PowerShell 101

The No-Nonsense Guide to Windows PowerShell

Mike F. Robbins

PowerShell 101

The No-Nonsense Guide to Windows PowerShell

Mike F. Robbins

This book is for sale at http://leanpub.com/powershell101

This version was published on 2024-03-04

Also By Mike F. Robbins

The PowerShell Conference Book

Contents

Preface

This is the story of my journey of how I went from being a dangerous script copy and paster to a Microsoft MVP and now the lead technical writer for Azure PowerShell at Microsoft.

In 2007, I was a senior-level engineer providing technical support to three companies that were among the early adopters of Exchange Server 2007. While working on their Exchange Servers, I often needed to make configuration changes and perform administrative tasks that I couldn't accomplish using the Graphical User Interface (GUI). So, I inadvertently became a dangerous script copy and paster.

Intending to learn PowerShell, I purchased a copy of the *PowerShell 1.0 Step-by-Step* book by Ed Wilson, The Scripting Guy (retired). Looking back, I realize it's the book I should have read, but I thought I never had the time.

> Good intentions are worthless without action.

At the rate I was going, I would have never learned PowerShell because I could only spend a little time trying to understand it at work.

Most people with a family, including me, value work-life balance and try never to miss out on family activities. However, I meet some people who are too busy because of family responsibilities and always seem to know what happened in recent television shows.

When someone says they don't have time for something, they really mean that it's not important enough for them to make time for it. This thought process reminds me of the funny pictures I've seen where Neanderthals are too busy pushing a cart around with square wheels to make time to learn about round wheels. Being too busy pointing and clicking in the GUI instead of making time to learn PowerShell is inefficient, like the Neanderthal analogy. Instead of being funny, though, it's just sad.

I thought I was too busy to learn PowerShell. Are you too busy? If so, I challenge you to make time to learn PowerShell. I've never met anyone who regretted learning PowerShell.

My employer's training budget was limited. I understood the importance of keeping my skills up to date and that my career was my responsibility, not my employer's. In addition to the few company-sponsored conferences I attended, I used free and low-cost resources like books, videos, podcasts, and blog articles to learn PowerShell. I also attended free technology events, including user group meetings, which allowed me to learn from and network with others in the industry. Many of these events are streamed online and recorded. To maximize my time, I often multitask by reading books or watching training videos while walking on the treadmill at the gym and listening to podcasts during my commute.

We're all in control of our own careers. A job and a career are two different things. A job is what we do to pay the bills. A career is what we do to advance our skills and knowledge.

When Microsoft released PowerShell 2.0 with remoting, it empowered automation in on-premises data centers. That was the pivotal moment when I decided to make time to learn PowerShell.

I dedicated an enormous amount of my personal time to learning PowerShell, often staying up until the wee hours of the morning while still having to be at work early. I not only learned PowerShell, but I also became proficient with it.

> Anything worth learning is worth learning well.

I competed in the beginner category of the Scripting Games in 2012, the last time Ed Wilson hosted them. I led the competition during most events and finished in third place. I competed in the advanced category in the 2013 Scripting Games, the first year that PowerShell.org[1] hosted them, and I won.

Once I figure something out, I help empower others by sharing my knowledge. In 2014, Microsoft awarded me their MVP award for my activities in the PowerShell community. I was re-awarded every year until 2020, after I became the lead technical writer for Azure PowerShell at Microsoft.

- *Mike F. Robbins*

[1] https://powershell.org/

About this book

Before PowerShell, I began my career as an IT Pro, pointing and clicking in the GUI. I wrote this book to save IT Pros from themselves by reducing the learning curve and helping them avoid being reluctant to learn PowerShell.

Instead of a book that covers topics with fictitious scenarios, this book is a condensed version targeting the specific topics I've found an IT Pro needs to know to succeed with PowerShell in a real-world production environment. It's a collection of what I wish someone would have told me when I started learning PowerShell, along with the tips, tricks, and best practices that I've learned while using PowerShell since 2007.

In every chapter, you'll find a curated collection of links to specific help topics if you want to know more about the information covered in that chapter. These resources help you dive deeper into the topics discussed in the book and broaden your understanding of PowerShell.

Who is this book for?

This book is for anyone wanting to learn PowerShell. Whether you're a beginner or an experienced user, this book will help you improve your PowerShell skills.

This book focuses on Windows PowerShell version 5.1 running on Windows 11 and Windows Server 2022 in a Microsoft Active Directory domain environment. However, the basic concepts apply to all versions of PowerShell running on any supported platform.

The mission of this book

The mission of this book is to help bootstrap others into the industry by donating all (100%) of the royalties from the sales of this book on Leanpub to the OnRamp

scholarship program (beginning June 3rd, 2020, and later). For more information about OnRamp scholarships, see powershellsummit.org/onramp/[1].

About the author

Mike F. Robbins, a former Microsoft MVP, is the lead technical writer for Azure PowerShell[2] at Microsoft. With extensive experience in PowerShell, he is a scripting, automation, and efficiency expert. As a lifelong learner, Mike continuously strives to improve his skills and empower others by sharing his knowledge and experience. He is also a published author, having written several books, including:

- Author of PowerShell 101: The No-Nonsense Guide to Windows PowerShell[3]
- Creator of The PowerShell Conference Book[4]
- Co-author of Windows PowerShell TFM 4th Edition[5]
- Contributing author in the PowerShell Deep Dives[6] book

When Mike's not writing documentation for Microsoft, he can be found sharing his thoughts and insights on his blog at mikefrobbins.com[7] and interacting with his followers on Twitter @mikefrobbins[8].

