Querying objects..... | 202 |
| List all objects for CIM-class..... | 202 |
| Using a filter..... | 202 |
| Using a WQL-query:..... | 203 |
| Classes and namespaces..... | 204 |
| List available classes..... | 204 |
| Search for a class..... | 204 |
| List classes in a different namespace..... | 205 |
| List available namespaces..... | 206 |

| | |
|---|------------|
| Chapter 72: Working with Objects | 207 |
| Examples..... | 207 |
| Updating Objects..... | 207 |
| Adding properties | 207 |
| Removing properties | 207 |
| Creating a new object..... | 208 |
| Option 1: New-Object | 208 |
| Option 2: Select-Object | 208 |
| Option 3: pscustomobject type accelerator (PSv3+ required) | 209 |
| Examining an object..... | 209 |
| Creating Instances of Generic Classes..... | 210 |
| Chapter 73: Working with the PowerShell pipeline | 212 |
| Introduction..... | 212 |
| Syntax..... | 212 |
| Remarks..... | 212 |
| Examples..... | 212 |
| Writing Functions with Advanced Lifecycle..... | 212 |
| Basic Pipeline Support in Functions..... | 213 |
| Working concept of pipeline..... | 214 |
| Chapter 74: Working with XML Files | 215 |
| Examples..... | 215 |
| Accessing an XML File..... | 215 |
| Creating an XML Document using XmlWriter()..... | 217 |
| Adding snippets of XML to current XmlDocument..... | 218 |
| Sample Data | 218 |
| XML Document..... | 218 |
| New Data..... | 219 |
| Templates..... | 220 |
| Adding the new data | 220 |
| Profit | 222 |
| Improvements | 222 |

About

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

Chapter 1: Getting started with PowerShell

Remarks

Windows PowerShell is a shell and scripting component of the Windows Management Framework, an automation/configuration management framework from Microsoft built on the .NET Framework. PowerShell is installed by default on all supported versions of Windows client and server operating systems since Windows 7 / Windows Server 2008 R2. Powershell can be updated at any time by downloading a later version of the [Windows Management Framework](#) (WMF). The "Alpha" version of PowerShell 6 is cross-platform (Windows, Linux, and OS X) and needs to be downloaded and installed from [this release page](#).

Additional resources:

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

Versions

| Version | Included with Windows | Notes | Release Date |
|---------------------|--------------------------------------|-------|--------------|
| 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

Installation or Setup

Windows

PowerShell is included with the Windows Management Framework. Installation and Setup are not required on modern versions of Windows.

Updates to PowerShell can be accomplished by installing a newer version of the Windows Management Framework.

Other Platforms

"Beta" version of PowerShell 6 can be installed on other platforms. The installation packages are available [here](#).

For example, PowerShell 6, for Ubuntu 16.04, is published to package repositories for easy installation (and updates).

To install run the following:

```
# 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
```

After registering the Microsoft repository once as superuser, from then on, you just need to use `sudo apt-get upgrade powershell` to update it. Then just run `powershell`

Allow scripts stored on your machine to run un-signed

For security reasons, PowerShell is set up by default to only allow signed scripts to execute. Executing the following command will allow you to run unsigned scripts (you must run PowerShell as Administrator to do this).

```
Set-ExecutionPolicy RemoteSigned
```

Another way to run PowerShell scripts is to use `Bypass` as `ExecutionPolicy`:

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

Or from within your existing PowerShell console or ISE session by running:

```
Set-ExecutionPolicy Bypass Process
```

A temporary workaround for execution policy can also be achieved by running the Powershell executable and passing any valid policy as `-ExecutionPolicy` parameter. The policy is in effect only during process' lifetime, so no administrative access to the registry is needed.

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

There are multiple other policies available, and sites online often encourage you to use `Set-ExecutionPolicy Unrestricted`. This policy stays in place until changed, and lowers the system security stance. This is not advisable. Use of `RemoteSigned` is recommended because it allows locally stored and written code, and requires remotely acquired code be signed with a certificate from a trusted root.

Also, beware that the Execution Policy may be enforced by Group Policy, so that even if the policy is changed to `Unrestricted` system-wide, Group Policy may revert that setting at its next enforcement interval (typically 15 minutes). You can see the execution policy set at the various scopes using `Get-ExecutionPolicy -List`

TechNet Documentation:

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

Aliases & Similar Functions

In PowerShell, there are many ways to achieve the same result. This can be illustrated nicely with the simple & familiar `Hello World` example:

Using `Write-Host`:

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

Using `Write-Output`:

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

It's worth noting that although `Write-Output` & `Write-Host` both write to the screen there is a subtle difference. `Write-Host` writes *only* to stdout (i.e. the console screen), whereas `Write-Output` writes to both stdout *AND* to the output [success] stream allowing for [redirection](#). Redirection (and streams in general) allow for the output of one command to be directed as input to another including assignment to a variable.

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

These similar functions are not aliases, but can produce the same results if one wants to avoid "polluting" the success stream.

`Write-Output` is aliased to `Echo` or `Write`


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

Or, by simply typing 'Hello world'!

```
'Hello world'
```

All of which will result with the expected console output

```
Hello world
```

Another example of aliases in PowerShell is the common mapping of both older command prompt commands and BASH commands to PowerShell cmdlets. All of the following produce a directory listing of the current directory.

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

Finally, you can create your own alias with the Set-Alias cmdlet! As an example let's alias `Test-NetConnection`, which is essentially the PowerShell equivalent to the command prompt's ping command, to "ping".

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

Now you can use `ping` instead of `Test-NetConnection`! Be aware that if the alias is already in use, you'll overwrite the association.

The Alias will be alive, till the session is active. Once you close the session and try to run the alias which you have created in your last session, it will not work. To overcome this issue, you can import all your aliases from an excel into your session once, before starting your work.

The Pipeline - Using Output from a PowerShell cmdlet

One of the first questions people have when they begin to use PowerShell for scripting is how to manipulate the output from a cmdlet to perform another action.

The pipeline symbol `|` is used at the end of a cmdlet to take the data it exports and feed it to the next cmdlet. A simple example is using `Select-Object` to only show the Name property of a file shown from `Get-ChildItem`:

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

More advanced usage of the pipeline allows us to pipe the output of a cmdlet into a foreach loop:

```
Get-ChildItem | ForEach-Object {
```

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

Note that the example above uses the `$_` automatic variable. `$_` is the short alias of `$PSItem` which is an automatic variable which contains the current item in the pipeline.

Commenting

To comment on power scripts by prepending the line using the `#` (hash) symbol

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

You can also have multi-line comments using `<#` and `#>` at the beginning and end of the comment respectively.

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

Calling .Net Library Methods

Static .Net library methods can be called from PowerShell by encapsulating the full class name in third bracket and then calling the method using `::`

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

Static methods can be called from the class itself, but calling non-static methods requires an instance of the .Net class (an object).

For example, the `AddHours` method cannot be called from the `System.DateTime` class itself. It requires an instance of the class :

```
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 this case, we first [create an object](#), for example :

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

Then, we can use methods of that object, even methods which cannot be called directly from the `System.DateTime` class, like the `AddHours` method :

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

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

Creating Objects

The `New-Object` cmdlet is used to create an object.

```
# 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 many instances, a new object will be created in order to export data or pass it to another commandlet. This can be done like so:

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

There are many ways of creating an object. The following method is probably the shortest and fastest way to create a `PSCustomObject`:

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

In case you already have an object, but you only need one or two extra properties, you can simply add that property by using `Select-Object`:

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

All objects can be stored in variables or passed into the pipeline. You could also add these objects to a collection and then show the results at the end.

Collections of objects work well with `Export-CSV` (and `Import-CSV`). Each line of the CSV is an object, each column a property.

Format commands convert objects into text stream for display. Avoid using Format-* commands until the final step of any data processing, to maintain the usability of the objects.

Read **Getting started with PowerShell** online: <https://riptutorial.com/powershell/topic/822/getting-started-with-powershell>

Chapter 2: ActiveDirectory module

Introduction

This topic will introduce you to some of the basic cmdlets used within the Active Directory Module for PowerShell, for manipulating Users, Groups, Computers and Objects.

Remarks

Please remember that PowerShell's Help System is one of the best resources you can possibly utilize.

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

All of the help documentation will provide examples, syntax and parameter help.

Examples

Module

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

Users

Retrieve Active Directory User

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

Retrieve All Properties Associated with User

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

Retrieve Selected Properties for User

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

New AD User

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

Groups

Retrieve Active Directory Group

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

Retrieve All Properties Associated with Group

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

Retrieve All Members of a Group

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

Add AD User to an AD Group

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

New AD Group

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

Computers

Retrieve AD Computer

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

Retrieve All Properties Associated with Computer

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

Retrieve Select Properties of Computer

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

Objects

Retrieve an Active Directory Object

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

Move an Active Directory Object

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

Modify an Active Directory Object

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

Read ActiveDirectory module online: <https://riptutorial.com/powershell/topic/8213/activedirectory-module>

Chapter 3: Aliases

Remarks

Powershell naming system has quite strict rules of naming cmdlets (Verb-Noun template; see [topic not yet created] for more information). But it is not really convenient to write `Get-ChildItems` every time you want to list files in directory interactively.

Therefore Powershell enables using shortcuts - aliases - instead of cmdlet names.

You can write `ls`, `dir` or `gci` instead of `Get-ChildItem` and get the same result. Alias is equivalent to its cmdlet.

Some of the common aliases are:

| alias | cmdlet |
|-------------------------------|-------------------|
| %, foreach | For-EachObject |
| ?, where | Where-Object |
| cat, gc, type | Get-Content |
| cd, chdir, sl | Set-Location |
| cls, clear | Clear-Host |
| cp, copy, cpi | Copy-Item |
| dir/ls/gci | Get-ChildItem |
| echo, write | 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, move | Move-Item |
| rm, rmdir, del, erase, rd, ri | Remove-Item |

| alias | cmdlet |
|-------------|---------------|
| sleep | Start-Sleep |
| start, saps | Start-Process |

In the table above, you can see how aliases enabled simulating commands known from other environments (cmd, bash), hence increased discoverability.

Examples

Get-Alias

To list all aliases and their functions:

```
Get-Alias
```

To get all aliases for specific cmdlet:

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

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

To find aliases by matching:

```
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

This cmdlet allows you to create new alternate names for existing cmdlets

```
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 |
| | | | | | | | | |

Keep in mind that any alias you create will be persisted only in current session. When you start new session you need to create your aliases again. Powershell Profiles (see [topic not yet created]) are great for these purposes.

Read Aliases online: <https://riptutorial.com/powershell/topic/5287/aliases>

Chapter 4: Amazon Web Services (AWS) Rekognition

Introduction

Amazon Rekognition is a service that makes it easy to add image analysis to your applications. With Rekognition, you can detect objects, scenes, and faces in images. You can also search and compare faces. Rekognition's API enables you to quickly add sophisticated deep learning-based visual search and image classification to your applications.

Examples

Detect Image Labels with AWS Rekognition

```
$BucketName = 'trevorrekognition'  
$FileName = 'kitchen.jpg'  
  
New-S3Bucket -BucketName $BucketName  
Write-S3Object -BucketName $BucketName -File $FileName  
$REKResult = Find-REKLabel -Region us-east-1 -ImageBucket $BucketName -ImageName $FileName  
  
$REKResult.Labels
```

After running the script above, you should have results printed in your PowerShell host that look something similar to the following:

```
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
```

Using the AWS PowerShell module in conjunction with the AWS Rekognition service, you can detect labels in an image, such as identifying objects in a room, attributes about photos you took, and the corresponding confidence level that AWS Rekognition has for each of those attributes.

The `Find-REKLabel` command is the one that enables you to invoke a search for these attributes / labels. While you can provide image content as a byte array during the API call, a better method is to upload your image files to an AWS S3 Bucket, and then point the Rekognition service over to the S3 Objects that you want to analyze. The example above shows how to accomplish this.

Compare Facial Similarity with AWS Rekognition

```
$BucketName = 'trevorrekognition'

### Create a new AWS S3 Bucket
New-S3Bucket -BucketName $BucketName

### Upload two different photos of myself to AWS S3 Bucket
Write-S3Object -BucketName $BucketName -File myphoto1.jpg
Write-S3Object -BucketName $BucketName -File myphoto2.jpg

### Perform a facial comparison between the two photos with AWS Rekognition
$Comparison = @{
    SourceImageBucket = $BucketName
    TargetImageBucket = $BucketName
    SourceImageName = 'myphoto1.jpg'
    TargetImageName = 'myphoto2.jpg'
    Region = 'us-east-1'
}
$Result = Compare-REKFace @Comparison
$Result.FaceMatches
```

The example script provided above should give you results similar to the following:

| Face | Similarity |
|---------------------------------------|------------|
| ----- | ----- |
| Amazon.Rekognition.Model.ComparedFace | 90 |

The AWS Rekognition service enables you to perform a facial comparison between two photos. Using this service is quite straightforward. Simply upload two image files, that you want to compare, to an AWS S3 Bucket. Then, invoke the `Compare-REKFace` command, similar to the example provided above. Of course, you'll need to provide your own, globally-unique S3 Bucket name and file names.

Read [Amazon Web Services \(AWS\) Rekognition online](https://riptutorial.com/powershell/topic/9581/amazon-web-services--aws--rekognition):

<https://riptutorial.com/powershell/topic/9581/amazon-web-services--aws--rekognition>

Chapter 5: Amazon Web Services (AWS) Simple Storage Service (S3)

Introduction

This documentation section focuses on developing against the Amazon Web Services (AWS) Simple Storage Service (S3). S3 is truly a simple service to interact with. You create S3 "buckets" which can contain zero or more "objects." Once you create a bucket, you can upload files or arbitrary data into the S3 bucket as an "object." You reference S3 objects, inside of a bucket, by the object's "key" (name).

Parameters

| Parameter | Details |
|---------------|--|
| BucketName | The name of the AWS S3 bucket that you are operating on. |
| CannedACLName | The name of the built-in (pre-defined) Access Control List (ACL) that will be associated with the S3 bucket. |
| File | The name of a file on the local filesystem that will be uploaded to an AWS S3 Bucket. |

Examples

Create a new S3 Bucket

```
New-S3Bucket -BucketName trevor
```

The Simple Storage Service (S3) bucket name must be globally unique. This means that if someone else has already used the bucket name that you want to use, then you must decide on a new name.

Upload a Local File Into an S3 Bucket

```
Set-Content -Path myfile.txt -Value 'PowerShell Rocks'  
Write-S3Object -BucketName powershell -File myfile.txt
```

Uploading files from your local filesystem into AWS S3 is easy, using the `Write-S3Object` command. In its most basic form, you only need to specify the `-BucketName` parameter, to indicate which S3 bucket you want to upload a file into, and the `-File` parameter, which indicates the relative or absolute path to the local file that you want to upload into the S3 bucket.

Delete a S3 Bucket

```
Get-S3Object -BucketName powershell | Remove-S3Object -Force  
Remove-S3Bucket -BucketName powershell -Force
```

In order to remove a S3 bucket, you must first remove all of the S3 objects that are stored inside of the bucket, provided you have permission to do so. In the above example, we are retrieving a list of all the objects inside a bucket, and then piping them into the `Remove-S3Object` command to delete them. Once all of the objects have been removed, we can use the `Remove-S3Bucket` command to delete the bucket.

Read [Amazon Web Services \(AWS\) Simple Storage Service \(S3\)](https://riptutorial.com/powershell/topic/9579/amazon-web-services--aws--simple-storage-service--s3-) online:

<https://riptutorial.com/powershell/topic/9579/amazon-web-services--aws--simple-storage-service--s3->

Chapter 6: Anonymize IP (v4 and v6) in text file with Powershell

Introduction

Manipulating Regex for IPv4 and IPv6 and replacing by fake IP address in a readed log file

Examples

Anonymize IP address in text file

```
# 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  
}
```

Read Anonymize IP (v4 and v6) in text file with Powershell online:

<https://riptutorial.com/powershell/topic/9171/anonymize-ip--v4-and-v6--in-text-file-with-powershell>

Chapter 7: Archive Module

Introduction

The Archive module `Microsoft.PowerShell.Archive` provides functions for storing files in ZIP archives (`Compress-Archive`) and extracting them (`Expand-Archive`). This module is available in PowerShell 5.0 and above.

In earlier versions of PowerShell the [Community Extensions](#) or `.NET System.IO.Compression.FileSystem` could be used.

Syntax

- **Expand-Archive / Compress-Archive**
- **-Path**
 - the path of the file(s) to compress (`Compress-Archive`) or the path of the archive to extract the file(s) form (`Expand-Archive`)
 - there are several other Path related options, please see below.
- **-DestinationPath** (optional)
 - if you do not supply this path, the archive will be created in the current working directory (`Compress-Archive`) or the contents of the archive will be extracted into the current working directory (`Expand-Archive`)

Parameters

| Parameter | Details |
|------------------|--|
| CompressionLevel | <i>(Compress-Archive only)</i> Set compression level to either <code>Fastest</code> , <code>Optimal</code> or <code>NoCompression</code> |
| Confirm | Prompts for confirmation before running |
| Force | Forces the command to run without confirmation |
| LiteralPath | Path that is used literally, <i>no wildcards supported</i> , use <code>,</code> to specify multiple paths |
| Path | Path that can contain wildcards, use <code>,</code> to specify multiple paths |
| Update | <i>(Compress-Archive only)</i> Update existing archive |
| WhatIf | Simulate the command |

Remarks

See [MSDN Microsoft.PowerShell.Archive \(5.1\)](#) for further reference

Examples

Compress-Archive with wildcard

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

This command:

- Compresses all files in `C:\Documents`
- Uses `Optimal` compression
- Save the resulting archive in `C:\Archives\Documents.zip`
 - `-DestinationPath` will add `.zip` if not present.
 - `-LiteralPath` can be used if you require naming it without `.zip`.

Update existing ZIP with Compress-Archive

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

- this will add or replace all files `Documents.zip` with the new ones from `C:\Documents`

Extract a Zip with Expand-Archive

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

- this will extract all files from `Documents.zip` into the folder `C:\Documents`

Read Archive Module online: <https://riptutorial.com/powershell/topic/9896/archive-module>

Chapter 8: Automatic Variables

Introduction

Automatic Variables are created and maintained by Windows PowerShell. One has the ability to call a variable just about any name in the book; The only exceptions to this are the variables that are already being managed by PowerShell. These variables, without a doubt, will be the most repetitious objects you use in PowerShell next to functions (like `$?` - indicates Success/ Failure status of the last operation)

Syntax

- `$$` - Contains the last token in the last line received by the session.
- `$^` - Contains the first token in the last line received by the session.
- `$?` - Contains the execution status of the last operation.
- `$_` - Contains the current object in the pipeline

Examples

`$pid`

Contains process ID of the current hosting process.

```
PS C:\> $pid
26080
```

Boolean values

`$true` and `$false` are two variables that represent logical TRUE and FALSE.

Note that you have to specify the dollar sign as the first character (which is different from 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
```

Notice that when you use boolean true/false in your code you write `$true` or `$false`, but when Powershell returns a boolean, it looks like `True` or `False`

`$null`

`$null` is used to represent absent or undefined value.

`$null` can be used as an empty placeholder for empty value in arrays:

```
PS C:\> $array = 1, "string", $null
PS C:\> $array.Count
3
```

When we use the same array as the source for `ForEach-Object`, it will process all three items (including `$null`):

```
PS C:\> $array | ForEach-Object {"Hello"}
Hello
Hello
Hello
```

Be careful! This means that `ForEach-Object` **WILL** process even `$null` all by itself:

```
PS C:\> $null | ForEach-Object {"Hello"} # THIS WILL DO ONE ITERATION !!!
Hello
```

Which is very unexpected result if you compare it to classic `foreach` loop:

```
PS C:\> foreach($i in $null) {"Hello"} # THIS WILL DO NO ITERATION
PS C:\>
```

\$OFS

Variable called Output Field Separator contains string value that is used when converting an array to a string. By default `$OFS = " "` (a space), but it can be changed:

```
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

Contains the object/item currently being processed by the 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
```

`$PSItem` and `$_` are identical and can be used interchangeably, but `$_` is by far the most commonly used.

\$?

Contains status of the last operation. When there is no error, it is set to `True`:

```
PS C:\> Write-Host "Hello"
Hello
PS C:\> $?
True
```

If there is some error, it is set to `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

Array of most recent error objects. The first one in the array is the most recent one:

```
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
```

Usage hints: When using the `$error` variable in a format cmdlet (e.g. `format-list`), be aware to use the `-Force` switch. Otherwise the format cmdlet is going to output the `$error` contents in above shown manner.

Error entries can be removed via e.g. `$Error.Remove($Error[0])`.

Read Automatic Variables online: <https://riptutorial.com/powershell/topic/5353/automatic-variables>

Chapter 9: Automatic Variables - part 2

Introduction

Topic "Automatic Variables" already has 7 examples listed and we can't add more. This topic will have a continuation of Automatic Variables.

Automatic Variables are variables that store state information for PowerShell. These variables are created and maintained by Windows PowerShell.

Remarks

Not sure if this is the best way to handle documenting Automatic Variables, yet this is better than nothing. Please comment if you find a better way :)

Examples

\$PSVersionTable

Contains a read-only hash table (Constant, AllScope) that displays details about the version of PowerShell that is running in the current session.