About the technical editor

Tommy Maynard is a Senior Systems Administrator with a passion for PowerShell. He has over 15 years of experience in Information Technology and finally feels he's found his calling. Luckily for him, PowerShell works right alongside the technologies he has long supported. He aims to help educate and inspire people in his industry to embrace PowerShell, scripting, and automation. Tommy blogs at tommymaynard.com[9] and can be found on Twitter @thetommymaynard[10].

[1] https://www.powershellsummit.org/onramp/
[2] https://aka.ms/azps
[3] https://leanpub.com/powershell101
[4] https://leanpub.com/powershell-conference-book
[5] https://www.sapien.com/books_training/Windows-PowerShell-4
[6] https://www.manning.com/books/powershell-deep-dives
[7] https://mikefrobbins.com/
[8] https://twitter.com/mikefrobbins
[9] https://tommymaynard.com/
[10] https://twitter.com/thetommymaynard

Lab environment

The examples in this book were specifically created and tested on Windows 11 and Windows Server 2022 operating systems, utilizing Windows PowerShell version 5.1. If you're running a different operating system or version of PowerShell, your results may vary from the ones presented in this book.

Disclaimer

I designed this book to provide accurate information. While I've made every effort to ensure the accuracy and completeness of the information contained in this book, I assume no responsibility for errors, inaccuracies, omissions, or any inconsistencies.

The code examples provided in this book are for learning and experimentation purposes only.

You should use a lab environment to work through the code examples found in this book. I do not guarantee that the code examples will work in all circumstances and under all conditions. Before using the code examples in a production environment, you should thoroughly test them.

Furthermore, the technology, software, and coding practices discussed in this book constantly evolve, and there is no guarantee that the code and software practices used will remain relevant or appropriate.

This book does not grant you any warranty, either express or implied. You are responsible for any damage or adverse consequences that may result from using the code examples found in this book.

Chapter 1 - Getting started with PowerShell

This chapter focuses on finding and launching PowerShell and solving the initial pain points that new users experience with PowerShell. Follow along and walk through the examples in this chapter on your lab environment computer.

What is PowerShell?

Windows PowerShell is an easy-to-use command-line shell and scripting environment for automating administrative tasks of Windows-based systems.

What you need to get started with PowerShell

All modern versions of Windows operating systems ship with Windows PowerShell preinstalled.

Where to find PowerShell

The easiest way to find PowerShell on Windows 11 is to type `PowerShell` into the search bar, as shown in Figure 1-1. Notice that there are four different shortcuts for Windows PowerShell.

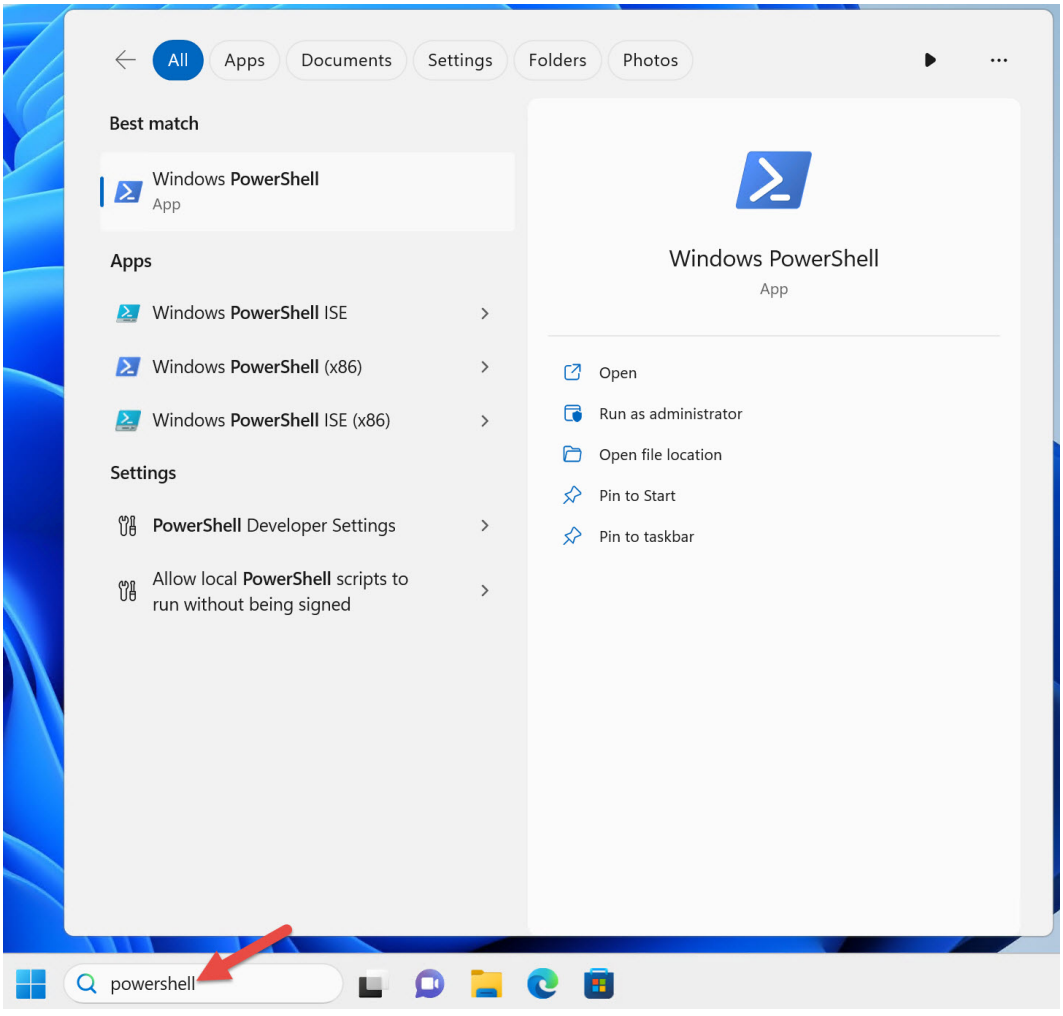

Figure 1-1 - Search for PowerShell

Windows PowerShell shortcuts on a 64-bit version of Windows:

- Windows PowerShell
- Windows PowerShell ISE
- Windows PowerShell (x86)
- Windows PowerShell ISE (x86)

On a 64-bit version of Windows, you have a 64-bit version of the Windows PowerShell console and the Windows PowerShell Integrated Scripting Environment (ISE) and a 32-bit version of each one, as indicated by the (x86) suffix on the shortcuts.

> Windows 11 only ships as a 64-bit operating system. There is no 32-bit version of Windows 11.

You only have two shortcuts if you're running an older 32-bit version of Windows. Those shortcuts don't have the (x86) suffix but are 32-bit versions.

I recommend using the 64-bit version of Windows PowerShell if you're running a 64-bit operating system unless you have a specific reason for using the 32-bit version.

Depending on what version of Windows 11 you're running, Windows PowerShell may open in Windows Terminal[1].

The PowerShell ISE is no longer in active feature development. I recommend using Visual Studio Code[2] (VS Code) with the PowerShell extension[3] to replace the ISE. You must install VS Code and the PowerShell extension because they don't ship preinstalled with Windows. You only need to install them on the computer where you create PowerShell scripts. You don't need to install them on all the computers where you run PowerShell.

For information about finding PowerShell on other versions of Windows, see Starting Windows PowerShell[4].

How to launch PowerShell

I use three different Active Directory user accounts in the production environments I support. I've mirrored those accounts in the lab environment used in this book. I log into my Windows 11 computer as a domain user without domain or local administrator rights.

[1] https://learn.microsoft.com/windows/terminal/
[2] https://code.visualstudio.com/
[3] https://code.visualstudio.com/docs/languages/powershell
[4] https://learn.microsoft.com/powershell/scripting/windows-powershell/starting-windows-powershell

Launch the PowerShell console by clicking the **Windows PowerShell** shortcut, as shown in Figure 1-1. Notice that the title bar of the Windows PowerShell console says **Windows PowerShell**, as shown in Figure 1-2.

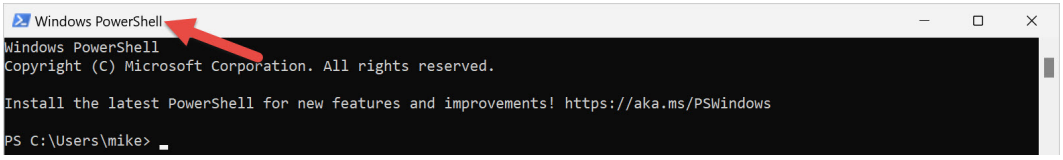

Figure 1-2 - Title bar of PowerShell window

Some commands run fine when you run PowerShell as an ordinary user. However, PowerShell doesn't participate in User Access Control (UAC). That means it's unable to prompt for elevation for tasks that require the approval of an administrator.

> UAC is a Windows security feature that helps prevent malicious code from running with elevated privileges.

When running as an ordinary user, PowerShell returns an error when you run a command that requires elevation, such as stopping a Windows service.

```
Stop-Service -Name W32Time


Stop-Service : Service 'Windows Time (W32Time)' cannot be stopped due to
the following error: Cannot open W32Time service on computer '.'.
At line:1 char:1
+ Stop-Service -Name W32Time
+ ~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : CloseError: (System.ServiceProcess.ServiceCon
   troller:ServiceController) [Stop-Service], ServiceCommandException
    + FullyQualifiedErrorId : CouldNotStopService,Microsoft.PowerShell.Comm
   ands.StopServiceCommand
```

The solution is to run PowerShell elevated as a user who is a local administrator. That's how I configured my second domain user account. Following the principle of least privilege, this account shouldn't be a domain administrator or have any elevated privileges in the domain.

Close the PowerShell console. When you relaunch it, right-click the **Windows PowerShell** shortcut and select **Run as administrator**, as shown in Figure 1-3.