```
$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
```

The fastest way to get a version of PowerShell running:

```
$PSVersionTable.PSVersion
# result :
Major Minor Build Revision
-----
5      0      10586 117
```

Read Automatic Variables - part 2 online: <https://riptutorial.com/powershell/topic/8639/automatic-variables---part-2>

Chapter 10: Basic Set Operations

Introduction

A set is a collection of items which can be anything. Whatever operator we need to work on these sets are in short the *set operators* and the operation is also known as *set operation*. Basic set operation includes Union, Intersection as well as addition, subtraction, etc.

Syntax

- Group-Object
- Group-Object -Property <propertyName>
- Group-Object -Property <propertyName>, <propertyName2>
- Group-Object -Property <propertyName> -CaseSensitive
- Group-Object -Property <propertyName> -Culture <culture>
- Group-Object -Property <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> -Unique
- Sort-Object -Property <propertyObject> -Culture <culture>

Examples

Filtering: Where-Object / where / ?

Filter an enumeration by using a conditional expression

Synonyms:

```
Where-Object
```

```
where
?
```

Example:

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

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

Returns:

```
Aaron
Albert
Alphonse
```

Ordering: Sort-Object / sort

Sort an enumeration in either ascending or descending order

Synonyms:

```
Sort-Object
sort
```

Assuming:

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

Ascending sort is the default:

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

```
Aaron
Aaron
Bernie
Charlie
Danny
```

To request descending order:

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

```
Danny
Charlie
Bernie
```


Aaron
Aaron

You can sort using an expression.

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

Aaron
Aaron
Danny
Bernie
Charlie

Grouping: Group-Object / group

You can group an enumeration based on an expression.

Synonyms:

```
Group-Object  
group
```

Examples:

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

Response:

| Count | Name | Group |
|-------|------|------------------------------|
| 4 | 5 | {Aaron, Danny, Ernie, Frank} |
| 2 | 6 | {Albert, Bernie} |
| 1 | 8 | {Alphonse} |
| 1 | 7 | {Charlie} |

Projecting: Select-Object / select

Projecting an enumeration allows you to extract specific members of each object, to extract all the details, or to compute values for each object

Synonyms:

```
Select-Object  
select
```

Selecting a subset of the properties:

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

| Name | FullName | Attributes |
|----------|----------------------|------------|
| Images | C:\MyFolder\Images | Directory |
| data.txt | C:\MyFolder\data.txt | Archive |
| source.c | C:\MyFolder\source.c | Archive |

Selecting the first element, and show all its properties:

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

| |
|---------------|
| PSPath |
| PSParentPath |
| PSChildName |
| PSDrive |
| PSProvider |
| PSIsContainer |
| BaseName |
| Mode |
| Name |
| Parent |
| Exists |
| Root |
| FullName |
| Extension |

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

Read Basic Set Operations online: <https://riptutorial.com/powershell/topic/1557/basic-set-operations>

Chapter 11: Built-in variables

Introduction

PowerShell offers a variety of useful "automatic" (built-in) variables. Certain automatic variables are only populated in special circumstances, while others are available globally.

Examples

\$PSScriptRoot

```
Get-ChildItem -Path $PSScriptRoot
```

This example retrieves the list of child items (directories and files) from the folder where the script file resides.

The `$PSScriptRoot` automatic variable is `$null` if used from outside a PowerShell code file. If used *inside* a PowerShell script, it automatically defined the fully-qualified filesystem path to the directory that contains the script file.

In Windows PowerShell 2.0, this variable is valid only in script modules (.psm1). Beginning in Windows PowerShell 3.0, it is valid in all scripts.

\$Args

```
$Args
```

Contains an array of the undeclared parameters and/or parameter values that are passed to a function, script, or script block. When you create a function, you can declare the parameters by using the `param` keyword or by adding a comma-separated list of parameters in parentheses after the function name.

In an event action, the `$Args` variable contains objects that represent the event arguments of the event that is being processed. This variable is populated only within the Action block of an event registration command. The value of this variable can also be found in the `SourceArgs` property of the `PSEventArgs` object (`System.Management.Automation.PSEventArgs`) that `Get-Event` returns.

\$PSItem

```
Get-Process | ForEach-Object -Process {  
    $PSItem.Name  
}
```

Same as `$_`. Contains the current object in the pipeline object. You can use this variable in commands that perform an action on every object or on selected objects in a pipeline.

\$?

```
Get-Process -Name doesnotexist  
Write-Host -Object "Was the last operation successful? $?"
```

Contains the execution status of the last operation. It contains TRUE if the last operation succeeded and FALSE if it failed.

\$error

```
Get-Process -Name doesnotexist  
Write-Host -Object ('The last error that occurred was: {0}' -f $error[0].Exception.Message)
```

Contains an array of error objects that represent the most recent errors. The most recent error is the first error object in the array (\$Error[0]).

To prevent an error from being added to the \$Error array, use the ErrorAction common parameter with a value of Ignore. For more information, see [about_CommonParameters](http://go.microsoft.com/fwlink/?LinkID=113216) (<http://go.microsoft.com/fwlink/?LinkID=113216>).

Read Built-in variables online: <https://riptutorial.com/powershell/topic/8732/built-in-variables>

Chapter 12: Calculated Properties

Introduction

Calculated Properties in Powershell are custom derived(Calculated) properties. It lets the user to format a certain property in a way he want it to be. The calculation(expression) can be a quite possibly anything.

Examples

Display file size in KB - Calculated Properties

Let's consider the below snippet,

```
Get-ChildItem -Path C:\MyFolder | Select-Object Name, CreationTime, Length
```

It simply output the folder content with the selected properties. Something like,

```
Name                CreationTime        Length
----                -
AnotherFile.txt    1/26/2017  2:45:02 PM  546000
filetomove.txt     1/5/2017    2:36:01 PM   5
```

What if I want to display the file size in KB ? This is where calculated properties comes handy.

```
Get-ChildItem C:\MyFolder | Select-Object Name, @{Name="Size_In_KB";Expression={$_.Length / 1Kb}}
```

Which produces,

```
Name                Size_In_KB
----                -
AnotherFile.txt     533.203125
SecondFile.txt     1066.4111328125
```

The `Expression` is what holds the calculation for calculated property. And yes, it can be anything!

Read Calculated Properties online: <https://riptutorial.com/powershell/topic/8913/calculated-properties>

Chapter 13: Cmdlet Naming

Introduction

CmdLets should be named using a `<verb>-<noun>` naming scheme in order to improve discoverability.

Examples

Verbs

Verbs used to name CmdLets should be named from verbs from the list supplied by `Get-Verb`

Further details on how to use verbs can be found at [Approved Verbs for Windows PowerShell](#)

Nouns

Nouns should always be singular.

Be consistent with the nouns. For instance `Find-Package` needs a provider the noun is `PackageProvider` **not** `ProviderPackage`.

Read Cmdlet Naming online: <https://riptutorial.com/powershell/topic/8703/cmdlet-naming>

Chapter 14: Comment-based help

Introduction

PowerShell features a documentation mechanism called comment-based help. It allows documenting scripts and functions with code comments. Comment-based help is most of the time written in comment blocks containing multiple help keywords. Help keywords start with dots and identify help sections that will be displayed by running the `Get-Help` cmdlet.

Examples

Function comment-based help

```
<#  
  
.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
}

```

The above function documentation can be displayed by running `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>
```

Notice that the comment-based keywords starting with a . match the `Get-Help` result sections.

Script comment-based help

```
<#

.SYNOPSIS
    Reads a CSV file and filters it.

.DESCRPTION
```

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}
```

The above script documentation can be displayed by running `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>

```

Read Comment-based help online: <https://riptutorial.com/powershell/topic/9530/comment-based-help>

Chapter 15: Common parameters

Remarks

Common parameters can be used with any cmdlet (that means as soon as you mark your function as cmdlet [see `CmdletBinding()`], you get all of these parameters for free).

Here is the list of all common parameters (alias is in parenthesis after corresponding parameter):

```
-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

ErrorAction parameter

Possible values are `Continue` | `Ignore` | `Inquire` | `SilentlyContinue` | `Stop` | `Suspend`.

Value of this parameter will determine how the cmdlet will handle non-terminating errors (those generated from `Write-Error` for example; to learn more about error handling see [*topic not yet created*]).

Default value (if this parameter is omitted) is `Continue`.

-ErrorAction Continue

This option will produce an error message and will continue with execution.

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Continue ; Write-Host "Second command"
Write-Error "test" -ErrorAction Continue ; Write-Host "Second command" : te
+ CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorEx
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorException
Second command
```

-ErrorAction Ignore

This option will not produce any error message and will continue with execution. Also no errors will be added to `$Error` automatic variable.

This option was introduced 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 Inquire

This option will produce an error message and will prompt user to choose an action to take.

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
```

```
PS C:\> Write-Error "test" -ErrorAction Inquire ; Write-Host "Second command"
Confirm
test
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend [?] Help (default is) N
```

-ErrorAction SilentlyContinue

This option will not produce an error message and will continue with execution. All errors will be added to `$Error` automatic variable.

```
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

This option will produce an error message and will not continue with execution.

```
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], WriteErrorEx
+ FullyQualifiedErrorId : Microsoft.PowerShell.Commands.WriteErrorExcep
```

-ErrorAction Suspend

Only available in Powershell Workflows. When used, if the command runs into an error, the workflow is suspended. This allows investigation of such error and gives a possibility to resume the workflow. To learn more about Workflow system, see [topic not yet created].

Read Common parameters online: <https://riptutorial.com/powershell/topic/5951/common-parameters>

Chapter 16: Communicating with RESTful APIs

Introduction

REST stands for Representational State Transfer (sometimes spelled "ReST"). It relies on a stateless, client-server, cacheable communications protocol and mostly HTTP protocol is used. It is primarily used to build Web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service and the APIs which are being used for it are RESTful APIs. In PowerShell, `Invoke-RestMethod` is used to deal with them.

Examples

Use Slack.com Incoming Webhooks

Define your payload to send for possible more complex data

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

Use `ConvertTo-Json` cmdlet and `Invoke-RestMethod` to execute the call

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

Post Message to 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
```

Using REST with PowerShell Objects to Get and Put individual data

GET your REST data and store in a PowerShell object:

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


Modify your data:

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

PUT the REST data back

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

Using REST with PowerShell Objects to GET and POST many items

GET your REST data and store in a PowerShell object:

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

Modify many items in your data:

```
$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 all of the REST data back:

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

Using REST with PowerShell to Delete items

Identify the item that is to be deleted and delete it:

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

Read [Communicating with RESTful APIs](https://riptutorial.com/powershell/topic/3869/communicating-with-restful-apis) online:

<https://riptutorial.com/powershell/topic/3869/communicating-with-restful-apis>

Chapter 17: Conditional logic

Syntax

- `if(expression){}`
- `if(expression){}else{}`
- `if(expression){}elseif(expression){}`
- `if(expression){}elseif(expression){}else{}`

Remarks

See also [Comparison Operators](#), which can be used in conditional expressions.

Examples

if, else and else if

Powershell supports standard conditional logic operators, much like many programming languages. These allow certain functions or commands to be run under particular circumstances.

With an `if` the commands inside the brackets (`{}`) are only executed if the conditions inside the `if()` are met

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

You can also do an `else`. Here the `else` commands are executed if the `if` conditions are **not** met:

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

or an `elseif`. An `else if` runs the commands if the `if` conditions are not met and the `elseif` conditions are met:

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

Note the above use `-eq`(equality) CmdLet and not `=` or `==` as many other languages do for equality.

Negation

You may want to negate a boolean value, i.e. enter an `if` statement when a condition is false rather than true. This can be done by using the `-Not` CmdLet

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

You can also use `!:`

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

there is also the `-ne` (not equal) operator:

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

If conditional shorthand

If you want to use the shorthand you can make use of conditional logic with the following shorthand. Only the string 'false' will evaluate to 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."  
}
```

Read Conditional logic online: <https://riptutorial.com/powershell/topic/7208/conditional-logic>

Chapter 18: Creating DSC Class-Based Resources

Introduction

Starting with PowerShell version 5.0, you can use PowerShell class definitions to create Desired State Configuration (DSC) Resources.

To aid in building DSC Resource, there's a `[DscResource()]` attribute that's applied to the class definition, and a `[DscProperty()]` resource to designate properties as configurable by the DSC Resource user.

Remarks

A class-based DSC Resource must:

- Be decorated with the `[DscResource()]` attribute
- Define a `Test()` method that returns `[bool]`
- Define a `Get()` method that returns its own object type (eg. `[Ticket]`)
- Define a `Set()` method that returns `[void]`
- At least one `Key` DSC Property

After creating a class-based PowerShell DSC Resource, it must be "exported" from a module, using a module manifest (`.psd1`) file. Within the module manifest, the `DscResourcesToExport` hashtable key is used to declare an array of DSC Resources (class names) to "export" from the module. This enables consumers of the DSC module to "see" the class-based resources inside the module.

Examples

Create a DSC Resource Skeleton Class

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

This example demonstrates how to build the outer section of a PowerShell class, that declares a DSC Resource. You still need to fill in the contents of the class definition.

DSC Resource Skeleton with Key Property

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

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

A DSC Resource must declare at least one key property. The key property is what uniquely identifies the resource from other resources. For example, let's say that you're building a DSC Resource that represents a ticket in a ticketing system. Each ticket would be uniquely represented with a ticket ID.

Each property that will be exposed to the *user* of the DSC Resource must be decorated with the `[DscProperty()]` attribute. This attribute accepts a `key` parameter, to indicate that the property is a key attribute for the DSC Resource.

DSC Resource with Mandatory Property

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

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

When building a DSC Resource, you'll often find that not every single property should be mandatory. However, there are some core properties that you'll want to ensure are configured by the user of the DSC Resource. You use the `Mandatory` parameter of the `[DscResource()]` attribute to declare a property as required by the DSC Resource's user.

In the example above, we've added a `Subject` property to a `Ticket` resource, that represents a unique ticket in a ticketing system, and designated it as a `Mandatory` property.

DSC Resource with Required Methods

```
[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
}
}
```

This is a complete DSC Resource that demonstrates all of the core requirements to build a valid resource. The method implementations are not complete, but are provided with the intention of showing the basic structure.

Read [Creating DSC Class-Based Resources](https://riptutorial.com/powershell/topic/8733/creating-dsc-class-based-resources) online:

<https://riptutorial.com/powershell/topic/8733/creating-dsc-class-based-resources>

Chapter 19: CSV parsing

Examples

Basic usage of Import-Csv

Given the following CSV-file

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

One can import the CSV rows in PowerShell objects using the `Import-Csv` command

```
> $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
```

Import from CSV and cast properties to correct type

By default, `Import-Csv` imports all values as strings, so to get `DateTime`- and integer-objects, we need to cast or parse them.

Using `Foreach-Object`:

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

    #Output object
    $_
}
```

Using calculated properties:

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


Output:

```
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
```

Read CSV parsing online: <https://riptutorial.com/powershell/topic/5691/csv-parsing>

Chapter 20: Desired State Configuration

Examples

Simple example - Enabling WindowsFeature

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

If you run this configuration in Powershell (EnableIISFeature), it will produce a localhost.mof file. This is the "compiled" configuration you can run on a machine.

To test the DSC configuration on your localhost, you can simply invoke the following:

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

Starting DSC (mof) on remote machine

Starting a DSC on a remote machine is almost just as simple. Assuming you've already set up Powershell remoting (or enabled WSMAN).

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

Nb: Assuming you have compiled a configuration for your node on your localmachine (and that the file myserver.somedomain.com.mof is present prior to starting the configuration)

Importing psd1 (data file) into local variable

Sometimes it can be useful to test your Powershell data files and iterate through the nodes and servers.

Powershell 5 (WMF5) added this neat little feature for doing this called Import-PowerShellDataFile

Example:

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

```
$data.AllNodes
```

List available DSC Resources

To list available DSC resources on your authoring node:

```
Get-DscResource
```

This will list all resources for all installed modules (that are in your PSModulePath) on your authoring node.

To list all available DSC resources that can be found in the online sources (PSGallery ++) on WMF 5 :

```
Find-DSCResource
```

Importing resources for use in DSC

Before you can use a resource in a configuration, you must explicitly import it. Just having it installed on your computer, will not let you use the resource implicitly.

Import a resource by using `Import-DscResource` .

Example showing how to import the `PSDesiredStateConfiguration` resource and the `File` resource.

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

    Import-DscResource PSDesiredStateConfiguration

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

Note: In order for DSC Resources to work, you must have the modules installed on the target machines when running the configuration. If you don't have them installed, the configuration will fail.

Read Desired State Configuration online: <https://riptutorial.com/powershell/topic/5662/desired-state-configuration>

Chapter 21: Embedding Managed Code (C# | VB)

Introduction

This topic is to briefly describe how C# or VB .NET Managed code can be scripted and utilised within a PowerShell script. This topic is not exploring all facets of the Add-Type cmdlet.

For more information on the Add-Type cmdlet, please refer to the MSDN documentation (for 5.1) here: <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.utility/add-type>

Parameters

| Parameter | Details |
|------------------------------|--|
| - TypeDefinition<String_> | Accepts the code as a string |
| -Language<String_> | Specifies the Managed Code language. Accepted values: CSharp, CSharpVersion3, CSharpVersion2, VisualBasic, JScript |

Remarks

Removing Added types

In later versions of PowerShell, Remove-TypeData has been added to the PowerShell cmdlet libraries which can allow for removal of a type within a session. For more details on this cmdlet, go here: <https://msdn.microsoft.com/en-us/powershell/reference/4.0/microsoft.powershell.utility/remove-typedata>

CSharp and .NET syntax

For those experience with .NET it goes without saying that the differing versions of C# can be quite radically different in their level of support for certain syntax.

If utilising Powershell 1.0 and/or -Language CSharp, the managed code will be utilising .NET 2.0 which is lacking in a number of features which C# developers typically use without a second thought these days, such as Generics, Linq and Lambda. On top of this is formal polymorphism, which is handled with defaulted parameters in later versions of C#/.NET.

Examples

C# Example

This example shows how to embed some basic C# into a PowerShell script, add it to the runspace/session and utilise the code within PowerShell syntax.

```
$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();
```

VB.NET Example

This example shows how to embed some basic C# into a PowerShell script, add it to the runspace/session and utilise the code within PowerShell syntax.

```
$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 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();
```

[Read Embedding Managed Code \(C# | VB\) online:](https://riptutorial.com/powershell/topic/9823/embedding-managed-code-csharp-vb)

[https://riptutorial.com/powershell/topic/9823/embedding-managed-code-csharp-vb-](https://riptutorial.com/powershell/topic/9823/embedding-managed-code-csharp-vb)

Chapter 22: Enforcing script prerequisites

Syntax

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

Remarks

#requires statement can be placed on any line in the script (it doesn't have to be the first line) but it must be the first statement on that line.

Multiple #requires statements may be used in one script.

For more reference, please refer to official documentation on Technet - [about_about_Requires](#).

Examples

Enforce minimum version of powershell host

```
#requires -version 4
```

After trying to run this script in lower version, you will see this error message

```
.\script.ps1 : The script 'script.ps1' cannot be run because it contained a "#requires" statement at line 1 for Windows PowerShell version 5.0. The version required by the script does not match the currently running version of Windows PowerShell version 2.0.
```

Enforce running the script as administrator

4.0

```
#requires -RunAsAdministrator
```

After trying to run this script without admin privileges, you will see this error message

```
.\script.ps1 : The script 'script.ps1' cannot be run because it contains a "#requires" statement for running as Administrator. The current Windows PowerShell session is not running as Administrator. Start Windows PowerShell by using the Run as Administrator option, and then try running the script again.
```

Read Enforcing script prerequisites online: <https://riptutorial.com/powershell/topic/5637/enforcing-script-prerequisites>

Chapter 23: Environment Variables

Examples

Windows environment variables are visible as a PS drive called Env:

You can see list with all environment variables with:
Get-Childitem env:

Instant call of Environment Variables with \$env:

```
$env:COMPUTERNAME
```

Read Environment Variables online: <https://riptutorial.com/powershell/topic/5635/environment-variables>

Chapter 24: Error handling

Introduction

This topic discuss about Error Types & Error Handling in PowerShell.

Examples

Error Types

An error is an error, one might wonder how could there be types in it. Well, with powershell the error broadly falls into two criteria,

- Terminating error
- Non-Terminating error

As the name says Terminating errors will terminate the execution and a Non-Terminating Errors let the execution continue to next statement.

This is true assuming that **\$ErrorActionPreference** value is default (Continue).
\$ErrorActionPreference is a [Preference Variable](#) which tells powershell what to do in case of an "Non-Terminating" error.

Terminating error

A terminating error can be handled with a typical try catch, as below

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

The above snippet will execute and the error will be caught thru the catch block.

Non-Terminating Error

A Non-Terminating error in the other hand will not be caught in the catch block by default. The reason behind that is a Non-Terminating error is not considered a critical error.

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

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

If you execute the above the line you wont get the output from catch block as since the error is not considered critical and the execution will simply continue from next command. However, the error will be displayed in the console. To handle a Non-Terminating error, you simple have to change the error preference.

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

Now, with the updated Error preference, this error will be considered a Terminating error and will be caught in the catch block.

Invoking Terminating & Non-Terminating Errors:

Write-Error cmdlet simply writes the error to the invoking host program. It doesn't stop the execution. Where as **throw** will give you a terminating error and stop the execution

```
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"
```

Read Error handling online: <https://riptutorial.com/powershell/topic/8075/error-handling>

Chapter 25: GUI in Powershell

Examples

WPF GUI for Get-Service cmdlet

```
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
```

This creates a dialog window which allows the user to select a computer name, then will display a table of services and their statuses on that computer.

This example uses WPF rather than Windows Forms.

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

Chapter 26: Handling Secrets and Credentials

Introduction

In Powershell, to avoid storing the password in *clear text* we use different methods of encryption and store it as secure string. When you are not specifying a key or securekey, this will only work for the same user on the same computer will be able to decrypt the encrypted string if you're not using Keys/SecureKeys. Any process that runs under that same user account will be able to decrypt that encrypted string on that same machine.

Examples

Prompting for Credentials

To prompt for credentials, you should almost always use the `Get-Credential` cmdlet:

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

Pre-filled user name:

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

Add a custom prompt message:

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

Accessing the Plaintext Password

The password in a credential object is an encrypted `[SecureString]`. The most straightforward way is to get a `[NetworkCredential]` which does not store the password encrypted:

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

The helper method (`.GetNetworkCredential()`) only exists on `[PSCredential]` objects.

To directly deal with a `[SecureString]`, use .NET methods:

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

Working with Stored Credentials

To store and retrieve encrypted credentials easily, use PowerShell's built-in XML serialization (Clixml):

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

To re-import:

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

The important thing to remember is that by default this uses the Windows data protection API, and the key used to encrypt the password is specific to both the *user and the machine* that the code is running under.

As a result, the encrypted credential cannot be imported by a different user nor the same user on a different computer.

By encrypting several versions of the same credential with different running users and on different computers, you can have the same secret available to multiple users.

By putting the user and computer name in the file name, you can store all of the encrypted secrets in a way that allows for the same code to use them without hard coding anything:

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"
```

The code that uses the stored credentials:

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

The correct version of the file for the running user will be loaded automatically (or it will fail because the file doesn't exist).

Storing the credentials in Encrypted form and Passing it as parameter when Required

```
$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.
```

Read Handling Secrets and Credentials online:

<https://riptutorial.com/powershell/topic/2917/handling-secrets-and-credentials>

Chapter 27: HashTables

Introduction

A Hash Table is a structure which maps keys to values. See [Hash Table](#) for details.

Remarks

An important concept which relies on Hash Tables is [Splatting](#). It is very useful for making a large number of calls with repetitive parameters.

Examples

Creating a Hash Table

Example of creating an empty HashTable:

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

Example of creating a HashTable with data:

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

Access a hash table value by key.

An example of defining a hash table and accessing a value by the key

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

An example of accessing a key with invalid characters for a property name:

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

```
Value3
```

Looping over a hash table

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

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

#Output
Key1 : Value1
Key2 : Value2
```

Add a key value pair to an existing hash table

An example, to add a "Key2" key with a value of "Value2" to the hash table, using the addition operator:

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

#Output

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

An example, to add a "Key2" key with a value of "Value2" to the hash table using the Add method:

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

#Output

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

Enumerating through keys and Key-Value Pairs

Enumerating through Keys

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

Enumerating through Key-Value Pairs

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

Remove a key value pair from an existing hash table

An example, to remove a "Key2" key with a value of "Value2" from the hash table, using the remove operator:

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

#Output

| Name | Value |
|-------|--------|
| ----- | ----- |
| Key1 | Value1 |

Read HashTables online: <https://riptutorial.com/powershell/topic/8083/hashtables>

Chapter 28: How to download latest artifact from Artifactory using Powershell script (v2.0 or below)?

Introduction

This documentation explains and provides steps to download latest artifact from a JFrog Artifactory repository using Powershell Script (v2.0 or below).

Examples

Powershell Script for downloading the latest artifact

```
$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)
```

Read [How to download latest artifact from Artifactory using Powershell script \(v2.0 or below\)?](https://riptutorial.com/powershell/topic/8883/how-to-download-latest-artifact-from-artifactory-using-powershell-script--v2-0-or-below--) online: <https://riptutorial.com/powershell/topic/8883/how-to-download-latest-artifact-from-artifactory-using-powershell-script--v2-0-or-below-->

Chapter 29: Infrastructure Automation

Introduction

Automating Infrastructure Management Services results in reducing the FTE as well as cumulatively getting better ROI using multiple tools, orchestrators, orchestration Engine , scripts and easy UI

Examples

Simple script for black-box integration test of console applications

This is a simple example on how you can automate tests for a console application that interact with standard input and standard output.

The tested application read and sum every new line and will provide the result after a single white line is provided. The power shell script write "pass" when the output match.

```
$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()
}
```

Read Infrastructure Automation online:

<https://riptutorial.com/powershell/topic/10909/infrastructure-automation>

Chapter 30: Introduction to Pester

Remarks

Pester is a test framework for PowerShell that allows you to run test cases for your PowerShell code. It can be used to run ex. unit tests to help you verify that your modules, scripts etc. work as intended.

[What is Pester and Why Should I Care?](#)

Examples

Getting Started with Pester

To get started with unit testing PowerShell code using the Pester-module, you need to be familiar with three keywords/commands:

- **Describe:** Defines a group of tests. All Pester test files need at least one Describe-block.
- **It:** Defines an individual test. You can have multiple It-blocks inside a Describe-block.
- **Should:** The verify/test command. It is used to define the result that should be considered a successful test.

Sample:

```
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
    }
}
```

Output:

Describing Validate Add-Numbers

[+] Should add 2 + 2 to equal 4 33ms

[+] Should handle strings 19ms

[+] Should return an integer 23ms

Read Introduction to Pester online: <https://riptutorial.com/powershell/topic/5753/introduction-to-pester>

Chapter 31: Introduction to Psake

Syntax

- Task - main function to execute a step of your build script
- Depends - property that specify what the current step depends upon
- default - there must always be a default task that will get executed if no initial task is specified
- FormatTaskName - specifies how each step is displayed in the result window.

Remarks

[psake](#) is a build automation tool written in PowerShell, and is inspired by Rake (Ruby make) and Bake (Boo make). It is used to create builds using dependency pattern. Documentation available [here](#)

Examples

Basic outline

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

Task Build {
    "Build"
}

Task Clean {
    "Clean"
}

Task default -Depends Build
```

FormatTaskName example

```
# 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
```

Run Task conditionally

```
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
```

Read Introduction to Psake online: <https://riptutorial.com/powershell/topic/5019/introduction-to-psake>

Chapter 32: ISE module

Introduction

Windows PowerShell Integrated Scripting Environment (ISE) is a host application that enables you to write, run, and test scripts and modules in a graphical and intuitive environment. Key features in Windows PowerShell ISE include syntax-coloring, tab completion, Intellisense, visual debugging, Unicode compliance, and context-sensitive Help, and provide a rich scripting experience.

Examples

Test Scripts

The simple, yet powerful use of the ISE is e.g. writing code in the top section (with intuitive syntax coloring) and run the code by simply marking it and hitting the F8 key.

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

1..10 | Get-Sum

#output
55
```

Read ISE module online: <https://riptutorial.com/powershell/topic/10954/ise-module>

Chapter 33: Loops

Introduction

A loop is a sequence of instruction(s) that is continually repeated until a certain condition is reached. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming. A loop lets you write a very simple statement to produce a significantly greater result simply by repetition. If the condition has been reached, the next instruction "falls through" to the next sequential instruction or branches outside the loop.

Syntax

- `for (<Initialization>; <Condition>; <Repetition>) { <Script_Block> }`
- `<Collection> | Foreach-Object { <Script_Block_with_$$as_current_item> }`
- `foreach (<Item> in <Collection>) { <Script_Block> }`
- `while (<Condition>){ <Script_Block> }`
- `do { <Script_Block> } while (<Condition>)`
- `do { <Script_Block> } until (<Condition>)`
- `<Collection>.foreach({ <Script_Block_with_$$as_current_item> })`

Remarks

Foreach

There are multiple ways to run a foreach-loop in PowerShell and they all bring their own advantages and disadvantages:

| Solution | Advantages | Disadvantages |
|-------------------|---|---|
| Foreach statement | Fastest. Works best with static collections (stored in a variable). | No pipeline input or output |
| ForEach() Method | Same scriptblock syntax as <code>Foreach-Object</code> , but faster. Works best with static collections (stored in a variable). Supports pipeline output. | No support for pipeline input. Requires PowerShell 4.0 or greater |
| Foreach-Object | Supports pipeline input and output. Supports begin and end-scriptblocks for initialization | Slowest |

| Solution | Advantages | Disadvantages |
|----------|---|---------------|
| (cmdlet) | and closing of connections etc. Most flexible solution. | |

Performance

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

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

Example output:

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

While `ForEach-Object` is the slowest, its pipeline-support might be useful as it lets you process items as they arrive (while reading a file, receiving data etc.). This can be very useful when working with big data and low memory as you don't need to load all the data to memory before processing.

Examples

For

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

A typical use of the for loop is to operate on a subset of the values in an array. In most cases, if you want to iterate all values in an array, consider using a foreach statement.

ForEach

`ForEach` has two different meanings in PowerShell. One is a [keyword](#) and the other is an alias for the [ForEach-Object](#) cmdlet. The former is described here.

This example demonstrates printing all items in an array to the console host:

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

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

This example demonstrates capturing the output of a ForEach loop:

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

Like the last example, this example, instead, demonstrates creating an array prior to storing the loop:

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

While

A while loop will evaluate a condition and if true will perform an action. As long as the condition evaluates to true the action will continue to be performed.

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

The following example creates a loop that will count down from 10 to 0

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

Unlike the [Do-While](#) loop the condition is evaluated prior to the action's first execution. The action will not be performed if the initial condition evaluates to false.

Note: When evaluating the condition, PowerShell will treat the existence of a return object as true. This can be used in several ways but below is an example to monitor for a process. This example will spawn a notepad process and then sleep the current shell as long as that process is running. When you manually close the notepad instance the while condition will fail and the loop will break.

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

ForEach-Object

The `ForEach-Object` cmdlet works similarly to the `foreach` statement, but takes its input from the pipeline.

Basic usage

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

Example:

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

`ForEach-Object` has two default aliases, `foreach` and `%` (shorthand syntax). Most common is `%` because `foreach` can be confused with the [foreach statement](#). Examples:

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

Advanced usage

`ForEach-Object` stands out from the alternative `foreach` solutions because it's a cmdlet which means it's designed to use the pipeline. Because of this, it has support for three scriptblocks just like a cmdlet or advanced function:

- **Begin:** Executed once before looping through the items that arrive from the pipeline. Usually used to create functions for use in the loop, creating variables, opening connections (database, web +) etc.
- **Process:** Executed once per item arrived from the pipeline. "Normal" foreach codeblock. This is the default used in the examples above when the parameter isn't specified.
- **End:** Executed once after processing all items. Usually used to close connections, generate a report etc.

Example:

```
"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  
}
```

Do

Do-loops are useful when you always want to run a codeblock at least once. A Do-loop will evaluate the condition after executing the codeblock, unlike a while-loop which does it before executing the codeblock.

You can use do-loops in two ways:

- Loop *while* the condition is true:

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

- Loop *until* the condition is true, in other words, loop while the condition is false:

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

Real Examples:

```
$i = 0

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

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

Do-While and Do-Until are antonymous loops. If the code inside the same, the condition will be reversed. The example above illustrates this behaviour.

ForEach() Method

4.0

Instead of the `ForEach-Object` cmdlet, there is also the possibility to use a `ForEach` method directly on object arrays like so

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

or - if desired - the parentheses around the script block can be omitted

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

Both will result in the output below

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

Continue

The `Continue` operator works in `For`, `ForEach`, `While` and `Do` loops. It skips the current iteration of the loop, jumping to the top of the innermost loop.

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

The above will output 1 to 20 to the console but miss out the number 7.

Note: When using a pipeline loop you should use `return` instead of `Continue`.

Break

The `break` operator will exit a program loop immediately. It can be used in `For`, `ForEach`, `While` and `Do` loops or in a `Switch Statement`.

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

The above will count to 15 but stop as soon as 7 is reached.

Note: When using a pipeline loop, `break` will behave as `continue`. To simulate `break` in the pipeline loop you need to incorporate some additional logic, cmdlet, etc. It is easier to stick with non-pipeline loops if you need to use `break`.

Break Labels

Break can also call a label that was placed in front of the instantiation of a loop:

```
$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++
}
```

Note: This code will increment `$i` to 8 and `$j` to 13 which will cause `$k` to equal 104. Since `$k` exceed 100, the code will then break out of both loops.

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

Chapter 34: Modules, Scripts and Functions

Introduction

PowerShell modules bring extendibility to the systems administrator, DBA, and developer. Whether it's simply as a method to share functions and scripts.

PowerShell Functions are to avoid repetitive codes. Refer [PS Functions][1] [1]: [PowerShell Functions](#)

PowerShell Scripts are used for automating administrative tasks which consists of command-line shell and associated cmdlets built on top of .NET Framework.

Examples

Function

A function is a named block of code which is used to define reusable code that should be easy to use. It is usually included inside a script to help reuse code (to avoid duplicate code) or distributed as part of a module to make it useful for others in multiple scripts.

Scenarios where a function might be useful:

- Calculate the average of a group of numbers
- Generate a report for running processes
- Write a function that tests is a computer is "healthy" by pinging the computer and accessing the `c$-share`

Functions are created using the `function` keyword, followed by a single-word name and a script block containing the code to executed when the function name is called.

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

Demo

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

Usage:

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

Script

A script is a text file with the file extension `.ps1` that contains PowerShell commands that will be executed when the script is called. Because scripts are saved files, they are easy to transfer between computers.

Scripts are often written to solve a specific problem, ex.:

- Run a weekly maintenance task
- To install and configure a solution/application on a computer

Demo

MyFirstScript.ps1:

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

You can run a script by entering the path to the file using an:

- Absolute path, ex. `c:\MyFirstScript.ps1`
- Relative path, ex. `.\MyFirstScript.ps1` if the current directory of your PowerShell console was `C:\`

Usage:

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

A script can also import modules, define it's own functions etc.

MySecondScript.ps1:

```
function HelloWorld {  
    Write-Host "Greetings from PowerShell!"  
}  
  
HelloWorld  
Write-Host "Let's get started!"  
2+2  
HelloWorld
```

Usage:

```
> .\MySecondScript.ps1  
Greetings from PowerShell!  
Let's get started!  
4  
Greetings from PowerShell!
```

Module

A module is a collection of related reusable functions (or cmdlets) that can easily be distributed to other PowerShell users and used in multiple scripts or directly in the console. A module is usually saved in its own directory and consists of:

- One or more code files with the `.psm1` file extension containing functions or binary assemblies (`.dll`) containing cmdlets
- A module manifest `.psd1` describing the module's name, version, author, description, which functions/cmdlets it provides etc.
- Other requirements for it to work incl. dependencies, scripts etc.

Examples of modules:

- A module containing functions/cmdlets that perform statistics on a dataset
- A module for querying and configuring databases

To make it easy for PowerShell to find and import a module, it is often placed in one of the known PowerShell module-locations defined in `$env:PSModulePath`.

Demo

List modules that are installed to one of the known module-locations:

```
Get-Module -ListAvailable
```

Import a module, ex. `Hyper-V` module:

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

List available commands in a module, ex. the `Microsoft.PowerShell.Archive`-module

```
> 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 |

Advanced Functions

Advanced functions behave the in the same way as cmdlets. The PowerShell ISE includes two skeletons of advanced functions. Access these via the menu, edit, code snippets, or by `Ctrl+J`. (As of PS 3.0, later versions may differ)

Key things that advanced functions include are,

- built-in, customized help for the function, accessible via `Get-Help`

- can use [CmdletBinding()] which makes the function act like a cmdlet
- extensive parameter options

Simple version:

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

Complete version:

```
<#
.Synopsis
    Short description
.DESRIPTION
    Long description
.EXAMPLE
    Example of how to use this cmdlet
.EXAMPLE
    Another example of how to use this cmdlet
.INPUTS
    Inputs to this cmdlet (if any)
.OUTPUTS
    Output from this cmdlet (if any)
.NOTES
```

```

General notes
.COMPONENT
    The component this cmdlet belongs to
.ROLE
    The role this cmdlet belongs to
.FUNCTIONALITY
    The functionality that best describes this cmdlet
#>
function Verb-Noun
{
    [CmdletBinding(DefaultParameterSetName='Parameter Set 1',
        SupportsShouldProcess=$true,
        PositionalBinding=$false,
        HelpUri = 'http://www.microsoft.com/',
        ConfirmImpact='Medium')]
    [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
    {

```

```
}  
}
```

Read Modules, Scripts and Functions online:

<https://riptutorial.com/powershell/topic/5755/modules--scripts-and-functions>

Chapter 35: MongoDB

Remarks

The most hard part is to attach a **subdocument** into the document which hasn't created yet if we need the subdocument need to be in the expected looking we will need to iterate with a for loop the array into a variable and using `$doc2.add("Key", "Value")` instead using the `foreach` current array with index. This will make the subdocument in two lines as you can see in the "Tags" =

```
[MongoDB.Bson.BsonDocument] $doc2.
```

Examples

MongoDB with C# driver 1.7 using PowerShell

I need to query all the details from virtual machine and update into the 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"
}
```

I have 3 sets of 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);
    }
}

```

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

Chapter 36: Naming Conventions

Examples

Functions

```
Get-User ()
```

- Use *Verb-Noun* pattern while naming a function.
- Verb implies an action e.g. `Get`, `Set`, `New`, `Read`, `Write` and many more. See [approved verbs](#).
- Noun should be singular even if it acts on multiple items. `Get-User ()` may return one or multiple users.
- Use Pascal case for both Verb and Noun. E.g. `Get-UserLogin ()`

Read Naming Conventions online: <https://riptutorial.com/powershell/topic/9714/naming-conventions>

Chapter 37: Operators

Introduction

An operator is a character that represents an action. It tells the compiler/interpreter to perform specific mathematical, relational or logical operation and produce final result. PowerShell interprets in a specific way and categorizes accordingly like arithmetic operators perform operations primarily on numbers, but they also affect strings and other data types. Along with the basic operators, PowerShell has a number of operators that save time and coding effort(eg: -like,-match,-replace,etc).

Examples

Arithmetic Operators

```
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
```

Logical Operators

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

Assignment Operators

Simple arithmetic:

```
$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
```

Increment and decrement:

```
$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
```

Comparison Operators

PowerShell comparison operators are comprised of a leading dash (-) followed by a name (`eq` for equal, `gt` for greater than, etc...).