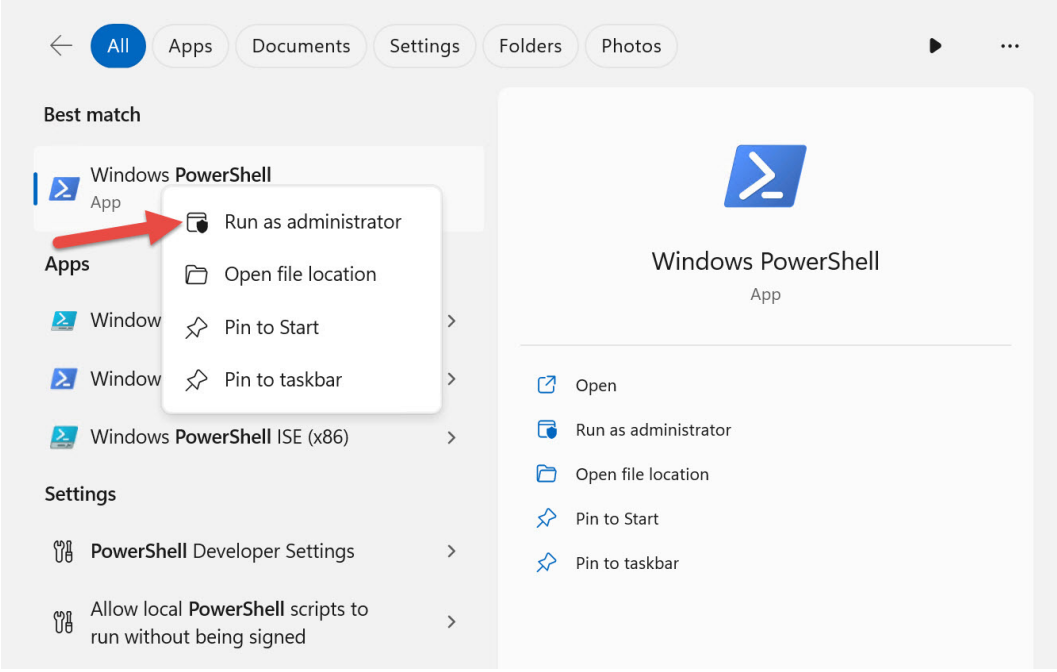

Figure 1-3 - Context menu - Run as administrator

You're prompted for credentials because you logged into Windows as an ordinary user. Enter the credentials of your domain user who is a local administrator, as shown in Figure 1-4.

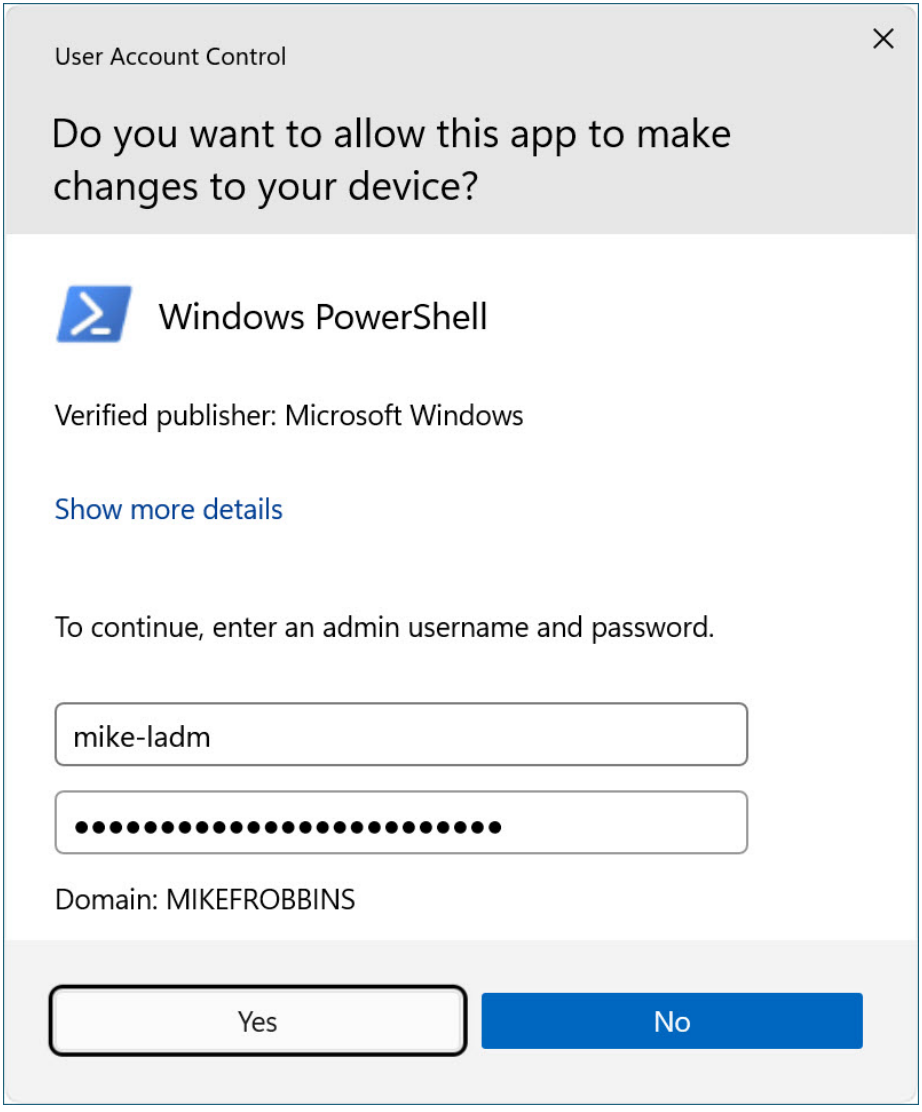

Figure 1-4 - User account control - Enter credentials

After you've relaunched the PowerShell console elevated as an administrator, the title bar says **Administrator: Windows PowerShell**, as shown in Figure 1-5.

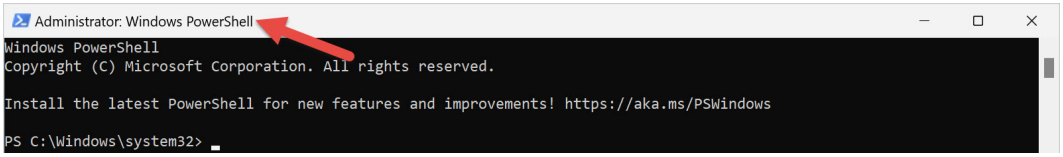

Figure 1-5 - Title bar of elevated PowerShell window

Now that you're running PowerShell elevated as an administrator, UAC is no longer a problem when you run a command that requires elevation.