Names can be preceded by special characters to modify the behavior of the operator:

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

Case-Insensitive is the default if not specified, ("`a`" `-eq` "`A`") same as ("`a`" `-ieq` "`A`").

Simple comparison operators:

```
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 (<=)
```

String comparison operators:

```
"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
```

Collection comparison operators:

```
"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)
```

Redirection Operators

Success output stream:

```
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
```

Error output stream:

```
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
```

Warning output stream: (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
```

Verbose output stream: (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
```

Information output stream: (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
```

All output streams:

```
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
```

Differences to the pipe operator (|)

Redirection operators only redirect streams to files or streams to streams. The pipe operator pumps an object down the pipeline to a cmdlet or the output. How the pipeline works differs in general from how redirection works and can be read on [Working with the PowerShell pipeline](#)

Mixing operand types : the type of the left operand dictates the behavior.

For Addition

```
"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"
```

For Multiplication

```
"3" * 2          # Gives "33"
```

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

The impact may have hidden consequences on comparison operators :

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

String Manipulation Operators

Replace operator:

The `-replace` operator replaces a pattern in an input value using a regular expression. This operator uses two arguments (separated by a comma): a regular expression pattern and its replacement value (which is optional and an empty string by default).

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

Split and Join operators:

The `-split` operator splits a string into an array of sub-strings.

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

The `-join` operator joins an array of strings into a single string.

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

Read Operators online: <https://riptutorial.com/powershell/topic/1071/operators>

Chapter 38: Package management

Introduction

PowerShell Package Management allows you to find, install, update and uninstall PowerShell Modules and other packages.

[PowerShellGallery.com](https://www.powershellgallery.com) is the default source for PowerShell modules. You can also browse the site for available packages, command and preview the code.

Examples

Find a PowerShell module using a pattern

To find a module that ends with `DSC`

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

Create the default PowerShell Module Repository

If for some reason, the default PowerShell module repository `PSGallery` gets removed. You will need to create it. This is the command.

```
Register-PSRepository -Default
```

Find a module by name

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

Install a Module by name

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

Uninstall a module my name and version

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

Update a module by name

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

Read Package management online: <https://riptutorial.com/powershell/topic/8698/package->

management

Chapter 39: Parameter sets

Introduction

Parameter sets are used to limit the possible combination of parameters, or to enforce the use of parameters when 1 or more parameters are selected.

The examples will explain the use and reason of a parameter set.

Examples

Simple parameter sets

```
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 to enforce the use of a parmeter when a other is selected.

When you want for example enforce the use of the parameter Password if the parameter User is provided. (and vise 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

```

Parameter set to limit the combination of parameters

```

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

```

Read Parameter sets online: <https://riptutorial.com/powershell/topic/6598/parameter-sets>

Chapter 40: PowerShell "Streams"; Debug, Verbose, Warning, Error, Output and Information

Remarks

<https://technet.microsoft.com/en-us/library/hh849921.aspx>

Examples

Write-Output

`Write-Output` generates output. This output can go to the next command after the pipeline or to the console so it's simply displayed.

The Cmdlet sends objects down the primary pipeline, also known as the "output stream" or the "success pipeline." To send error objects down the error pipeline, use `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. The `Write-Output` cmdlet sends the specified object down the pipeline to the next command.
2. If the command is the last command in the pipeline, the object is displayed in the console.
3. The PowerShell interpreter treats this as an implicit `Write-Output`.

Because `Write-Output`'s default behavior is to display the objects at the end of a pipeline, it is generally not necessary to use the Cmdlet. For example, `Get-Process | Write-Output` is equivalent to `Get-Process`.

Write Preferences

Messages can be written with;

```
Write-Verbose "Detailed Message"
Write-Information "Information Message"
Write-Debug "Debug Message"
```

```
Write-Progress "Progress Message"  
Write-Warning "Warning Message"
```

Each of these has a preference variable;

```
$VerbosePreference = "SilentlyContinue"  
$InformationPreference = "SilentlyContinue"  
$DebugPreference = "SilentlyContinue"  
$ProgressPreference = "Continue"  
$WarningPreference = "Continue"
```

The preference variable controls how the message and subsequent execution of the script are handled;

```
$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"
```

The color of the messages can be controlled for `Write-Error` by setting;

```
$host.PrivateData.ErrorBackgroundColor = "Black"  
$host.PrivateData.ErrorForegroundColor = "Red"
```

Similar settings are available for `Write-Verbose`, `Write-Debug` and `Write-Warning`.

Read PowerShell "Streams"; Debug, Verbose, Warning, Error, Output and Information online:
<https://riptutorial.com/powershell/topic/3255/powershell--streams---debug--verbose--warning--error--output-and-information>

Chapter 41: PowerShell Background Jobs

Introduction

Jobs were introduced in PowerShell 2.0 and helped to solve a problem inherent in the command-line tools. In a nutshell, if you start a long running task, your prompt is unavailable until the task finishes. As an example of a long running task, think of this simple PowerShell command:

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

It will take a while to fetch full directory list of your C: drive. If you run it as Job then the console will get the control back and you can capture the result later on.

Remarks

PowerShell Jobs run in a new process. This has pros and cons which are related.

Pros:

1. The job runs in a clean process, including environment.
2. The job can run asynchronously to your main PowerShell process

Cons:

1. Process environment changes will not be present in the job.
2. Parameters pass to and returned results are serialized.
 - This means if you change a parameter object while the job is running it will not be reflected in the job.
 - This also means if an object cannot be serialized you cannot pass or return it (although PowerShell may Copy any parameters and pass/return a PSObject.)

Examples

Basic job creation

Start a Script Block as background job:

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

Start a script as background job:

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

Start a job using `Invoke-Command` on a remote machine:

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

Start job as a different user (Prompts for password):

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

Or

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

Start job as a different user (No 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
```

Basic job management

Get a list of all jobs in the current session:

```
Get-Job
```

Waiting on a job to finish before getting the result:

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

Timeout a job if it runs too long (10 seconds in this example)

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

Stopping a job (completes all tasks that are pending in that job queue before ending):

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

Remove job from current session's background jobs list:

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

Note: The following will only work on `Workflow Jobs`.

Suspend a `Workflow Job` (Pause):

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

Resume a Workflow Job:

\$job | Resume-Job

Read PowerShell Background Jobs online:

<https://riptutorial.com/powershell/topic/3970/powershell-background-jobs>

Chapter 42: PowerShell Classes

Introduction

A class is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods). A class is a blueprint for an object. It is used as a model to define the structure of objects. An object contains data that we access through properties and that we can work on using methods. PowerShell 5.0 added the ability to create your own classes.

Examples

Methods and properties

```
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!"
```

Listing available constructors for a class

5.0

In PowerShell 5.0+ you can list available constructors by calling the static `new`-method without parentheses.

```
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)
```

This is the same technique that you can use to list overload definitions for any method

```
> 'abc'.CompareTo

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

For earlier versions you can create your own function to list available constructors:

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

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

Usage:

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

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

```
endar calendar, System.DateTimeKind kind
```

Constructor overloading

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

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

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

Get All Members of an Instance

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

This will return all members of the type instance. Here is a part of a sample output for String instance

```
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()
...
```

Basic Class Template

```
# 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
}
}

```

Inheritance from Parent Class to Child Class

```

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

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

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

$Inherit = [ChildClass]::new()

```

SO, **\$Inherit.Message** will give you the

"Its under the Parent Class"

Read PowerShell Classes online: <https://riptutorial.com/powershell/topic/1146/powershell-classes>

Chapter 43: PowerShell Dynamic Parameters

Examples

"Simple" dynamic parameter

This example adds a new parameter to MyTestFunction if `$SomeUsefulNumber` is greater than 5.

```
function MyTestFunction
{
    [CmdletBinding(DefaultParameterSetName='DefaultConfiguration')]
    Param
    (
        [Parameter(Mandatory=$true)][int]$SomeUsefulNumber
    )

    DynamicParam
    {
        $paramDictionary = New-Object -Type
System.Management.Automation.RuntimeDefinedParameterDictionary
        $attributes = New-Object System.Management.Automation.ParameterAttribute
        $attributes.ParameterSetName = "__AllParameterSets"
        $attributes.Mandatory = $true
        $attributeCollection = New-Object -Type
System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($attributes)
        # If "SomeUsefulNumber" is greater than 5, then add the "MandatoryParam1" parameter
        if($SomeUsefulNumber -gt 5)
        {
            # Create a mandatory string parameter called "MandatoryParam1"
            $dynParam1 = New-Object -Type
System.Management.Automation.RuntimeDefinedParameter("MandatoryParam1", [String],
$attributeCollection)
            # Add the new parameter to the dictionary
            $paramDictionary.Add("MandatoryParam1", $dynParam1)
        }
        return $paramDictionary
    }

    process
    {
        Write-Host "SomeUsefulNumber = $SomeUsefulNumber"
        # Notice that dynamic parameters need a specific syntax
        Write-Host ("MandatoryParam1 = {0}" -f $PSBoundParameters.MandatoryParam1)
    }
}
```

Usage:

```
PS > MyTestFunction -SomeUsefulNumber 3
SomeUsefulNumber = 3
MandatoryParam1 =

PS > MyTestFunction -SomeUsefulNumber 6
```

```
cmdlet MyTestFunction at command pipeline position 1
Supply values for the following parameters:
MandatoryParam1:

PS >MyTestFunction -SomeUsefulNumber 6 -MandatoryParam1 test
SomeUsefulNumber = 6
MandatoryParam1 = test
```

In the second usage example, you can clearly see that a parameter is missing.

Dynamic parameters are also taken into account with auto completion.

Here's what happens if you hit ctrl + space at the end of the line:

```
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
```

Read PowerShell Dynamic Parameters online:

<https://riptutorial.com/powershell/topic/6704/powershell-dynamic-parameters>

Chapter 44: PowerShell Functions

Introduction

A function is basically a named block of code. When you call the function name, the script block within that function runs. It is a list of PowerShell statements that has a name that you assign. When you run a function, you type the function name. It is a method of saving time when tackling repetitive tasks. PowerShell formats in three parts: the keyword 'Function', followed by a Name, finally, the payload containing the script block, which is enclosed by curly/parenthesis style bracket.

Examples

Simple Function with No Parameters

This is an example of a function which returns a string. In the example, the function is called in a statement assigning a value to a variable. The value in this case is the return value of the function.

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

# Invoking the function
$greeting = Get-Greeting

# demonstrate output
$greeting
Get-Greeting
```

`function` declares the following code to be a function.

`Get-Greeting` is the name of the function. Any time that function needs to be used in the script, the function can be called by means of invoking it by name.

`{ ... }` is the script block that is executed by the function.

If the above code is executed in the ISE, the results would be something like:

```
Hello World
Hello World
```

Basic Parameters

A function can be defined with parameters using the `param` block:

```
function Write-Greeting {
    param(
        [Parameter(Mandatory,Position=0)]
```

```

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

```

Or using the simple function syntax:

```

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

```

Note: Casting parameters is not required in either type of parameter definition.

Simple function syntax (SFS) has very limited capabilities in comparison to the param block. Though you can define parameters to be exposed within the function, you cannot specify [Parameter Attributes](#), utilize [Parameter Validation](#), include `[CmdletBinding()]`, with SFS (and this is a non-exhaustive list).

Functions can be invoked with ordered or named parameters.

The order of the parameters on the invocation is matched to the order of the declaration in the function header (by default), or can be specified using the `Position` Parameter Attribute (as shown in the advanced function example, above).

```

$greeting = Write-Greeting "Jim" 82

```

Alternatively, this function can be invoked with named parameters

```

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

```

Mandatory Parameters

Parameters to a function can be marked as mandatory

```

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

```

If the function is invoked without a value, the command line will prompt for the value:

```

$greeting = Get-Greeting

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

```

Advanced Function

This is a copy of the advanced function snippet from the Powershell ISE. Basically this is a template for many of the things you can use with advanced functions in Powershell. Key points to note:

- get-help integration - the beginning of the function contains a comment block that is set up to be read by the get-help cmdlet. The function block may be located at the end, if desired.
- cmdletbinding - function will behave like a cmdlet
- parameters
- parameter sets

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

    [Alias()]
    [OutputType([String])]
    Param
    (
        # Param1 help description
        [Parameter(Mandatory=$true,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true,
            ValueFromRemainingArguments=$false,
            Position=0,
            ParameterSetName='Parameter Set 1')]

        [ValidateNotNull()]
        [ValidateNotNullOrEmpty()]
        [ValidateCount(0,5)]
        [ValidateSet("sun", "moon", "earth")]
        [Alias("p1")]
    )
}
```



```

$Param1,

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

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

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

```

Parameter Validation

There are a variety of ways to validate parameter entry, in PowerShell.

Instead of writing code within functions or scripts to validate parameter values, these `ParameterAttributes` will throw if invalid values are passed.

ValidateSet

Sometimes we need to restrict the possible values that a parameter can accept. Say we want to allow only red, green and blue for the `$Color` parameter in a script or function.

We can use the `ValidateSet` parameter attribute to restrict this. It has the additional benefit of allowing tab completion when setting this argument (in some environments).

```

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

```

You can also specify `IgnoreCase` to disable case sensitivity.

ValidateRange

This method of parameter validation takes a min and max Int32 value, and requires the parameter to be within that range.

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

ValidatePattern

This method of parameter validation accepts parameters that match the regex pattern specified.

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

ValidateLength

This method of parameter validation tests the length of the passed string.

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

ValidateCount

This method of parameter validation tests the amount of arguments passed in, for example, an array of strings.

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

ValidateScript

Finally, the ValidateScript method is extraordinarily flexible, taking a scriptblock and evaluating it using `$_` to represent the passed argument. It then passes the argument if the result is `$true` (including any output as valid).

This can be used to test that a file exists:

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

To check that a user exists in AD:

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

And pretty much anything else you can write (as it's not restricted to oneliners):

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

Read PowerShell Functions online: <https://riptutorial.com/powershell/topic/1673/powershell-functions>

Chapter 45: Powershell Modules

Introduction

Starting with PowerShell version 2.0, developers can create PowerShell modules. PowerShell modules encapsulate a set of common functionality. For example, there are vendor-specific PowerShell modules that manage various cloud services. There are also generic PowerShell modules that interact with social media services, and perform common programming tasks, such as Base64 encoding, working with Named Pipes, and more.

Modules can expose command aliases, functions, variables, classes, and more.

Examples

Create a Module 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 = @()
}
```

Every good PowerShell module has a module manifest. The module manifest simply contains metadata about a PowerShell module, and doesn't define the actual contents of the module.

The manifest file is a PowerShell script file, with a `.psd1` file extension, that contains a `HashTable`. The `HashTable` in the manifest must contain specific keys, in order for PowerShell to correctly interpret it as a PowerShell module file.

The example above provides a list of the core `HashTable` keys that make up a module manifest, but there are many others. The `New-ModuleManifest` command helps you create a new module manifest skeleton.

Simple Module Example

```
function Add {
  [CmdletBinding()]
  param (
    [int] $x
  )
}
```

```
, [int] $y
)

return $x + $y
}

Export-ModuleMember -Function Add
```

This is a simple example of what a PowerShell script module file might look like. This file would be called `MyCoolModule.psm1`, and is referenced from the module manifest (`.psd1`) file. You'll notice that the `Export-ModuleMember` command enables us to specify which functions in the module we want to "export," or expose, to the user of the module. Some functions will be internal-only, and shouldn't be exposed, so those would be omitted from the call to `Export-ModuleMember`.

Exporting a Variable from a Module

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

To export a variable from a module, you use the `Export-ModuleMember` command, with the `-Variable` parameter. Remember, however, that if the variable is also not explicitly exported in the module manifest (`.psd1`) file, then the variable will not be visible to the module consumer. Think of the module manifest like a "gatekeeper." If a function or variable isn't allowed in the module manifest, it won't be visible to the module consumer.

Note: Exporting a variable is similar to making a field in a class public. It is not advisable. It would be better to expose a function to get the field and a function to set the field.

Structuring PowerShell Modules

Rather than defining all of your functions in a single `.psm1` PowerShell script module file, you might want to break apart your function into individual files. You can then dot-source these files from your script module file, which in essence, treats them as if they were part of the `.psm1` file itself.

Consider this module directory structure:

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

Inside your `MyCoolModule.psm1` file, you could insert the following code:

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

This would dot-source the individual function files into the `.psm1` module file.

Location of Modules

PowerShell looks for modules in the directories listed in the `$Env:PSModulePath`.

A module called *foo*, in a folder called *foo* will be found with `Import-Module foo`

In that folder, PowerShell will look for a module manifest (*foo.psd1*), a module file (*foo.psm1*), a DLL (*foo.dll*).

Module Member Visibility

By default, only functions defined in a module are visible outside of the module. In other words, if you define variables and aliases in a module, they won't be available except in the module's code.

To override this behavior, you can use the `Export-ModuleMember` cmdlet. It has parameters called `-Function`, `-Variable`, and `-Alias` which allow you to specify exactly which members are exported.

It is important to note that if you use `Export-ModuleMember`, **only** the items you specify will be visible.

Read Powershell Modules online: <https://riptutorial.com/powershell/topic/8734/powershell-modules>

Chapter 46: Powershell profiles

Remarks

Profile file is a powershell script that will run while the powershell console is starting. This way we can have our environment prepared for us each time we start new powershell session.

Typical things we want to do on powershell start are:

- importing modules we use often (ActiveDirectory, Exchange, some specific DLL)
- logging
- changing the prompt
- diagnostics

There are several profile files and locations that have different uses and also hierarchy of start-up order:

| Host | User | Path |
|---------|---------|---|
| All | All | %WINDIR%\System32\WindowsPowerShell\v1.0\profile.ps1 |
| All | Current | %USERPROFILE%\Documents\WindowsPowerShell\profile.ps1 |
| Console | All | %WINDIR%\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1 |
| Console | Current | %USERPROFILE%\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1 |
| ISE | All | %WINDIR%\System32\WindowsPowerShell\v1.0\Microsoft.PowerShellISE_profile.ps1 |
| ISE | Current | %USERPROFILE%\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1 |

Examples

Create an basic profile

A PowerShell profile is used to load user defined variables and functions automatically.

PowerShell profiles are not automatically created for users.

To create a PowerShell profile `C:>New-Item -ItemType File $profile.`

If you are in ISE you can use the built in editor `C:>psEdit $profile`

An easy way to get started with your personal profile for the current host is to save some text to path stored in the `$profile`-variable

```
"#Current host, current user" > $profile
```

Further modification to the profile can be done using PowerShell ISE, notepad, Visual Studio Code or any other editor.

The `$profile`-variable returns the current user profile for the current host by default, but you can access the path to the machine-policy (all users) and/or the profile for all hosts (console, ISE, 3rd party) by using it's properties.

```
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
```

Read Powershell profiles online: <https://riptutorial.com/powershell/topic/5636/powershell-profiles>

Chapter 47: Powershell Remoting

Remarks

- [about_Remote](#)
- [about_RemoteFAQ](#)
- [about_RemoteTroubleshooting](#)

Examples

Enabling PowerShell Remoting

PowerShell remoting must first be enabled on the server to which you wish to remotely connect.

```
Enable-PSRemoting -Force
```

This command does the following:

- Runs the Set-WSManQuickConfig cmdlet, which performs the following tasks:
- Starts the WinRM service.
- Sets the startup type on the WinRM service to Automatic.
- Creates a listener to accept requests on any IP address, if one does not already exist.
- Enables a firewall exception for WS-Management communications.
- Registers the Microsoft.PowerShell and Microsoft.PowerShell.Workflow session configurations, if they are not already registered.
- Registers the Microsoft.PowerShell32 session configuration on 64-bit computers, if it is not already registered.
- Enables all session configurations.
- Changes the security descriptor of all session configurations to allow remote access.
- Restarts the WinRM service to make the preceding changes effective.

Only for non-domain environments

For servers in an AD Domain the PS remoting authentication is done through Kerberos ('Default'), or NTLM ('Negotiate'). If you want to allow remoting to a non-domain server you have two options.

Either set up WSMan communication over HTTPS (which requires certificate generation) or enable basic authentication which sends your credentials across the wire base64-encoded (that's basically the same as plain-text so be careful with this).

In either case you'll have to add the remote systems to your WSMan trusted hosts list.

Enabling Basic Authentication

```
Set-Item WSMan:\localhost\Service\AllowUnencrypted $true
```

Then on the computer you wish to connect *from*, you must tell it to trust the computer you're connecting *to*.

```
Set-Item WSMan:\localhost\Client\TrustedHosts '192.168.1.1,192.168.1.2'
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *.contoso.com
```

```
Set-Item WSMan:\localhost\Client\TrustedHosts *
```

Important: You must tell your client to trust the computer addressed in the way you want to connect (e.g. if you connect via IP, it must trust the IP not the hostname)

Connecting to a Remote Server via PowerShell

Using credentials from your local computer:

```
Enter-PSSession 192.168.1.1
```

Prompting for credentials on the remote computer

```
Enter-PSSession 192.168.1.1 -Credential $(Get-Credential)
```

Run commands on a Remote Computer

Once Powershell remoting is enabled (Enable-PSRemoting) You can run commands on the remote computer like this:

```
Invoke-Command -ComputerName "RemoteComputerName" -ScriptBlock {  
    Write host "Remote Computer Name: $ENV:ComputerName"  
}
```

The above method creates a temporary session and closes it right after the command or scriptblock ends.

To leave the session open and run other command in it later, you need to create a remote session first:

```
$Session = New-PSSession -ComputerName "RemoteComputerName"
```

Then you can use this session each time you invoke commands on the remote computer:

```
Invoke-Command -Session $Session -ScriptBlock {  
    Write host "Remote Computer Name: $ENV:ComputerName"  
}
```

```
Invoke-Command -Session $Session -ScriptBlock {
```

```
Get-Date
}
```

If you need to use different Credentials, you can add them with the `-Credential` Parameter:

```
$Cred = Get-Credential
Invoke-Command -Session $Session -Credential $Cred -ScriptBlock {...}
```

Remoting serialization warning

Note:

It is important to know that remoting serializes PowerShell objects on the remote system and deserializes them on your end of the remoting session, i.e. they are converted to XML during transport and lose all of their methods.

```
$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...
```

Whereas you have the methods on the regular PS object:

```
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)
```

Argument Usage

To use arguments as parameters for the remote scripting block, one might either use the `ArgumentList` parameter of `Invoke-Command`, or use the `$Using:` syntax.

Using `ArgumentList` with unnamed parameters (i.e. in the order they are passed to the scriptblock):

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $args[0]
    Remove-Item -Path $args[1] -ErrorAction SilentlyContinue -Force
}
```

Using `ArgumentList` with named parameters:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ArgumentList $servicesToShow,$fileName -ScriptBlock {
    Param($serviceToShowInRemoteSession,$fileToDelete)

    Write-Host "Calling script block remotely with $($Args.Count)"
    Get-Service -Name $serviceToShowInRemoteSession
    Remove-Item -Path $fileToDelete -ErrorAction SilentlyContinue -Force
}
```

Using `$Using:` syntax:

```
$servicesToShow = "service1"
$fileName = "C:\temp\servicestatus.csv"
Invoke-Command -Session $session -ScriptBlock {
    Get-Service $Using:servicesToShow
    Remove-Item -Path $fileName -ErrorAction SilentlyContinue -Force
}
```

A best practise for automatically cleaning-up PSSessions

When a remote session is created via the `New-PSSession` cmdlet, the `PSSession` persists until the current PowerShell session ends. Meaning that, by default, the `PSSession` and all associated resources will continue to be used until the current PowerShell session ends.

Multiple active `PSSessions` can become a strain on resources, particularly for long running or interlinked scripts that create hundreds of `PSSessions` in a single PowerShell session.

It is best practise to explicitly remove each `PSSession` after it is finished being used. [1]

The following code template utilises `try-catch-finally` in order to achieve the above, combining error handling with a secure way to ensure all created `PSSessions` are removed when they are finished being used:

```
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
    }
}
```

References: [1] <https://msdn.microsoft.com/en-us/powershell/reference/5.1/microsoft.powershell.core/new-pssession>

Read Powershell Remoting online: <https://riptutorial.com/powershell/topic/3087/powershell-remoting>

Chapter 48: powershell sql queries

Introduction

By going through this document You can get to know how to use SQL queries with powershell

Parameters

| Item | Description |
|----------------------------|---|
| \$ServerInstance | Here we have to mention the instance in which the database is present |
| \$Database | Here we have to mention the database in which the table is present |
| \$Query | Here we have to the query which you we want to execute in SQ |
| \$Username & \$Password | UserName and Password which have access in database |

Remarks

You can use the below function if in case you are not able to import SQLPS module

```
function Import-Xls
{
    [CmdletBinding(SupportsShouldProcess=$true)]

    Param(
        [parameter(
            mandatory=$true,
            position=1,
            ValueFromPipeline=$true,
            ValueFromPipelineByPropertyName=$true)]
        [String[]]
        $Path,

        [parameter(mandatory=$false)]
        $Worksheet = 1,

        [parameter(mandatory=$false)]
        [switch]
        $Force
    )

    Begin
    {
        function GetTempFileName($extension)
```

```

{
    $temp = [io.path]::GetTempFileName();
    $params = @{
        Path = $temp;
        Destination = $temp + $extension;
        Confirm = $false;
        Verbose = $VerbosePreference;
    }
    Move-Item @params;
    $temp += $extension;
    return $temp;
}

# since an extension like .xls can have multiple formats, this
# will need to be changed
#
$xmlFileFormats = @{
    # single worksheet formats
    '.csv' = 6;          # 6, 22, 23, 24
    '.dbf' = 11;         # 7, 8, 11
    '.dif' = 9;          #
    '.prn' = 36;         #
    '.slk' = 2;          # 2, 10
    '.wk1' = 31;         # 5, 30, 31
    '.wk3' = 32;         # 15, 32
    '.wk4' = 38;         #
    '.wks' = 4;          #
    '.xlw' = 35;         #

    # multiple worksheet formats
    '.xls' = -4143;      # -4143, 1, 16, 18, 29, 33, 39, 43
    '.xlsb' = 50;        #
    '.xlsm' = 52;        #
    '.xlsx' = 51;        #
    '.xml' = 46;         #
    '.ods' = 60;         #
}

$xml = New-Object -ComObject Excel.Application;
$xml.DisplayAlerts = $false;
$xml.Visible = $false;
}

Process
{
    $Path | ForEach-Object {

        if ($Force -or $psCmdlet.ShouldProcess($_)) {

            $fileExist = Test-Path $_

            if (-not $fileExist) {
                Write-Error "Error: $_ does not exist" -Category ResourceUnavailable;
            } else {
                # create temporary .csv file from excel file and import .csv
                #
                $_ = (Resolve-Path $_).ToString();
                $wb = $xml.Workbooks.Add($_);
                if ($?) {
                    $csvTemp = GetTempFileName(".csv");
                }
            }
        }
    }
}

```



```
Invoke-Sqlcmd2 -Serverinstance $Inst -Database $DBName -query $Query -Username $UID -Password $Password
```

Read powershell sql queries online: <https://riptutorial.com/powershell/topic/8217/powershell-sql-queries>

Chapter 49: PowerShell Workflows

Introduction

PowerShell Workflow is a feature that was introduced starting with PowerShell version 3.0. Workflow definitions look very similar to PowerShell function definitions, however they execute within the Windows Workflow Foundation environment, instead of directly in the PowerShell engine.

Several unique "out of box" features are included with the Workflow engine, most notably, job persistence.

Remarks

The PowerShell Workflow feature is exclusively supported on the Microsoft Windows platform, under PowerShell Desktop Edition. PowerShell Core Edition, which is supported on Linux, Mac, and Windows, does not support the PowerShell Workflow feature.

When authoring a PowerShell Workflow, keep in mind that workflows call activities, not cmdlets. You can still call cmdlets from a PowerShell Workflow, but the Workflow Engine will implicitly wrap the cmdlet invocation in an `InlineScript` activity. You can also explicitly wrap code inside of the `InlineScript` activity, which executes PowerShell code; by default the `InlineScript` activity runs in a separate process, and returns the result to the calling Workflow.

Examples

Simple Workflow Example

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

This is a basic example of a PowerShell Workflow definition.

Workflow with Input Parameters

Just like PowerShell functions, workflows can accept input parameter. Input parameters can optionally be bound to a specific data type, such as a string, integer, etc. Use the standard `param` keyword to define a block of input parameters, directly after the workflow declaration.

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

```
DoSomeWork -ComputerName server01, server02, server03
```

Run Workflow as a Background Job

PowerShell Workflows are inherently equipped with the ability to run as a background job. To call a workflow as a PowerShell background job, use the `-AsJob` parameter when invoking the workflow.

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

DoSomeWork -AsJob
```

Add a Parallel Block to a Workflow

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

One of the unique features of PowerShell Workflow is the ability to define a block of activities as parallel. To use this feature, use the `parallel` keyword inside your Workflow.

Calling workflow activities in parallel may help to improve performance of your workflow.

Read PowerShell Workflows online: <https://riptutorial.com/powershell/topic/8745/powershell-workflows>

Chapter 50: PowerShell.exe Command-Line

Parameters

| Parameter | Description |
|--|---|
| -Help -? /? | Shows the help |
| -File <FilePath> [<Args>] | Path to script-file that should be executed and arguments (optional) |
| -Command { - <script-block> [-args <arg-array>] <string> [<CommandParameters>] } | Commands to be executed followed by arguments |
| -EncodedCommand <Base64EncodedCommand> | Base64 encoded commands |
| -ExecutionPolicy <ExecutionPolicy> | Sets the execution policy for this process only |
| -InputFormat { Text XML } | Sets input format for data sent to process. Text (strings) or XML (serialized CLIXML) |
| -Mta | PowerShell 3.0+: Runs PowerShell in multi-threaded apartment (STA is default) |
| -Sta | PowerShell 2.0: Runs PowerShell in a single-threaded apartment (MTA is default) |
| -NoExit | Leaves PowerShell console running after executing the script/command |
| -NoLogo | Hides copyright-banner at launch |
| -NonInteractive | Hides console from user |
| -NoProfile | Avoid loading of PowerShell profiles for machine or user |
| -OutputFormat { Text XML } | Sets output format for data returned from PowerShell. Text (strings) or XML (serialized CLIXML) |
| -PSConsoleFile <FilePath> | Loads a pre-created console file that configures the environment (created using <code>Export-Console</code>) |

| Parameter | Description |
|--|---|
| <code>-Version <Windows PowerShell version></code> | Specify a version of PowerShell to run. Mostly used with <code>2.0</code> |
| <code>-WindowStyle <style></code> | Specifies whether to start the PowerShell process as a <code>normal</code> , <code>hidden</code> , <code>minimized</code> or <code>maximized</code> window. |

Examples

Executing a command

The `-Command` parameter is used to specify commands to be executed on launch. It supports multiple data inputs.

-Command <string>

You can specify commands to be executed on launch as a string. Multiple semicolon `;`-separated statements may be executed.

```
>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 }

The `-Command` parameter also supports a scriptblock input (one or multiple statements wrapped in braces `{ #code }`). This only works when calling `PowerShell.exe` from another Windows PowerShell session.

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

-Command - (standard input)

You can pass in commands from the standard input by using `-Command -`. The standard input can come from `echo`, reading a file, a legacy console application etc.

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

Executing a script file

You can specify a file to a `ps1`-script to execute its content on launch using the `-File` parameter.

Basic script

MyScript.ps1

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

Output:

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

Using parameters and arguments

You can add parameters and/or arguments after filepath to use them in the script. Arguments will be used as values for undefined/available script-parameters, the rest will be available in the `$args`-array

MyScript.ps1

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

Output:

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

Read [PowerShell.exe Command-Line](https://riptutorial.com/powershell/topic/5839/powershell-exe-command-line) online:

<https://riptutorial.com/powershell/topic/5839/powershell-exe-command-line>

Chapter 51: PSScriptAnalyzer - PowerShell Script Analyzer

Introduction

PSScriptAnalyzer, <https://github.com/PowerShell/PSScriptAnalyzer>, is a static code checker for Windows PowerShell modules and scripts. PSScriptAnalyzer checks the quality of Windows PowerShell code by running a set of rules based on PowerShell best practices identified by the PowerShell Team and community. It generates DiagnosticResults (errors and warnings) to inform users about potential code defects and suggests possible solutions for improvements.

```
PS> Install-Module -Name PSScriptAnalyzer
```

Syntax

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

Analyzing scripts with the built-in preset rulesets

ScriptAnalyzer ships with sets of built-in preset rules that can be used to analyze scripts. These include: `PSGallery`, `DSC` and `CodeFormatting`. They can be executed as follows:

PowerShell Gallery rules

To execute the PowerShell Gallery rules use the following command:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings PSGallery -Recurse
```

DSC rules

To execute the DSC rules use the following command:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings DSC -Recurse
```

Code formatting rules

To execute the code formatting rules use the following command:

```
Invoke-ScriptAnalyzer -Path /path/to/module/ -Settings CodeFormatting -Recurse
```

Analyzing scripts against every built-in rule

To run the script analyzer against a single script file execute:

```
Invoke-ScriptAnalyzer -Path myscript.ps1
```

This will analyze your script against every built-in rule. If your script is sufficiently large that could result in a lot of warnings and/or errors.

To run the script analyzer against a whole directory, specify the folder containing the script, module and DSC files you want analyzed. Specify the Recurse parameter if you also want sub-directories searched for files to analyze.

```
Invoke-ScriptAnalyzer -Path . -Recurse
```

List all built-in rules

To see all the built-in rules execute:

```
Get-ScriptAnalyzerRule
```

Read PSScriptAnalyzer - PowerShell Script Analyzer online:

<https://riptutorial.com/powershell/topic/9619/psscriptanalyzer---powershell-script-analyzer>

Chapter 52: Regular Expressions

Syntax

- 'text' -match 'RegexPattern'
- 'text' -replace 'RegexPattern', 'newvalue'
- [regex]::Match("text","pattern") #Single match
- [regex]::Matches("text","pattern") #Multiple matches
- [regex]::Replace("text","pattern","newvalue")
- [regex]::Replace("text","pattern", {param(\$m) }) #MatchEvaluator
- [regex]::Escape("input") #Escape special characters

Examples

Single match

You can quickly determine if a text includes a specific pattern using Regex. There are multiple ways to work with Regex in PowerShell.

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

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

Using the -Match operator

To determine if a string matches a pattern using the built-in `-matches` operator, use the syntax `'input' -match 'pattern'`. This will return `true` or `false` depending on the result of the search. If there was match you can view the match and groups (if defined in pattern) by accessing the `$Matches`-variable.

```
> $text -match $pattern
True

> $Matches

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

You can also use `-match` to filter through an array of strings and only return the strings containing a

match.

```
> $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

Using Select-String

PowerShell 2.0 introduced a new cmdlet for searching through text using regex. It returns a `MatchInfo` object per textinput that contains a match. You can access it's properties to find matching groups etc.

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

> $m

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

> $m | Format-List *

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

Like `-match`, `Select-String` can also be used to filter through an array of strings by piping an array to it. It creates a `MatchInfo`-object per string that includes a match.

```
> $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` can also search using a normal text-pattern (no regex) by adding the `-SimpleMatch` switch.

Using [Regex]::Match()

You can also use the static `Match()` method available in the .NET `[Regex]`-class.

```
> [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)
```

Replace

A common task for regex is to replace text that matches a pattern with a new value.

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

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

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

Using -Replace operator

The `-replace` operator in PowerShell can be used to replace text matching a pattern with a new value using the syntax `'input' -replace 'pattern', 'newvalue'`.

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

Using [Regex]::Replace() method

Replacing matches can also be done using the `Replace()` method in the `[Regex]` .NET class.

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

Replace text with dynamic value using a MatchEvaluator

Sometimes you need to replace a value matching a pattern with a new value that's based on that specific match, making it impossible to predict the new value. For these types of scenarios, a `MatchEvaluator` can be very useful.

In PowerShell, a `MatchEvaluator` is as simple as a scriptblock with a single parameter that contains a `Match`-object for the current match. The output of the action will be the new value for that specific match. `MatchEvaluator` can be used with the `[Regex]::Replace()` static method.

Example: Replacing the text inside `()` with it's length

```
#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
}


```

Output:

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

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

Example: Make `sample` upper-case

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

$MatchEvaluator = {
    param($match)

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

Output:

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

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

Escape special characters

A regex-pattern uses many special characters to describe a pattern. Ex., `.` means "any character", `+` is "one or more" etc.

To use these characters, as a `.`, `+` etc., in a pattern, you need to escape them to remove their special meaning. This is done by using the escape character which is a backslash `\` in regex. Example: To search for `+`, you would use the pattern `\+`.

It can be hard to remember all special characters in regex, so to escape every special character in a string you want to search for, you could use the `[Regex]::Escape("input")` method.

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

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

Multiple matches

There are multiple ways to find all matches for a pattern in a text.

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

```
text, this is
a (sample text)
"@

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

Using Select-String

You can find all matches (global match) by adding the `-AllMatches` switch to `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)
```

Using [Regex]::Matches()

The `Matches()` method in the .NET `[regex]`-class can also be used to do a global search for multiple matches.

```
> [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)
```

Read Regular Expressions online: <https://riptutorial.com/powershell/topic/6674/regular-expressions>

Chapter 53: Return behavior in PowerShell

Introduction

It can be used to Exit the current scope, which can be a function, script, or script block. In PowerShell, the result of each statement is returned as output, even without an explicit Return keyword or to indicate that the end of the scope has been reached.

Remarks

You can read more about the return semantics on the [about_Return](#) page on TechNet, or by invoking `get-help return` from a PowerShell prompt.

Notable Q&A question(s) with more examples/explanation:

- [Function return value in PowerShell](#)
- [PowerShell: Function doesn't have proper return value](#)

[about_return](#) on MSDN explains it succinctly:

The Return keyword exits a function, script, or script block. It can be used to exit a scope at a specific point, to return a value, or to indicate that the end of the scope has been reached.

Users who are familiar with languages like C or C# might want to use the Return keyword to make the logic of leaving a scope explicit.

In Windows PowerShell, the results of each statement are returned as output, even without a statement that contains the Return keyword. Languages like C or C# return only the value or values that are specified by the Return keyword.

Examples

Early exit

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

"Hello" will be placed in the output pipeline, "World" will not

Gotcha! Return in the pipeline


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

Pipeline cmdlets (ex: `ForEach-Object`, `Where-Object`, etc) operate on closures. The return here will only move to the next item on the pipeline, not exit processing. You can use **break** instead of **return** if you want to exit processing.

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

Gotcha! Ignoring unwanted output

Inspired by

- [PowerShell: Function doesn't have proper return value](#)

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

The `Out-Null` is necessary because the .NET `ArrayList.Add` method returns the number of items in the collection after adding. If omitted, the pipeline would have contained `1, 2, "a", "b"`

There are multiple ways to omit unwanted output:

```
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"  
}
```

Note: to learn more about why to prefer `> $null`, see [topic not yet created].

Return with a value

(paraphrased from [about_return](#))

The following methods will have the same values on the pipeline

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

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

How to work with functions returns

A function returns everything that is not captured by something else.

If u use the **return** keyword, every statement after the return line will not be executed!

Like this:

```
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

Will return:

- Start
- The newly created registry key (this is because there are some statements that create output that you may not be expecting)
- Yes, it worked!

Test-Function -ExceptionalReturn Will return:

- Start
- Damn, it didn't work!

If you do it like this:

```
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

Will return:

- Yes, it worked!

Test-Function -ExceptionalReturn Will return:

- Damn, it didn't work!

With this trick you can control the returned output even if you are not sure what will each statement will spit out.

It works like this

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

the `.` makes the following scriptblock included in the code

the `{}` marks the script block

the `| Out-Null` pipes any unexpected output to Out-Null (so it is gone!)

Because the scriptblock is included it gets the same scope as the rest of the function.

So you can access variables who were made inside the scriptblock.

Read Return behavior in PowerShell online: <https://riptutorial.com/powershell/topic/4781/return-behavior-in-powershell>

Chapter 54: Running Executables

Examples

Console Applications

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

GUI Applications

```
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)
```

GUI applications launch in a different process, and will immediately return control to the PowerShell host. Sometimes you need the application to finish processing before the next PowerShell statement must be executed. This can be achieved by piping the application output to `$null` (3) or by using `Start-Process` with the `-Wait` switch (5).

Console Streams

```
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 contains `System.Management.Automation.ErrorRecord` objects. Note that some applications like `git.exe` use the "error stream" for informational purposes, that are not necessarily errors at all. In this case it is best to look at the exit code to determine whether the error stream should be interpreted as errors.