> You should only run PowerShell elevated as an administrator when absolutely necessary.

When targeting remote computers, there is no need to run PowerShell elevated. Running PowerShell elevated as an administrator only affects commands that run against your local computer, not those that target remote computers.

You can simplify finding and launching PowerShell. Pin the PowerShell console or Windows Terminal shortcut to your taskbar, but don't set it to launch automatically as an administrator.

> The original version of this book, published in 2017, recommended pinning a shortcut to the taskbar to launch an elevated instance automatically every time you start PowerShell. However, this guidance is no longer recommended due to potential security implications. When you launch an application from an elevated instance of PowerShell, it also runs elevated and bypasses UAC. For example, if you launch a web browser from an elevated instance of PowerShell, any website you visit containing malicious code also runs elevated.

Search for PowerShell again, except this time right-click on it and select **Pin to taskbar** as shown in Figure 1-6.

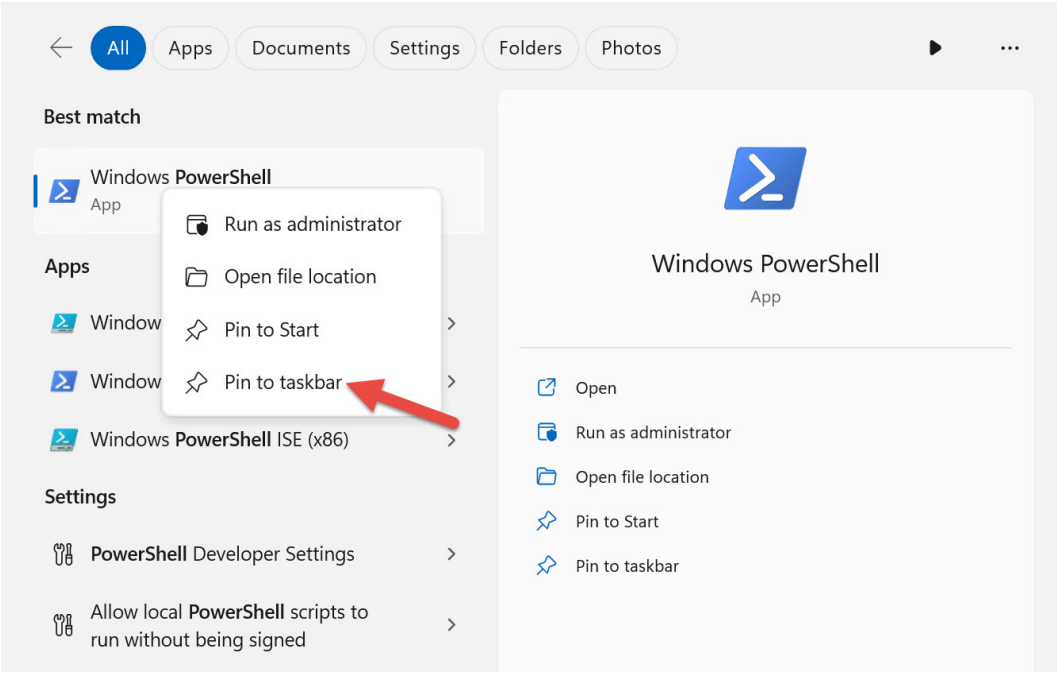

Figure 1-6 - Context menu - Pin to taskbar

When you need to run PowerShell with elevated permissions, right-click the PowerShell shortcut pinned to your taskbar while pressing *Shift* and select **Run as administrator**, as shown in Figure 1-7.

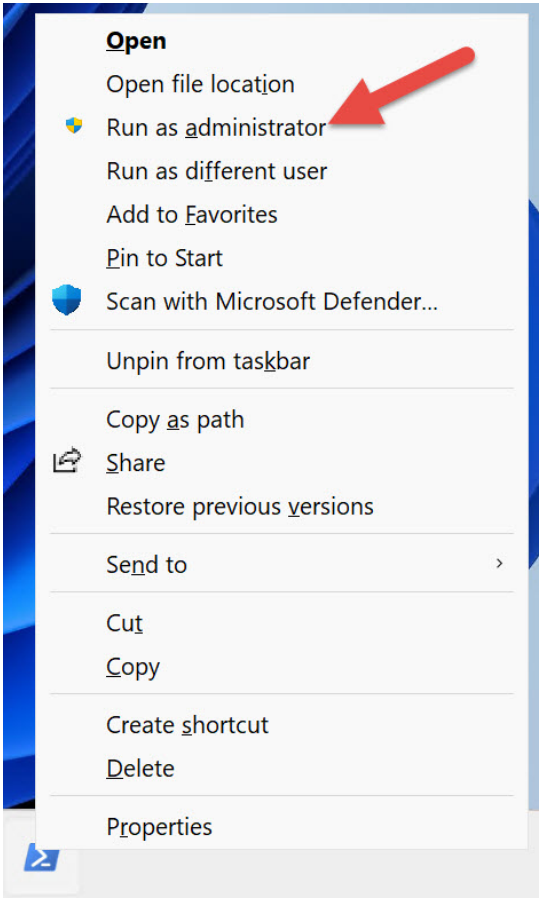

Figure 1-7 - Context menu - Run as administrator

You'll never have to worry about finding PowerShell or whether it's running elevated as an administrator again.