PowerShell understands these streams: Output, Error, Warning, Verbose, Debug, Progress. Native applications commonly use only these streams: Output, Error, Warning.

In PowerShell 5, all streams can be redirected to the standard output/success stream (6).

In earlier PowerShell versions, only specific streams can be redirected to the standard output/success stream (7). In this example, the "error stream" will be redirected to the output stream.

Exit Codes

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

These are built-in PowerShell variables that provide additional information about the most recent error. `$LastExitCode` is the final exit code of the last native application that was executed. `$?` and `$Error[0]` is the last error record that was generated by PowerShell.

Read **Running Executables** online: <https://riptutorial.com/powershell/topic/7707/running-executables>

Chapter 55: Scheduled tasks module

Introduction

Examples of how to use the Scheduled Tasks module available in Windows 8/Server 2012 and on.

Examples

Run PowerShell Script in Scheduled Task

Creates a scheduled task that executes immediately, then on start up to run `C:\myscript.ps1` as 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
```

Read Scheduled tasks module online: <https://riptutorial.com/powershell/topic/10940/scheduled-tasks-module>

Chapter 56: Security and Cryptography

Examples

Calculating a string's hash codes via .Net Cryptography

Utilizing .Net `System.Security.Cryptography.HashAlgorithm` namespace to generate the message hash code with the algorithms supported.

```
$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
```

The "sha256" part was the hash algorithm used.

the - can be removed or change to lower case

```
#convert to lower case hex without '-'
[System.BitConverter]::ToString($hash).Replace("-", "").ToLower()

#2edfdada56525b1290ff16fb1744cfb482dd2914ffbcb649790c0e589e462d3d
```

If base64 format was preferred, using base64 converter for output

```
#convert to base64
[Convert]::ToBase64String($hash)

#Lt/a2lZSWxKQ/xb7F0TPtILdKRT/vLZJeQwOWJ5GLT0=
```

Read Security and Cryptography online: <https://riptutorial.com/powershell/topic/5683/security-and-cryptography>

Chapter 57: Sending Email

Introduction

A useful technique for Exchange Server administrators is to be able to send email messages via SMTP from PowerShell. Depending on the version of PowerShell installed on your computer or server, there are multiple ways to send emails via powershell. There is a native cmdlet option that is simple and easy to use. It uses the cmdlet **Send-MailMessage**.

Parameters

| Parameter | Details |
|----------------------------|--|
| Attachments<String[]> | Path and file names of files to be attached to the message. Paths and filenames can be piped to Send-MailMessage. |
| Bcc<String[]> | Email addresses that receive a copy of an email message but does not appear as a recipient in the message. Enter names (optional) and the email address (required), such as Name someone@example.com or someone@example.com. |
| Body <String_> | Content of the email message. |
| BodyAsHtml | It indicates that the content is in HTML format. |
| Cc<String[]> | Email addresses that receive a copy of an email message. Enter names (optional) and the email address (required), such as Name someone@example.com or someone@example.com. |
| Credential | Specifies a user account that has permission to send message from specified email address. The default is the current user. Enter name such as User or Domain\User, or enter a PSCredential object. |
| DeliveryNotificationOption | Specifies the delivery notification options for the email message. Multiple values can be specified. Delivery notifications are sent in message to address specified in To parameter. Acceptable values: None, OnSuccess, OnFailure, Delay, Never. |
| Encoding | Encoding for the body and subject. Acceptable values: ASCII, UTF8, UTF7, UTF32, Unicode, BigEndianUnicode, Default, OEM. |
| From | Email addresses from which the mail is sent. Enter names (optional) and the email address (require), such as Name someone@example.com or someone@example.com. |

| Parameter | Details |
|------------|---|
| Port | Alternate port on the SMTP server. The default value is 25. Available from Windows PowerShell 3.0. |
| Priority | Priority of the email message. Acceptable values: Normal, High, Low. |
| SmtpServer | Name of the SMTP server that sends the email message. Default value is the value of the \$PSEmailServer variable. |
| Subject | Subject of the email message. |
| To | Email addresses to which the mail is sent. Enter names (optional) and the email address (required), such as Name someone@example.com or someone@example.com |
| UseSsl | Uses the Secure Sockets Layer (SSL) protocol to establish a connection to the remote computer to send mail |

Examples

Simple Send-MailMessage

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

Send-MailMessage with predefined parameters

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

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

SMTPClient - Mail with .txt file in body message

```
# 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
```

Read Sending Email online: <https://riptutorial.com/powershell/topic/2040/sending-email>

Chapter 58: SharePoint Module

Examples

Loading SharePoint Snap-In

Loading the SharePoint Snapin can be done using the following:

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

This only works in the 64bit version of PowerShell. If the window says "Windows PowerShell (x86)" in the title you are using the incorrect version.

If the Snap-In is already loaded, the code above will cause an error. Using the following will load only if necessary, which can be used in Cmdlets/functions:

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

Alternatively, if you start the SharePoint Management Shell, it will automatically include the Snap-In.

To get a list of all the available SharePoint Cmdlets, run the following:

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

Iterating over all lists of a site collection

Print out all list names and the item count.

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

Get all installed features on a site collection

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

Get-SPFeature can also be run on web scope (-Web <WebUrl>), farm scope (-Farm) and web application scope (-WebApplication <WebAppUrl>).

Get all orphaned features on a site collection

Another usage of Get-SPFeature can be to find all features that have no scope:

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

Read SharePoint Module online: <https://riptutorial.com/powershell/topic/5147/sharepoint-module>

Chapter 59: Signing Scripts

Remarks

Signing a script will make your scripts comply with all execution policies in PowerShell and ensure the integrity of a script. Signed scripts will fail to run if they have been modified after being signed.

Scripts signing requires a code signing certificate. Recommendations:

- Personal scripts/testing (not shared): Certificate from trusted certificate authority (internal or third-party) **OR** a self-signed certificate.
- Shared inside organization: Certificate from trusted certificate authority (internal or third-party)
- Shared outside organization: Certificate from trusted third party certificate authority

Read more at [about_Signing @ TechNet](#)

Execution policies

PowerShell has configurable execution policies that control which conditions are required for a script or configuration to be executed. An execution policy can be set for multiple scopes; computer, current user and current process. **Execution policies can easily be bypassed and is not designed to restrict users, but rather protect them from violating signing policies unintentionally.**

The available policies are:

| Setting | Description |
|--------------|--|
| Restricted | No scripts allowed |
| AllSigned | All scripts need to be signed |
| RemoteSigned | All local scripts allowed; only signed remote scripts |
| Unrestricted | No requirements. All scripts allowed, but will warn before running scripts downloaded from the internet |
| Bypass | All scripts are allowed and no warnings are displayed |
| Undefined | Remove the current execution policy for the current scope. Uses the parent policy. If all policies are undefined, restricted will be used. |

You can modify the current execution policies using `Set-ExecutionPolicy`-cmdlet, Group Policy or the `-ExecutionPolicy` parameter when launching a `powershell.exe` process.

Read more at [about_Execution_Policies @ TechNet](#)

Examples

Signing a script

Signing a script is done by using the `Set-AuthenticodeSignature`-cmdlet and a code-signing certificate.

```
#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
```

You can also read a certificate from a `.pfx`-file using:

```
$cert = Get-PfxCertificate -FilePath "C:\MyCodeSigningCert.pfx"
```

The script will be valid until the certificate expires. If you use a timestamp-server during the signing, the script will continue to be valid after the certificate expires. It is also useful to add the trust chain for the certificate (including root authority) to help most computers trust the certificated used to sign the script.

```
Set-AuthenticodeSignature -Certificate $cert -FilePath c:\MyScript.ps1 -IncludeChain All -
TimeStampServer "http://timestamp.verisign.com/scripts/timestamp.dll"
```

It's recommended to use a timestamp-server from a trusted certificate provider like Verisign, Comodo, Thawte etc.

Changing the execution policy using Set-ExecutionPolicy

To change the execution policy for the default scope (LocalMachine), use:

```
Set-ExecutionPolicy AllSigned
```

To change the policy for a specific scope, use:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy AllSigned
```

You can suppress the prompts by adding the `-Force` switch.

Bypassing execution policy for a single script

Often you might need to execute an unsigned script that doesn't comply with the current execution policy. An easy way to do this is by bypassing the execution policy for that single process.

Example:

```
powershell.exe -ExecutionPolicy Bypass -File C:\MyUnsignedScript.ps1
```

Or you can use the shorthand:

```
powershell -ep Bypass C:\MyUnsignedScript.ps1
```

Other Execution Policies:

| Policy | Description |
|---------------|--|
| AllSigned | Only scripts signed by a trusted publisher can be run. |
| Bypass | No restrictions; all Windows PowerShell scripts can be run. |
| Default | Normally RemoteSigned, but is controlled via ActiveDirectory |
| RemoteSigned | Downloaded scripts must be signed by a trusted publisher before they can be run. |
| Restricted | No scripts can be run. Windows PowerShell can be used only in interactive mode. |
| Undefined | NA |
| Unrestricted* | Similar to bypass |

Unrestricted **Caveat:** If you run an unsigned script that was downloaded from the Internet, you are prompted for permission before it runs.*

More Information available [here](#).

Get the current execution policy

Getting the effective execution policy for the current session:

```
PS> Get-ExecutionPolicy
RemoteSigned
```

List all effective execution policies for the current session:

```
PS> Get-ExecutionPolicy -List

Scope ExecutionPolicy
-----
MachinePolicy Undefined
UserPolicy Undefined
Process Undefined
CurrentUser Undefined
```

```
LocalMachine RemoteSigned
```

List the execution policy for a specific scope, ex. process:

```
PS> Get-ExecutionPolicy -Scope Process
Undefined
```

Getting the signature from a signed script

Get information about the Authenticode signature from a signed script by using the `Get-AuthenticodeSignature-cmdlet`:

```
Get-AuthenticodeSignature .\MyScript.ps1 | Format-List *
```

Creating a self-signed code signing certificate for testing

When signing personal scripts or when testing code signing it can be useful to create a self-signed code signing certificate.

5.0

Beginning with PowerShell 5.0 you can generate a self-signed code signing certificate by using the `New-SelfSignedCertificate-cmdlet`:

```
New-SelfSignedCertificate -FriendlyName "StackOverflow Example Code Signing" -
CertStoreLocation Cert:\CurrentUser\My -Subject "SO User" -Type CodeSigningCert
```

In earlier versions, you can create a self-signed certificate using the `makecert.exe` tool found in the .NET Framework SDK and Windows SDK.

A self-signed certificate will only be trusted by computers that have installed the certificate. For scripts that will be shared, a certificate from a trusted certificate authority (internal or trusted third-party) are recommended.

Read Signing Scripts online: <https://riptutorial.com/powershell/topic/5670/signing-scripts>

Chapter 60: Special Operators

Examples

Array Expression Operator

Returns the expression as an array.

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

Will return an array with one item

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

Will return an array with all the items in the folder (which is not a change of behavior from the inner expression).

Call Operation

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

Will execute `Get-ChildItem`

Dot sourcing operator

```
.\myScript.ps1
```

runs `.\myScript.ps1` in the current scope making any functions, and variable available in the current scope.

Read Special Operators online: <https://riptutorial.com/powershell/topic/8981/special-operators>

Chapter 61: Splatting

Introduction

Splatting is a method of passing multiple parameters to a command as a single unit. This is done by storing the parameters and their values as key-value pairs in a [hashtable](#) and splatting it to a cmdlet using the splatting operator `@`.

Splatting can make a command more readable and allows you to reuse parameters in multiple command calls.

Remarks

Note: The [Array expression operator](#) `@()` have very different behavior than the Splatting operator `@`.

Read more at [about_Splatting @ TechNet](#)

Examples

Splatting parameters

Splatting is done by replacing the dollar-sign `$` with the splatting operator `@` when using a variable containing a [HashTable](#) of parameters and values in a command call.

```
$MyParameters = @{
    Name = "iexplore"
    FileVersionInfo = $true
}

Get-Process @MyParameters
```

Without splatting:

```
Get-Process -Name "iexplore" -FileVersionInfo
```

You can combine normal parameters with splatted parameters to easily add common parameters to your calls.

```
$MyParameters = @{
    ComputerName = "StackOverflow-PC"
}

Get-Process -Name "iexplore" @MyParameters

Invoke-Command -ScriptBlock { "Something to excute remotely" } @MyParameters
```

Passing a Switch parameter using Splatting

To use Splatting to call `Get-Process` with the `-FileVersionInfo` switch similar to this:

```
Get-Process -FileVersionInfo
```

This is the call using splatting:

```
$MyParameters = @{  
    FileVersionInfo = $true  
}  
  
Get-Process @MyParameters
```

Note: This is useful because you can create a default set of parameters and make the call many times like this

```
$MyParameters = @{  
    FileVersionInfo = $true  
}  
  
Get-Process @MyParameters -Name WmiPrvSE  
Get-Process @MyParameters -Name explorer
```

Piping and Splatting

Declaring the splat is useful for reusing sets of parameters multiple times or with slight variations:

```
$splat = @{  
    Class = "Win32_SystemEnclosure"  
    Property = "Manufacturer"  
    ErrorAction = "Stop"  
}  
  
Get-WmiObject -ComputerName $env:COMPUTERNAME @splat  
Get-WmiObject -ComputerName "Computer2" @splat  
Get-WmiObject -ComputerName "Computer3" @splat
```

However, if the splat is not indented for reuse, you may not wish to declare it. It can be piped instead:

```
@{  
    ComputerName = $env:COMPUTERNAME  
    Class = "Win32_SystemEnclosure"  
    Property = "Manufacturer"  
    ErrorAction = "Stop"  
} | % { Get-WmiObject @_ }
```

Splatting From Top Level Function to a Series of Inner Function

Without splatting it is very cumbersome to try and pass values down through the call stack. But if

you combine splatting with the power of the **@PSBoundParameters** then you can pass the top level parameter collection down through the layers.

```
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
```

Read Splatting online: <https://riptutorial.com/powershell/topic/5647/splatting>

Chapter 62: Strings

Syntax

- "(Double-quoted) String"
- 'Literal string'
- @"
Here-string
"@
- @'
Literal here-string
'@"

Remarks

Strings are objects representing text.

Examples

Creating a basic string

String

Strings are created by wrapping the text with double quotes. Double-quoted strings can evaluate variables and special characters.

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

To use a double quote inside a string it needs to be escaped using the escape character, backtick (`). Single quotes can be used inside a double-quoted string.

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

Literal string

Literal strings are strings that doesn't evaluate variables and special characters. It's created using single quotes.

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

To use single quotes inside a literal string, use double single quotes or a literal here-string. Double quotes can be used safely inside a literal string

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

Format string

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

Format strings can be used with the `-f` operator or the static `[String]::Format(string format, args)` .NET method.

Multiline string

There are multiple ways to create a multiline string in PowerShell:

- You can use the special characters for carriage return and/or newline manually or use the `NewLine`-environment variable to insert the systems "newline" value)

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

- Create a linebreak while defining a string (before closing quote)

```
"Hello  
World"
```

- Using a here-string. *This is the most common technique.*

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

Here-string

Here-strings are very useful when creating multiline strings. One of the biggest benefits compared to other multiline strings are that you can use quotes without having to escape them using a backtick.

Here-string

Here-strings begin with `@` and a linebreak and end with `@` on it's own line (**@ must be first characters on the line, not even whitespace/tab**).

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

Literal here-string

You could also create a literal here-string by using single quotes, when you don't want any expressions to be expanded just like a normal literal string.

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

Concatenating strings

Using variables in a string

You can concatenate strings using variables inside a double-quoted string. This does not work with properties.

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

Using the + operator

You can also join strings using the + operator.

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

This also works with properties of objects.

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

Using subexpressions

The output/result of a subexpressions `$()` can be used in a string. This is useful when accessing properties of an object or performing a complex expression. Subexpressions can contain multiple statements separated by semicolon ;

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

Special characters

When used inside a double-quoted string, the escape character (backtick ```) represents a special character.

```
`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)
```

Example:

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

You can also escape special characters with special meanings:

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

Read Strings online: <https://riptutorial.com/powershell/topic/5124/strings>

Chapter 63: Switch statement

Introduction

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a *case*, and the variable being *switched* on is checked for each switch case. It enables you to write a script that can choose from a series of options, but without requiring you to write a long series of if statements.

Remarks

This topic is documenting the ***switch statement*** used for branching the flow of the script. Do not confuse it with ***switch parameters*** which are used in functions as boolean flags.

Examples

Simple Switch

Switch statements compare a single test value to multiple conditions, and performs any associated actions for successful comparisons. It can result in multiple matches/actions.

Given the following switch...

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

'First Action' will be output if \$myValue is set as 'First Condition'.

'Section Action' will be output if \$myValue is set as 'Second Condition'.

Nothing will be output if \$myValue does not match either conditions.

Switch Statement with Regex Parameter

The `-Regex` parameter allows switch statements to perform regular expression matching against conditions.

Example:

```
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'}
}
```

Output:

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

Simple Switch With Break

The `break` keyword can be used in switch statements to exit the statement before evaluating all conditions.

Example:

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

Output:

```
First Action
Second Action
```

Because of the `break` keyword in the second action, the third condition is not evaluated.

Switch Statement with Wildcard Parameter

The `-Wildcard` parameter allows switch statements to perform wildcard matching against conditions.

Example:

```
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'}
}
```

Output:

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

Switch Statement with Exact Parameter

The `-Exact` parameter enforces switch statements to perform exact, case-insensitive matching against string-conditions.

Example:

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

Output:

```
First Action
Second Action
Third Action
```

The first through third actions are executed because their associated conditions matched the input. The regex and wildcard strings in the fourth and fifth conditions fail matching.

Note that the fourth condition would also match the input string if regular expression matching was being performed, but was ignored in this case because it is not.

Switch Statement with CaseSensitive Parameter

The `-CaseSensitive` parameter enforces switch statements to perform exact, case-sensitive matching against conditions.

Example:

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

```
}
```

Output:

```
Second Action
```

The second action is the only action executed because it is the only condition that exactly matches the string 'Condition' when accounting for case-sensitivity.

Switch Statement with File Parameter

The `-file` parameter allows the switch statement to receive input from a file. Each line of the file is evaluated by the switch statement.

Example file `input.txt`:

```
condition
test
```

Example switch statement:

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

Output:

```
First Action
Second Action
```

Simple Switch with Default Condition

The `Default` keyword is used to execute an action when no other conditions match the input value.

Example:

```
switch('Condition')
{
  'Skip Condition'
  {
    'First Action'
  }
  'Skip This Condition Too'
  {
    'Second Action'
  }
  Default
  {
```

```
'Default Action'  
}  
}
```

Output:

```
Default Action
```

Switch Statement with Expressions

Conditions can also be expressions:

```
$myInput = 0  
  
switch($myInput) {  
    # because the result of the expression, 4,  
    # does not equal our input this block should not be run.  
    (2+2) { 'True. 2 +2 = 4' }  
  
    # because the result of the expression, 0,  
    # does equal our input this block should be run.  
    (2-2) { 'True. 2-2 = 0' }  
  
    # because our input is greater than -1 and is less than 1  
    # the expression evaluates to true and the block should be run.  
    { $_ -gt -1 -and $_ -lt 1 } { 'True. Value is 0' }  
}  
  
#Output  
True. 2-2 = 0  
True. Value is 0
```

Read Switch statement online: <https://riptutorial.com/powershell/topic/1174/switch-statement>

Chapter 64: TCP Communication with PowerShell

Examples

TCP listener

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

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

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

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

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

Start listening with the following and capture any message in the variable `$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 writer
    $Stream = $Socket.GetStream()
    $Writer = New-Object System.IO.StreamWriter($Stream)

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

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

```

Send a message with:

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

Note: TCP messages may be blocked by your software firewall or any external facing firewalls you are trying to go through. Ensure that the TCP port you set in the above command is open and that you are have setup the listener on the same port.

Read TCP Communication with PowerShell online:

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

Chapter 65: URL Encode/Decode

Remarks

The regular expression used in the *Decode URL* examples was taken from [RFC 2396, Appendix B: Parsing a URI Reference with a Regular Expression](#); for posterity, here's a quote:

The following line is the regular expression for breaking-down a URI reference into its components.

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

The numbers in the second line above are only to assist readability; they indicate the reference points for each subexpression (i.e., each paired parenthesis). We refer to the value matched for subexpression as \$. For example, matching the above expression to

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

results in the following subexpression matches:

```
$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

Quick Start: Encoding

```
$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
```

Note: [More info on HTTPUtility](#).

Quick Start: Decoding

Note: these examples use the variables created in the *Quick Start: Encoding* section above.

```
# 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
```

Note: [More info on HTTPUtility.](#)

Encode Query String with `[uri]::EscapeDataString()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qqs_data = @{'fool'='bar1';
              'foo2'='complex;/?:@&+,$, bar''''';
              'complex;/?:@&+,$, foo''''='bar2'}
}

[System.Collections.ArrayList] $qqs_array = @()
foreach ($qs in $qqs_data.GetEnumerator()) {
    $qs_key = [uri]::EscapeDataString($qs.Name)
    $qs_value = [uri]::EscapeDataString($qs.Value)
    $qqs_array.Add("$qs_key=$qs_value") | Out-Null
}

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

With `[uri]::EscapeDataString()`, you will notice that the apostrophe (') was not encoded:

<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
```

Encode Query String with `[System.Web.HttpUtility]::UrlEncode()`

```
$scheme = 'https'
$url_format = '{0}://example.vertigion.com/foos?{1}'
$qqs_data = @{
    'foo1'='bar1';
    'foo2'= 'complex;/?:@&+$/, bar'''';
    'complex;/?:@&+$/, foo''''='bar2';
}

[System.Collections.ArrayList] $qqs_array = @()
foreach ($qs in $qqs_data.GetEnumerator()) {
    $qs_key = [System.Web.HttpUtility]::UrlEncode($qs.Name)
    $qs_value = [System.Web.HttpUtility]::UrlEncode($qs.Value)
    $qqs_array.Add("${qs_key}=${qs_value}") | Out-Null
}

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

With `[System.Web.HttpUtility]::UrlEncode()`, you will notice that spaces are turned into plus signs (+) instead of %20:

```
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
```

Decode URL with `[uri]::UnescapeDataString()`

Encoded with `[uri]::EscapeDataString()`

First, we'll decode the URL and Query String encoded with `[uri]::EscapeDataString()` in the above example:

```
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('=')
    $url_parts.QueryStringParts.Add(
        [uri]::UnescapeDataString($qs_key),
        [uri]::UnescapeDataString($qs_value)
    ) | Out-Null
}
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}

```

This gives you back `[hashtable]$url_parts`; which equals (**Note:** the spaces in the complex parts are spaces):

```

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

```

Encoded with `[System.Web.HttpUtility]::UrlEncode()`

Now, we'll decode the URL and Query String encoded with `[System.Web.HttpUtility]::UrlEncode()` in the above example:

[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](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"
}

```

This gives you back `[hashtable]$url_parts`, which equals (**Note:** the *spaces* in the complex parts are *plus signs* (+) in the first part and *spaces* in the second part):

```

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=bar%27%22=bar1

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

PS > $url_parts.QueryStringParts

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

```

Decode URL with `[System.Web.HttpUtility]::UrlDecode()`

Encoded with `[uri]::EscapeDataString()`

First, we'll decode the URL and Query String encoded with `[uri]::EscapeDataString()` in the above example:

[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](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"
}

```

This gives you back `[hashtable]$url_parts`; which equals (**Note:** the spaces in the complex parts are spaces):

```

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

```

Encoded with `[System.Web.HttpUtility]::UrlEncode()`

Now, we'll decode the URL and Query String encoded with `[System.Web.HttpUtility]::UrlEncode()` in the above example:

[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](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('=')
        $url_parts.QueryStringParts.Add(
            [System.Web.HttpUtility]::UrlDecode($qs_key),
            [System.Web.HttpUtility]::UrlDecode($qs_value)
        ) | Out-Null
    }
} else {
    Throw [System.Management.Automation.ParameterBindingException] "Invalid URL Supplied"
}

```

This gives you back `[hashtable]$url_parts`; which equals (**Note:** the spaces in the complex parts are spaces):

```

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

```

Read URL Encode/Decode online: <https://riptutorial.com/powershell/topic/7352/url-encode-decode>

Chapter 66: Using existing static classes

Introduction

These classes are reference libraries of methods and properties that do not change state, in one word, immutable. You don't need to create them, you simply use them. Classes and methods such as these are called static classes because they are not created, destroyed, or changed. You can refer to a static class by surrounding the class name with square brackets.

Examples

Creating new GUID instantly

Use existing .NET classes instantly with PowerShell by using `[class]::Method(args)`:

```
PS C:\> [guid]::NewGuid()

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

Similarly, in PowerShell 5+ you may use the `New-Guid` cmdlet:

```
PS C:\> New-Guid

Guid
----
8874a185-64be-43ed-a64c-d2fe4b6e31bc
```

To get the GUID as a `[String]` only, referenced the `.Guid` property:

```
[guid]::NewGuid().Guid
```

Using the .Net Math Class

You can use the .Net Math class to do calculations (`[System.Math]`)

If you want to know which methods are available you can use:

```
[System.Math] | Get-Member -Static -MemberType Methods
```

Here are some examples how to use the Math class:

```
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
```

Adding types

By Assembly Name, add library

```
Add-Type -AssemblyName "System.Math"
```

or by file path:

```
Add-Type -Path "D:\Libs\CustomMath.dll"
```

To Use added type:

```
[CustomMath.Namespace]::Method(param1, $variableParam, [int]castMeAsIntParam)
```

Read Using existing static classes online: <https://riptutorial.com/powershell/topic/1522/using-existing-static-classes>

Chapter 67: Using ShouldProcess

Syntax

- `$PSCmdlet.ShouldProcess("Target")`
- `$PSCmdlet.ShouldProcess("Target", "Action")`

Parameters

| Parameter | Details |
|-----------|--|
| Target | The resource being changed. |
| Action | The operation being performed. Defaults to the name of the cmdlet. |

Remarks

`$PSCmdlet.ShouldProcess()` will also automatically write a message to the verbose output.

```
PS> Invoke-MyCmdlet -Verbose
VERBOSE: Performing the operation "Invoke-MyCmdlet" on target "Target of action"
```

Examples

Adding -WhatIf and -Confirm support to your cmdlet

```
function Invoke-MyCmdlet {
    [CmdletBinding(SupportsShouldProcess = $true)]
    param()
    # ...
}
```

Using ShouldProcess() with one argument

```
if ($PSCmdlet.ShouldProcess("Target of action")) {
    # Do the thing
}
```

When using `-WhatIf`:

```
What if: Performing the action "Invoke-MyCmdlet" on target "Target of action"
```

When using `-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"):
```

Full Usage Example

Other examples couldn't clearly explain to me how to trigger the conditional logic.

This example also shows that underlying commands will also listen to the `-Confirm` flag!

```
<#
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
```

Read Using ShouldProcess online: <https://riptutorial.com/powershell/topic/1145/using-shouldprocess>

Chapter 68: Using the Help System

Remarks

`Get-Help` is a cmdlet for reading help topics in PowerShell.

Read more a [TechNet](#)

Examples

Updating the Help System

3.0

Beginning with PowerShell 3.0, you can download and update the offline help documentation using a single cmdlet.

```
Update-Help
```

To update help on multiple computers (or computers not connected to the internet).

Run the following on a computer with the help files

```
Save-Help -DestinationPath \\Server01\Share\PSHelp -Credential $Cred
```

To run on many computers remotely

```
Invoke-Command -ComputerName (Get-Content Servers.txt) -ScriptBlock {Update-Help -SourcePath \\Server01\Share\Help -Credential $cred}
```

Using Get-Help

`Get-Help` can be used to view help in PowerShell. You can search for cmdlets, functions, providers or other topics.

In order to view the help documentation about jobs, use:

```
Get-Help about_Jobs
```

You can search for topics using wildcards. If you want to list available help topics with a title starting with `about_`, try:

```
Get-Help about_*
```

If you wanted help on `Select-Object`, you would use:

```
Get-Help Select-Object
```

You can also use the aliases `help` or `man`.

Viewing online version of a help topic

You can access online help documentation using:

```
Get-Help Get-Command -Online
```

Viewing Examples

Show usage examples for a specific cmdlet.

```
Get-Help Get-Command -Examples
```

Viewing the Full Help Page

View the full documentation for the topic.

```
Get-Help Get-Command -Full
```

Viewing help for a specific parameter

You can view help for a specific parameter using:

```
Get-Help Get-Content -Parameter Path
```

Read [Using the Help System](https://riptutorial.com/powershell/topic/5644/using-the-help-system) online: <https://riptutorial.com/powershell/topic/5644/using-the-help-system>

Chapter 69: Using the progress bar

Introduction

A progress bar can be used to show something is in a process. It is a time-saving and slick feature one should have. Progress bars are incredibly useful while debugging to figure out which part of the script is executing, and they're satisfying for the people running scripts to track what's happening. It is common to display some kind of progress when a script takes a long time to complete. When a user launches the script and nothing happens, one begins to wonder if the script launched correctly.

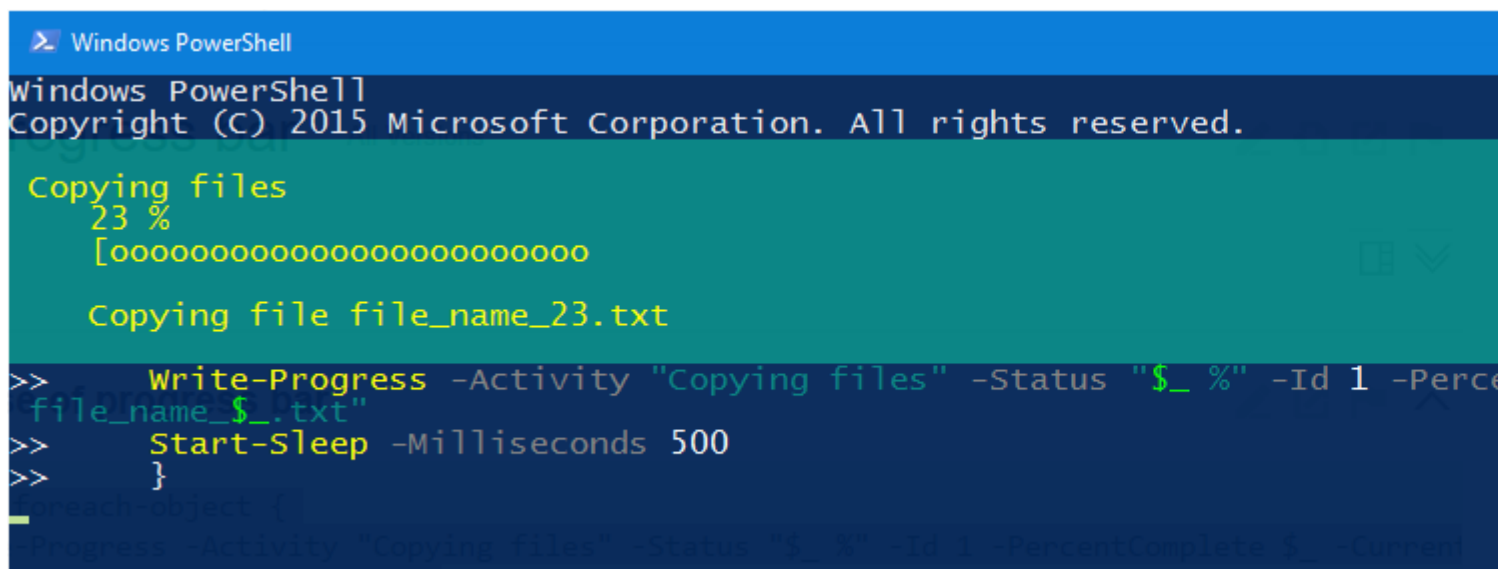
Examples

Simple use of progress bar

```
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)
}
```

Please note that for brevity this example does not contain any executive code (simulated with Start-Sleep). However it is possible to run it directly as is and then modify and play with it.

This is how result looks in PS console:



The screenshot shows a Windows PowerShell window with a blue title bar. The command prompt shows the execution of the script. The output displays a progress bar for 'Copying files' at 23% completion, with a green bar and a progress indicator. Below the progress bar, it shows 'Copying file file_name_23.txt'. The command prompt shows the script code being executed, including the Write-Progress and Start-Sleep commands.

This is how result looks in PS ISE:



Usage of inner progress bar

```
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)
}
```

Please note that for brevity this example does not contain any executive code (simulated with `Start-Sleep`). However it is possible to run it directly as is and then modify and play with it.

This is how result looks in PS console:

```
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

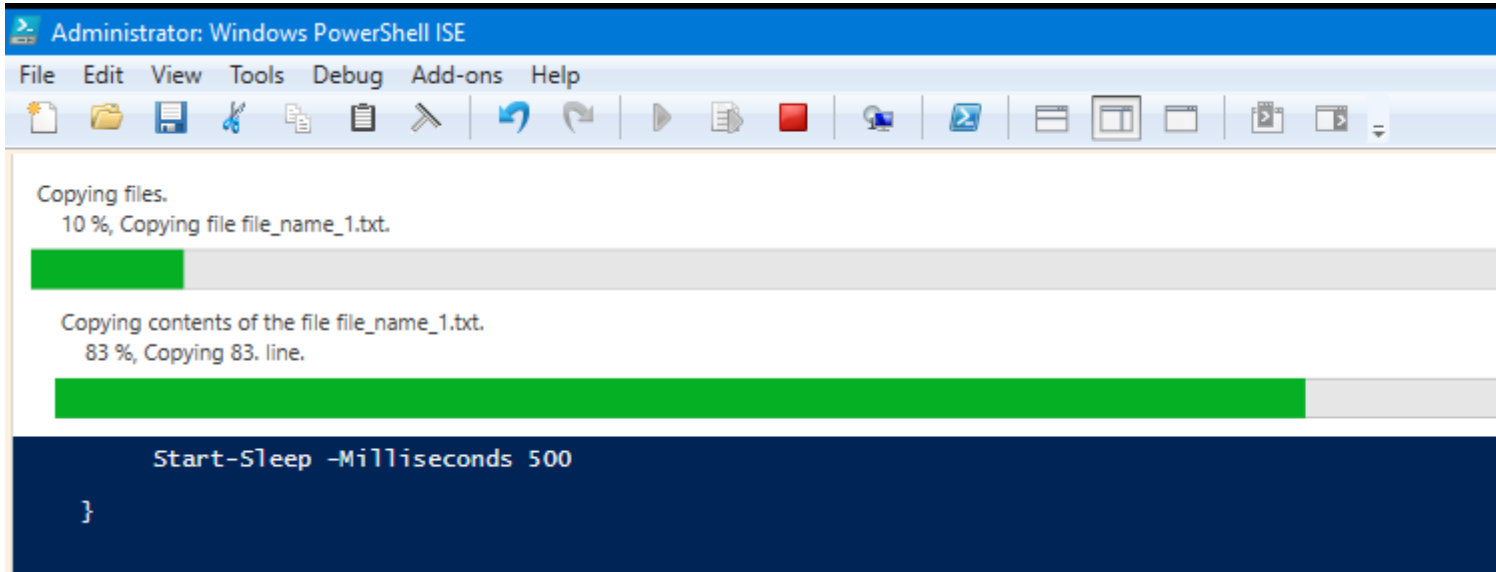
Copying files
30 %
[ooooooooooooooooooooooooooooooooooooo

Copying file file_name_3.txt
Copying contents of the file file_name_3.txt
46 %
[ooooooooooooooooooooooooooooooooooooooooooooo

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
>>> }
```

This is how result looks in PS ISE:



Read Using the progress bar online: <https://riptutorial.com/powershell/topic/5020/using-the-progress-bar>

Chapter 70: Variables in PowerShell

Introduction

Variables are used for storing values. Let the value be of any type, we need to store it somewhere so that we can use it throughout the console/script. Variable names in PowerShell begin with a \$, as in `$Variable1`, and values are assigned using `=`, like `$Variable1 = "Value 1"`. PowerShell supports a huge number of variable types; such as text strings, integers, decimals, arrays, and even advanced types like version numbers or IP addresses.

Examples

Simple variable

All variables in powershell begin with a US dollar sign (\$). The simplest example of this is:

```
$foo = "bar"
```

This statement allocates a variable called `foo` with a string value of "bar".

Removing a variable

To remove a variable from memory, one can use the `Remove-Item` cmdlet. Note: The variable name does NOT include the \$.

```
Remove-Item Variable:\foo
```

`Variable` has a provider to allow most `*-item` cmdlets to work much like file systems.

Another method to remove variable is to use `Remove-Variable` cmdlet and its 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
```

Scope

The default [scope](#) for a variable is the enclosing container. If outside a script, or other container then the scope is `Global`. To specify a [scope](#), it is prefixed to the variable name `$scope:varname` like so:

```
$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
```

Output:

```
Global Scope
Function (local) scope
Function (local) scope
Global Scope
Global Scope
```

Reading a CmdLet Output

By Default, powershell would return the output to the calling Entity. Consider Below Example,

```
Get-Process -Name excel
```

This would simply, return the running process which matches the name excel, to the calling entity. In this case, the PowerShell Host. It prints something like,

| Handles | NPM(K) | PM(K) | WS(K) | VM(M) | CPU(s) | Id | SI | ProcessName |
|---------|--------|-------|-------|-------|--------|------|----|-------------|
| ----- | ----- | ----- | ----- | ----- | ----- | -- | -- | ----- |
| 1037 | 54 | 67632 | 62544 | 617 | 5.23 | 4544 | 1 | EXCEL |

Now if you assign the output to a variable, it simply wont print anything. And of course the variable holds the output. (Be it a string, Object - Any type for that matter)

```
$allExcel = Get-Process -Name excel
```

So, lets say you have a scenario where you want to assign a variable by a Dynamic name, you can use the `-outvariable` parameter

```
Get-Process -Name excel -OutVariable AllRunningExcel
```

Note that the '\$' is missing here. A major difference between these two assignments is that, it also prints the output apart from assigning it to the variable AllRunningExcel. You can also choose to assign it to an another variable.

```
$VarOne = Get-Process -Name excel -OutVariable VarTwo
```

Albeit, the above scenario is very rare, both variables \$VarOne & \$VarTwo will have the same value.

Now consider this,

```
Get-Process -Name EXCEL -OutVariable MSOFFICE
Get-Process -Name WINWORD -OutVariable +MSOFFICE
```

The first statement would simply get excel process & assign it to MSOFFICE variable, and next would get ms word processes running and "Append" it to the existing value of MSOFFICE. It would look something like this,

| 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 |

List Assignment of Multiple Variables

Powershell allows multiple assignment of variables and treats almost everything like an array or list. This means that instead of doing something like this:

```
$input = "foo.bar.baz"
$parts = $input.Split(".")
$foo = $parts[0]
$bar = $parts[1]
$baz = $parts[2]
```

You can simply do this:

```
$foo, $bar, $baz = $input.Split(".")
```

Since Powershell treats assignments in this manner like lists, if there are more values in the list than items in your list of variables to assign them to, the last variable becomes an array of the remaining values. This means you can also do things like this:

```
$foo, $leftover = $input.Split(".") #Sets $foo = "foo", $leftover = ["bar","baz"]
$bar = $leftover[0] # $bar = "bar"
$baz = $leftover[1] # $baz = "baz"
```

Arrays

Array declaration in Powershell is almost the same as instantiating any other variable, i.e. you use a \$name = syntax. The items in the array are declared by separating them by commas(,):

```
$myArrayOfInts = 1,2,3,4
$myArrayOfStrings = "1","2","3","4"
```

Adding to an array

Adding to an array is as simple as using the + operator:

```
$myArrayOfInts = $myArrayOfInts + 5  
//now contains 1,2,3,4 & 5!
```

Combining arrays together

Again this is as simple as using the + operator

```
$myArrayOfInts = 1,2,3,4  
$myOtherArrayOfInts = 5,6,7  
$myArrayOfInts = $myArrayOfInts + $myOtherArrayOfInts  
//now 1,2,3,4,5,6,7
```

Read Variables in PowerShell online: <https://riptutorial.com/powershell/topic/3457/variables-in-powershell>

Chapter 71: WMI and CIM

Remarks

CIM vs WMI

As of PowerShell 3.0, there are two ways to work with management classes in PowerShell, WMI and CIM. PowerShell 1.0 and 2.0 only supported the WMI-module which is now superseded by the new and improved CIM-module. In a later release of PowerShell, the WMI-cmdlets will be removed.