Determine your version of PowerShell

There are automatic variables in PowerShell that store state information. One of these variables is `$PSVersionTable`, which contains a hashtable that you can use to display the PowerShell version.

```
$PSVersionTable
```

```
Name                           Value
----                           -----
PSVersion                      5.1.22621.2428
PSEdition                      Desktop
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0...}
BuildVersion                   10.0.22621.2428
CLRVersion                     4.0.30319.42000
WSManStackVersion              3.0
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
```

If you're running a version of Windows PowerShell older than 5.1, you should update to Windows PowerShell 5.1. Microsoft distributes new versions of Windows PowerShell as part of the Windows Management Framework (WMF). Depending on the WMF version, a specific version of the .NET Framework is required. To upgrade Windows PowerShell, see Upgrading existing Windows PowerShell[5].

PowerShell version 7 isn't an upgrade to Windows PowerShell 5.1; it installs side-by-side with Windows PowerShell. Windows PowerShell version 5.1 and PowerShell version 7 are two different products. For more information about the differences between Windows PowerShell version 5.1 and PowerShell version 7, see Migrating from Windows PowerShell 5.1 to PowerShell 7[6].

> PowerShell version 6, formerly known as PowerShell Core, is no longer supported.

Execution policy

PowerShell execution policy controls the conditions under which you can run PowerShell scripts. The execution policy in PowerShell is a safety feature designed

[5] https://learn.microsoft.com/powershell/scripting/windows-powershell/install/installing-windows-powershell#upgrading-existing-windows-powershell

[6] https://learn.microsoft.com/powershell/scripting/whats-new/migrating-from-windows-powershell-51-to-powershell-7

to help prevent the unintentional execution of malicious scripts. However, it's not a security boundary because it can't stop determined users from deliberately running scripts. A determined user can bypass the execution policy in PowerShell.

You can set an execution policy for the local computer, current user, or a PowerShell session. You can also set execution policies for users and computers with Group Policy.

The following table shows the default execution policy for current Windows operating systems.

Windows Operating System Version	Default Execution Policy
Windows Server 2022	Remote Signed
Windows Server 2019	Remote Signed
Windows Server 2016	Remote Signed
Windows 11	Restricted
Windows 10	Restricted

Regardless of the execution policy setting, you can run any PowerShell command interactively. The execution policy only affects commands running in a script. Use the `Get-ExecutionPolicy` cmdlet to determine the current execution policy setting.

Check the execution policy setting on your computer.

```
Get-ExecutionPolicy
```

```
Restricted
```

List the execution policy settings for all scopes.

```
Get-ExecutionPolicy -List
```

```
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser       Undefined
 LocalMachine       Undefined
```

All Windows client operating systems have the default execution policy setting of `Restricted`. You can't run PowerShell scripts using the `Restricted` execution policy setting. To test the execution policy, save the following code as a `.ps1` file named `Get-TimeService.ps1`.

> A PowerShell script is a plaintext file with a `.ps1` extension that contains the commands you want to run. To create a PowerShell script, use a code editor like Visual Studio Code (VS Code) or any text editor such as Notepad.

```
Get-Service -Name W32Time
```

When you run the previous command interactively, it completes without error. PowerShell returns an error when you run the same command from a script.

```
.\Get-TimeService.ps1
```

Notice the error message tells you why the command failed: "*Running scripts is disabled on this system*".

```
.\Get-TimeService.ps1 : File C:\tmp\Get-TimeService.ps1 cannot be loaded
because running scripts is disabled on this system. For more information,
see about_Execution_Policies at
https:/go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\Get-TimeService.ps1
+ ~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : SecurityError: (:) [], PSSecurityException
    + FullyQualifiedErrorId : UnauthorizedAccess
```

When you run a command in PowerShell that generates an error, read the error message before retrying the command. The error message often tells you why the command failed.

You change the execution policy with the `Set-ExecutionPolicy` cmdlet. `LocalMachine` is the default scope when you don't specify the **Scope** parameter. You must run PowerShell elevated as an administrator to change the execution policy for the local machine. Unless you're signing your scripts, I recommend using the `RemoteSigned` execution policy. `RemoteSigned` requires downloaded scripts to be signed by a trusted publisher.

Before you change the execution policy, read the about_Execution_Policies[7] help topic to understand the security implications.

Change the execution policy setting on your computer to `RemoteSigned`.

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
```

Read the warning that's displayed when you change the execution policy.

[7] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_execution_policies

```
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust.
Changing the execution policy might expose you to the security risks
described in the about_Execution_Policies help topic at
https:/go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the
execution policy?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "N"):y
```

If you're not running PowerShell elevated as an administrator, you'll receive the following error message when you attempt to change the execution policy for the local machine.

```
Set-ExecutionPolicy : Access to the registry key 'HKEY_LOCAL_MACHINE\SOFTWAR
E\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell' is denied. To
change the execution policy for the default (LocalMachine) scope, start
Windows PowerShell with the "Run as administrator" option. To change the
execution policy for the current user, run "Set-ExecutionPolicy -Scope
CurrentUser".
At line:1 char:1
+ Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : PermissionDenied: (:) [Set-ExecutionPolicy],
   UnauthorizedAccessException
    + FullyQualifiedErrorId : System.UnauthorizedAccessException,Microsoft.
   PowerShell.Commands.SetExecutionPolicyCommand
```

It's also possible to change the execution policy for the current user without requiring you to run PowerShell elevated as an administrator.

Set the execution policy for the current user to `RemoteSigned`. This step is unnecessary if you successfully set the execution policy for the local machine to `RemoteSigned`.

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Now that you've changed the execution policy to `RemoteSigned`, the `Get-TimeService.ps1` script runs successfully.