Comparison of CIM and WMI-modules:

| CIM-cmdlet | WMI-cmdlet | What it does |
|-----------------------------|--------------------|--|
| Get-CimInstance | Get-WmiObject | Gets CIM/WMI-objects for a class |
| Invoke-CimMethod | Invoke-WmiMethod | Invokes a CIM/WMI-class method |
| Register-CimIndicationEvent | Register-WmiEvent | Registers event for a CIM/WMI-class |
| Remove-CimInstance | Remove-WmiObject | Remove CIM/WMI-object |
| Set-CimInstance | Set-WmiInstance | Updates/Saves CIM/WMI-object |
| Get-CimAssociatedInstance | N/A | Get associated instances (linked object/classes) |
| Get-CimClass | Get-WmiObject-List | List CIM/WMI-classes |
| New-CimInstance | N/A | Create new CIM-object |
| Get-CimSession | N/A | Lists CIM-sessions |
| New-CimSession | N/A | Create new CIM-session |
| New-CimSessionOption | N/A | Creates object with session options; protocol, encoding, disable encryption etc. (for use with <code>New-CimSession</code>) |
| Remove-CimSession | N/A | Removes/Stops CIM-session |

Additional resources

[Should I use CIM or WMI with Windows PowerShell? @ Hey, Scripting Guy! Blog](#)

Examples

Querying objects

CIM/WMI is most commonly used to query information or configuration on a device. Thof a class that represents a configuration, process, user etc. In PowerShell there are multiple ways to access these classes and instances, but the most common ways are by using the `Get-CimInstance` (CIM) or `Get-WmiObject` (WMI) cmdlets.

List all objects for CIM-class

You can list all instances of a class.

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
```

Using a filter

You can apply a filter to only get specific instances of a CIM/WMI-class. Filters are written using WQL (default) or CQL (add `-QueryDialect CQL`). `-Filter` uses the `WHERE`-part of a full WQL/CQL-query.

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
....
```

Using a WQL-query:

You can also use a WQL/CQL-query to query and filter instances.

3.0

CIM:

```
Get-CimInstance -Query "SELECT * FROM Win32_Process WHERE Name = 'powershell.exe'"
```

| ProcessId | Name | HandleCount | WorkingSetSize | VirtualSize |
|-----------|----------------|-------------|----------------|---------------|
| 4800 | powershell.exe | 673 | 88387584 | 2199696674816 |

Querying objects in a different namespace:

3.0

CIM:

```
> Get-CimInstance -Namespace "root/SecurityCenter2" -ClassName AntiVirusProduct
```

```
displayName            : Windows Defender
instanceGuid           : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState           : 397568
timestamp              : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName         :
```

WMI:

```
> Get-WmiObject -Namespace "root\SecurityCenter2" -Class AntiVirusProduct

__GENUS                : 2
__CLASS                 : AntiVirusProduct
__SUPERCLASS           :
__DYNASTY               : AntiVirusProduct
__RELPATH               : AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-
DA132C1ACF46}"
__PROPERTY_COUNT       : 6
__DERIVATION            : {}
__SERVER                : STACKOVERFLOW-PC
__NAMESPACE             : ROOT\SecurityCenter2
__PATH                  : \\STACKOVERFLOW-
PC\ROOT\SecurityCenter2:AntiVirusProduct.instanceGuid="{D68DDC3A-831F-4fae-9E44-DA132C1ACF46}"
displayName             : Windows Defender
instanceGuid            : {D68DDC3A-831F-4fae-9E44-DA132C1ACF46}
pathToSignedProductExe : %ProgramFiles%\Windows Defender\MSASCui.exe
pathToSignedReportingExe : %ProgramFiles%\Windows Defender\MsMpeng.exe
productState            : 397568
timestamp                : Fri, 09 Sep 2016 21:26:41 GMT
PSComputerName          : STACKOVERFLOW-PC
```

Classes and namespaces

There are many classes available in CIM and WMI which are separated into multiple namespaces. The most common (and default) namespace in Windows is `root/cimv2`. To find the right class, it can be useful to list all or search.

List available classes

You can list all available classes in the default namespace (`root/cimv2`) on a computer.

3.0

CIM:

```
Get-CimClass
```

WMI:

```
Get-WmiObject -List
```

Search for a class

You can search for specific classes using wildcards. Ex: Find classes containing the word `process`.

3.0

CIM:

```
> Get-CimClass -ClassName "*Process*"

    Namespace: ROOT/CIMV2

CimClassName                CimClassMethods          CimClassProperties
-----
Win32_ProcessTrace          {}                        {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStartTrace     {}                        {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
Win32_ProcessStopTrace      {}                        {SECURITY_DESCRIPTOR, TIME_CREATED,
ParentProcessID, ProcessID...}
CIM_Process                  {}                        {Caption, Description, InstallDate,
Name...}
Win32_Process                {Create, Terminat... {Caption, Description, InstallDate,
Name...}
CIM_Processor                {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
Win32_Processor              {SetPowerState, R... {Caption, Description, InstallDate,
Name...}
...
```

WMI:

```
Get-WmiObject -List -Class "*Process*"
```

List classes in a different namespace

The root namespace is simply called `root`. You can list classes in another namespace using the `Namespace` parameter.

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"
```

List available namespaces

To find available child-namespaces of `root` (or another namespace), query the objects in the `__NAMESPACE`-class for that namespace.

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"
```

Read WMI and CIM online: <https://riptutorial.com/powershell/topic/6808/wmi-and-cim>

Chapter 72: Working with Objects

Examples

Updating Objects

Adding properties

If you'd like to add properties to an existing object, you can use the `Add-Member` cmdlet. With `PSObjects`, values are kept in a type of "Note Properties"

```
$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
```

You can also add properties with `Select-Object` Cmdlet (so called calculated properties):

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

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

The command above can be shortened to this:

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

Removing properties

You can use the `Select-Object` Cmdlet to remove properties from an object:

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

# Returns
PS> $object
```

```
Name SomeNewProp SomeOtherProp
---- -
nem A value Another value
```

Creating a new object

PowerShell, unlike some other scripting languages, sends objects through the pipeline. What this means is that when you send data from one command to another, it's essential to be able to create, modify, and collect objects.

Creating an object is simple. Most objects you create will be custom objects in PowerShell, and the type to use for that is `PSObject`. PowerShell will also allow you to create any object you could create in .NET.

Here's an example of creating a new objects with a few properties:

Option 1: New-Object

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

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

You can store the object in a variable by prefacing the command with `$newObject =`

You may also need to store collections of objects. This can be done by creating an empty collection variable, and adding objects to the collection, like so:

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

You may then wish to iterate through this collection object by object. To do that, locate the Loop section in the documentation.

Option 2: Select-Object

A less common way of creating objects that you'll still find on the internet is the following:

```

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

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

```

Option 3: pscustomobject type accelerator (PSv3+ required)

The ordered type accelerator forces PowerShell to keep our properties in the order that we defined them. You don't need the ordered type accelerator to use `[PSCustomObject]`:

```

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

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

```

Examining an object

Now that you have an object, it might be good to figure out what it is. You can use the `Get-Member` cmdlet to see what an object is and what it contains:

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

This yields:

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

Followed by a list of properties and methods the object has.

Another way to get the type of an object is to use the `GetType` method, like so :

```

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

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

```

To view a list of properties the object has, along with their values, you can use the Format-List cmdlet with its Property parameter set to : * (meaning all).

Here is a example, with the resulting output :

```
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 |

Creating Instances of Generic Classes

Note: examples written for PowerShell 5.1 You can create instances of Generic Classes

```
#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
```

Gives the 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

```

Generic Collections are also possible

```

[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

```

Gives the 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

```

Read Working with Objects online: <https://riptutorial.com/powershell/topic/1328/working-with-objects>

Chapter 73: Working with the PowerShell pipeline

Introduction

PowerShell introduces an object pipelining model, which allows you to send whole objects down through the pipeline to consuming commandlets or (at least) the output. In contrast to classical string-based pipelining, information in piped objects don't have to be on specific positions. Commandlets can declare to interact with Objects from the pipeline as input, while return values are sent to the pipeline automatically.

Syntax

- **BEGIN** The first block. Executed once at the beginning. The pipeline input here is `$null`, as it has not been set.
- **PROCESS** The second block. Executed for each element of the pipeline. The pipeline parameter is equal to the currently processed element.
- **END** Last block. Executed once at the end. The pipeline parameter is equal to the last element of the input, because it has not been changed since it was set.

Remarks

In most cases, the input of the pipeline will be an array of objects. Although the behavior of the `PROCESS{}` block may seem similar to the `foreach{}` block, skipping an element in the array requires a different process.

If, like in `foreach{}`, you used `continue` inside the `PROCESS{}` block, it would break the pipeline, skipping all following statements including the `END{}` block. Instead, use `return` - it will only end the `PROCESS{}` block for the current element and move to the next.

In some cases, there is a need to output the result of functions with different encoding. The encoding of the output of the CmdLets is controlled by the `$OutputEncoding` variable. When the output is intended to be put into a pipeline to native applications, it might be a good idea to fix the encoding to match the target `$OutputEncoding = [Console]::OutputEncoding`

Additional references:

Blog article with more insight about `$OutputEncoding`
<https://blogs.msdn.microsoft.com/powershell/2006/12/11/outputencoding-to-the-rescue/>

Examples

Writing Functions with Advanced Lifecycle

This example shows how a function can accept pipelined input, and iterate efficiently.

Note, that the `begin` and `end` structures of the function are optional when pipelining, but that `process` is required when using `ValueFromPipeline` OR `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
```

Output:

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

Basic Pipeline Support in Functions

This is an example of a function with the simplest possible support for pipelining.

Any function with pipeline support must have at least one parameter with the `ParameterAttribute` `ValueFromPipeline` OR `ValueFromPipelineByPropertyName` set, as shown below.

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

$foo = 'Hello World!'

$foo | Write-FromPipeline
```

Output:

Hello World!

Note: In PowerShell 3.0 and above, Default Values for ParameterAttributes is supported. In earlier versions, you must specify `ValueFromPipeline=$true`.

Working concept of pipeline

In a pipeline series each function runs parallel to the others, like parallel threads. The first processed object is transmitted to the next pipeline and the next processing is immediately executed in another thread. This explains the high speed gain compared to the standard `ForEach`

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

1. step - copy the first file (in `Copy-file` Thread)
2. step - copy second file (in `Copy-file` Thread) and simultaneously Encrypt the first (in `Encrypt-File`)
3. step - copy third file (in `Copy-file` Thread) and simultaneously encrypt second file (in `Encrypt-File`) and simultaneously `get-Md5` of the first (in `Get-Md5`)

Read Working with the PowerShell pipeline online:

<https://riptutorial.com/powershell/topic/3937/working-with-the-powershell-pipeline>

Chapter 74: Working with XML Files

Examples

Accessing an XML File

```
<!-- 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>
```

Loading an XML File

To load an XML file, you can use any of these:

```
# 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")
```

Accessing XML as Objects

```
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

```

Accessing XML with 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

```

Accessing XML containing namespaces with 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 |

Creating an XML Document using 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()
```

Output XML File

```
<?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>
```

Adding snippets of XML to current XMLDocument

Sample Data

XML Document

First, let's define a sample XML document named "**books.xml**" in our current directory:

```
<?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>
      <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>

```

New Data

What we want to do is add a few new books to this document, let's say *Patriot Games* by Tom Clancy (yes, I'm a fan of Clancy's works ^__^) and a Sci-Fi favourite: *The Hitchhiker's Guide to the Galaxy* by Douglas Adams mainly because Zaphod Beeblebrox is just fun to read.

Somehow we've acquired the data for the new books and saved them as a list of `PSCustomObjects`:

```

$newBooks = @(
    [PSCustomObject] @{
        "Title" = "Patriot Games";
        "Author" = "Tom Clancy";
        "PageCount" = 540;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-39-913241-4";
                "Year" = "1987";
                "First" = $True;
                "Name" = "Putnam";
                "Binding" = "Hardcover";
            }
        );
        "Characters" = @(
            "Jack Ryan", "Prince of Wales", "Princess of Wales",
            "Robby Jackson", "Cathy Ryan", "Sean Patrick Miller"
        );
        "film" = $True;
    },
    [PSCustomObject] @{
        "Title" = "The Hitchhiker's Guide to the Galaxy";
        "Author" = "Douglas Adams";
        "PageCount" = 216;
        "Publishers" = @(
            [PSCustomObject] @{
                "ISBN" = "978-0-33-025864-7";
                "Year" = "1979";
                "First" = $True;
                "Name" = "Pan Books";
                "Binding" = "Hardcover";
            }
        );
    }
)

```

```

    }
  );
  "Characters" = @(
    "Arthur Dent", "Marvin", "Zaphod Beeblebrox", "Ford Prefect",
    "Trillian", "Slartibartfast", "Dirk Gently"
  );
  "film" = $True;
}
);

```

Templates

Now we need to define a few skeleton XML structures for our new data to go into. Basically, you want to create a skeleton/template for each list of data. In our example, that means we need a template for the book, characters, and publishers. We can also use this to define a few default values, such as the value for the `film` tag.

```

$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="" />
"@;

```

We're done with set-up.

Adding the new data

Now that we're all set-up with our sample data, let's add the custom objects to the XML Document Object.

```

# 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

```

We can now write our XML to disk, or screen, or web, or wherever!

Profit

While this may not be the procedure for everyone I found it to help avoid a whole bunch of

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

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

I think the method detailed in the example is cleaner and easier to parse for normal humans.

Improvements

It may be possible to change the template to include elements with children instead of breaking out each section as a separate template. You just have to take care to clone the previous element when you loop through the list.

Read [Working with XML Files](https://riptutorial.com/powershell/topic/4882/working-with-xml-files) online: <https://riptutorial.com/powershell/topic/4882/working-with-xml-files>

Credits

| S. No | Chapters | Contributors |
|-------|---|---|
| 1 | Getting started with 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 , oswaj , Poorkenny , Sam Martin , th1rdey3 , TheIncorrigible1 , Tim , tjrobinson , TravisEz13 , vonPryz , Xalorous |
| 2 | ActiveDirectory module | Lachie White |
| 3 | Aliases | jumbo |
| 4 | Amazon Web Services (AWS) Rekognition | Trevor Sullivan |
| 5 | Amazon Web Services (AWS) Simple Storage Service (S3) | Trevor Sullivan |
| 6 | Anonymize IP (v4 and v6) in text file with Powershell | NooJ |
| 7 | Archive Module | James Ruskin , RapidCoder |
| 8 | Automatic Variables | Brant Bobby , jumbo , Mateusz Piotrowski , Moerwald , Ranadip Dutta , Roman |
| 9 | Automatic Variables - part 2 | Roman |
| 10 | Basic Set Operations | Euro Micelli , Ranadip Dutta , TravisEz13 |
| 11 | Built-in variables | Trevor Sullivan |
| 12 | Calculated Properties | Prageeth Saravanan |
| 13 | Cmdlet Naming | TravisEz13 |
| 14 | Comment-based | Christophe |

| | | |
|----|---|--|
| | help | |
| 15 | Common parameters | autosvet , jumbo , RamenChef |
| 16 | Communicating with RESTful APIs | autosvet , Clijsters , HAL9256 , kdtong , RamenChef , Ranadip Dutta , Sam Martin , YChi Lu |
| 17 | Conditional logic | Liam , lloyd , miken32 , TravisEz13 |
| 18 | Creating DSC Class-Based Resources | Trevor Sullivan |
| 19 | CSV parsing | Andrei Epure , Frode F. |
| 20 | Desired State Configuration | autosvet , CmdrTchort , Frode F. , RamenChef |
| 21 | Embedding Managed Code (C# VB) | ajb101 |
| 22 | Enforcing script prerequisites | autosvet , Frode F. , jumbo , RamenChef |
| 23 | Environment Variables | autosvet |
| 24 | Error handling | Prageeth Saravanan |
| 25 | GUI in Powershell | Sam Martin |
| 26 | Handling Secrets and Credentials | 4444 , briantist , Ranadip Dutta , TravisEz13 |
| 27 | HashTables | Florian Meyer , Ranadip Dutta , TravisEz13 |
| 28 | How to download latest artifact from Artifactory using Powershell script (v2.0 or below)? | ANIL |
| 29 | Infrastructure Automation | Giulio Caccin , Ranadip Dutta |
| 30 | Introduction to Pester | Frode F. , Sam Martin |
| 31 | Introduction to Psake | Roman |
| 32 | ISE module | Florian Meyer |

| | | |
|----|--|---|
| 33 | 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 |
| 34 | Modules, Scripts and Functions | Frode F. , Ranadip Dutta , Xalorous |
| 35 | MongoDB | Thomas Gerot , Zteffer |
| 36 | Naming Conventions | niksofteng |
| 37 | Operators | Anthony Neace , Bevo , Clijsters , Gordon Bell , JPBlanc , Mark Wragg , Ranadip Dutta |
| 38 | Package management | TravisEz13 |
| 39 | Parameter sets | Bert Levrau , Poorkenny |
| 40 | PowerShell "Streams"; Debug, Verbose, Warning, Error, Output and Information | DarkLite1 , Dave Anderson , megamorf |
| 41 | PowerShell Background Jobs | Clijsters , mattnicola , Ranadip Dutta , Richard , TravisEz13 |
| 42 | PowerShell Classes | boeproxx , Brant Bobby , Frode F. , Jaqueline Vanek , Mert Gülsoy , Ranadip Dutta , xvorsx |
| 43 | PowerShell Dynamic Parameters | Poorkenny |
| 44 | PowerShell Functions | Bert Levrau , Eris , James Ruskin , Luke Ryan , niksofteng , Ranadip Dutta , Richard , TessellatingHeckler , TravisEz13 , Xalorous |
| 45 | Powershell Modules | autosvet , Mike Shepard , TravisEz13 , Trevor Sullivan |
| 46 | Powershell profiles | Frode F. , Kolob Canyon |
| 47 | Powershell Remoting | Avshalom , megamorf , Moerwald , Sam Martin , ShaneC |
| 48 | powershell sql queries | Venkatakrisshnan |
| 49 | PowerShell | Trevor Sullivan |

| | | |
|----|---|--|
| | Workflows | |
| 50 | PowerShell.exe Command-Line | Frode F. |
| 51 | PSScriptAnalyzer - PowerShell Script Analyzer | Mark Wragg , mattnicola |
| 52 | Regular Expressions | Frode F. |
| 53 | Return behavior in PowerShell | Bert Levrau , camilohe , Eris , jumbo , Ranadip Dutta , Thomas Gerot |
| 54 | Running Executables | RamenChef , W1M0R |
| 55 | Scheduled tasks module | Sam Martin |
| 56 | Security and Cryptography | YChi Lu |
| 57 | Sending Email | Adam M. , jimmyb , megamorf , NooJ , Ranadip Dutta , void , Yusuke Arakawa |
| 58 | SharePoint Module | Raziel |
| 59 | Signing Scripts | AP. , Frode F. |
| 60 | Special Operators | TravisEz13 |
| 61 | Splatting | autosvet , Frode F. , Moerwald , Petru Zaharia , Poorkenny , RamenChef , Ranadip Dutta , TravisEz13 , xXhRQ8sD2L7Z |
| 62 | Strings | Frode F. , restless1987 , void |
| 63 | Switch statement | Anthony Neace , Frode F. , jumbo , oɔwəɹ , Ranadip Dutta , TravisEz13 |
| 64 | TCP Communication with PowerShell | autosvet , RamenChef , Richard |
| 65 | URL Encode/Decode | VertigoRay |
| 66 | Using existing static classes | Austin T French , briantist , motcke , Ranadip Dutta , Xenophane |
| 67 | Using ShouldProcess | Brant Bobby , Charlie Joynt , Schwarzie2478 |
| 68 | Using the Help | Frode F. , Madniz , mattnicola , RamenChef |

| | System | |
|----|--------------------------------------|--|
| 69 | Using the progress bar | Clijsters , jumbo , Ranadip Dutta |
| 70 | Variables in PowerShell | autosvet , Eris , Liam , Prageeth Saravanan , Ranadip Dutta , restless1987 , Steve K |
| 71 | WMI and CIM | Frode F. |
| 72 | Working with Objects | Chris N , djwork , Mathieu Buisson , megamorf |
| 73 | Working with the PowerShell pipeline | Alban , Atsch , Clijsters , Deptor , James Ruskin , Keith , o3w0j , Sam Martin |
| 74 | Working with XML Files | autosvet , Frode F. , Giorgio Gambino , Lieven Keersmaekers , RamenChef , Richard , Rowshi |