```
.\Get-TimeService.ps1


Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Windows Time
```

Summary

In this chapter, you've learned where to find and how to launch PowerShell. You've also learned how to determine the version of PowerShell and the purpose of execution policies.

Review

1. How do you determine what PowerShell version a computer is running?
2. When should you launch PowerShell elevated as an administrator?
3. What's the default execution policy on Windows client computers, and what does it prevent you from doing?
4. How do you determine the current PowerShell execution policy setting?
5. How do you change the PowerShell execution policy?

References

If you're interested in learning more about the topics covered in this chapter, you should read the following PowerShell help topics.

- about_Automatic_Variables[8]
- about_Execution_Policies[9]
- about_Hash_Tables[10]

[8] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_automatic_variables
[9] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_execution_policies
[10] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_hash_tables

Next steps

In the next chapter, you'll learn about the discoverability of commands in PowerShell. You'll also learn how to download PowerShell's help topics to view offline.

Chapter 2 - The help system

In an experiment designed to assess proficiency in PowerShell, two distinct groups of IT professionals —beginners and experts— were first given a written examination without access to a computer. Surprisingly, the test scores indicated comparable skills across both groups. A subsequent test was then administered, mirroring the first but with one key difference: participants had access to an offline computer equipped with PowerShell. The results revealed a significant skills gap between the two groups this time.

What factors contributed to the outcomes observed between the two assessments?

> Experts don't always know the answers, but they know how to figure out the answers.

The outcomes observed in the results of the two tests were because experts don't memorize thousands of PowerShell commands. Instead, they excel at using the help system within PowerShell, enabling them to discover and learn how to use commands when necessary.

> Mastering the help system is the key to being successful with PowerShell.

I've heard Jeffrey Snover, the creator of PowerShell, share a similar story on multiple occasions.

Discoverability

Compiled commands in PowerShell are known as cmdlets, pronounced as *"command-let"*, not *"CMD-let"*. The naming convention for cmdlets follows a singular **Verb-Noun** format to make them easily discoverable. For instance, `Get-Process` is the cmdlet to determine what processes are running, and `Get-Service` is the cmdlet to

retrieve a list of services. Other types of commands in PowerShell, such as aliases and functions, are discussed later in this book. The term *"PowerShell command"* describes any command in PowerShell, regardless of whether it's a cmdlet, function, or alias.

The three core cmdlets in PowerShell

- `Get-Help`
- `Get-Command`
- `Get-Member` (covered in chapter 3)

I'm often asked: *"How do you figure out what the commands are in PowerShell?"*. Both `Get-Help` and `Get-Command` are invaluable resources for discovering and understanding commands in PowerShell.

Get-Help

The first thing you need to know about the help system in PowerShell is how to use the `Get-Help` cmdlet.

`Get-Help` is a multipurpose command that helps you learn how to use commands once you find them. You can also use `Get-Help` to locate commands, but in a different and more indirect way when compared to `Get-Command`.

When using `Get-Help` to locate commands, it initially performs a wildcard search for command names based on your input. If that doesn't find any matches, it conducts a comprehensive full-text search across all PowerShell help topics on your system. If that also fails to find any results, it returns an error.

Here's how to use `Get-Help` to view the help content for the `Get-Help` cmdlet.

```
Get-Help -Name Get-Help
```

Beginning with PowerShell version 3.0, the help content doesn't ship pre-installed with the operating system. When you run `Get-Help` for the first time, a message asks if you want to download the PowerShell help files to your computer.

Answering **Yes** by pressing *Y* executes the `Update-Help` cmdlet, downloading the help content.

```
Do you want to run Update-Help?
The Update-Help cmdlet downloads the most current Help files for Windows
PowerShell modules, and installs them on your computer. For more information
about the Update-Help cmdlet, see
https:/go.microsoft.com/fwlink/?LinkId=210614.
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

You only receive this message if you run PowerShell elevated as an administrator. You also don't receive it if you're using the `help` function or `man` alias. If you don't receive this message, run `Update-Help` from an elevated PowerShell session running as an administrator.

Once the update is complete, the help topic is displayed.

Take a moment to run the example on your computer, review the output, and observe how the help system organizes the information.

- NAME
- SYNOPSIS
- SYNTAX
- DESCRIPTION
- RELATED LINKS
- REMARKS

As you review the output, keep in mind that help topics often contain a vast amount of information, and what you see by default isn't the entire help topic.

Parameters

When you run a command in PowerShell, you may need to provide additional information or input to the command. Parameters allow you to specify options and arguments that change a command's behavior. The **SYNTAX** section of each help topic outlines a command's available parameters.

`Get-Help` has many parameters that you can specify to return the entire help topic or a subset for a command. To view all the available parameters for `Get-Help`, see the **SYNTAX** section of its help topic, as shown in the following example.

```
...
SYNTAX
    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
    Class | Configuration}] [-Component <System.String[]>] [-Full]
    [-Functionality <System.String[]>] [-Path <System.String>] [-Role
    <System.String[]>] [<CommonParameters>]

    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
    Class | Configuration}] [-Component <System.String[]>] -Detailed
    [-Functionality <System.String[]>] [-Path <System.String>] [-Role
    <System.String[]>] [<CommonParameters>]

    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
    Class | Configuration}] [-Component <System.String[]>] -Examples
    [-Functionality <System.String[]>] [-Path <System.String>] [-Role
    <System.String[]>] [<CommonParameters>]

    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
    Class | Configuration}] [-Component <System.String[]>] [-Functionality
    <System.String[]>] -Online [-Path <System.String>] [-Role
    <System.String[]>] [<CommonParameters>]

    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
    Class | Configuration}] [-Component <System.String[]>] [-Functionality
    <System.String[]>] -Parameter <System.String> [-Path <System.String>]
    [-Role <System.String[]>] [<CommonParameters>]

    Get-Help [[-Name] <System.String>] [-Category {Alias | Cmdlet | Provider
    | General | FAQ | Glossary | HelpFile | ScriptCommand | Function |
    Filter | ExternalScript | All | DefaultHelp | Workflow | DscResource |
```

```
    Class | Configuration}] [-Component <System.String[]>] [-Functionality
    <System.String[]>] [-Path <System.String>] [-Role <System.String[]>]
    -ShowWindow [<CommonParameters>]
...
```

Parameter sets

When you review the **SYNTAX** section, you'll notice that information appears to be repeated six times. Each of those blocks is an individual parameter set, indicating the `Get-Help` cmdlet features six distinct sets of parameters. A closer look reveals each parameter set contains at least one unique parameter, making it different from the others.

Parameter sets are mutually exclusive. Once you specify a unique parameter that only exists in one parameter set, you're limited to using the parameters contained within that parameter set. For instance, you can't use the **Full** and **Detailed** parameters of `Get-Help` together because they belong to different parameter sets.

Each of the following parameters belongs to a different parameter set for the `Get-Help` cmdlet.

- Full
- Detailed
- Examples
- Online
- Parameter
- ShowWindow

The help syntax

If you're new to PowerShell, comprehending the cryptic help syntax —characterized by square and angle brackets— in the **SYNTAX** section of the help topics may seem overwhelming. However, learning these syntax elements is essential to becoming proficient with PowerShell. The more frequently you use the PowerShell help system, the easier it becomes to remember all the nuances.

View the syntax section of the help topic for the `Get-EventLog` cmdlet.

```
help Get-EventLog
```

The following output shows the relevant portion of the help topic.

```
...
SYNTAX
    Get-EventLog [-LogName] <System.String> [[-InstanceId]
    <System.Int64[]>] [-After <System.DateTime>] [-AsBaseObject] [-Before
    <System.DateTime>] [-ComputerName <System.String[]>] [-EntryType {Error
    | Information | FailureAudit | SuccessAudit | Warning}] [-Index
    <System.Int32[]>] [-Message <System.String>] [-Newest <System.Int32>]
    [-Source <System.String[]>] [-UserName <System.String[]>]
    [<CommonParameters>]

    Get-EventLog [-AsString] [-ComputerName <System.String[]>] [-List]
    [<CommonParameters>]
...
```

The help syntax in the **SYNTAX** section of PowerShell help topics includes pairs of square brackets (`[ ]`). These square brackets serve two different purposes depending on their usage.

- Elements enclosed within square brackets are optional.
- When an empty set of square brackets follows a datatype, such as `<string[]>`, it indicates the parameter can accept multiple values as an array or a comma-separated list.

Positional parameters

When creating a PowerShell cmdlet or function, developers can optionally designate one or more parameters as positional. Positional parameters allow you to provide a value without specifying the parameter's name. When using a parameter positionally, you must specify its value in the correct position. You can find the positional information for a parameter in the **PARAMETERS** section of a command's help topic. When you explicitly specify parameter names, their order is irrelevant.

For the `Get-EventLog` cmdlet, the first parameter in the first parameter set is **LogName**. **LogName** is enclosed in square brackets, indicating it's a positional parameter.

Since **LogName** is a positional parameter, you can specify it by either name or position. According to the angle brackets following the parameter name, the value for **LogName** must be a single string. The absence of square brackets enclosing both the parameter name and datatype indicates that **LogName** is a required parameter within this particular parameter set.

```
Get-EventLog [-LogName] <System.String>
```

The second parameter in that parameter set is **InstanceId**. Both the parameter name and datatype are entirely enclosed in square brackets, signifying that **InstanceId** is an optional parameter. Furthermore, **InstanceId** has its own pair of square brackets, indicating that it's a positional parameter similar to the **LogName** parameter. Following the datatype, an empty set of square brackets implies that **InstanceId** can accept multiple values.

```
[[-InstanceId] <System.Int64[]>]
```

Switch parameters

A parameter that doesn't require a value is called a switch parameter. You can easily identify switch parameters because a datatype doesn't follow their parameter names. When you specify a switch parameter, its value is `true`; when you don't specify a switch parameter, its value is `false`.

The second parameter set includes a **List** parameter, which is a switch parameter. When you specify the **List** parameter, it returns a list of event logs on the local computer.

```
[-List]
```

A simplified approach to syntax

There's a more user-friendly method to obtain the same information as the cryptic help syntax for some commands, except in plain English. PowerShell returns the complete help topic when using `Get-Help` with the **Full** parameter, making it easier to understand a command's usage.

```
Get-Help -Name Get-Help -Full
```

Take a moment to run the example on your computer, review the output, and observe how the help system organizes the information.

- NAME
- SYNOPSIS
- SYNTAX
- DESCRIPTION
- PARAMETERS
- INPUTS
- OUTPUTS
- NOTES
- EXAMPLES
- RELATED LINKS

By specifying the **Full** parameter with the `Get-Help` cmdlet, you'll notice the output includes several additional sections. Among these, the **PARAMETERS** section often provides a detailed explanation for each parameter. However, the extent of this information varies depending on the specific command you're investigating.

```
...
    -Detailed <System.Management.Automation.SwitchParameter>
        Adds parameter descriptions and examples to the basic help display.
        This parameter is effective only when the help files are installed
        on the computer. It has no effect on displays of conceptual ( About_
        ) help.

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

    -Examples <System.Management.Automation.SwitchParameter>
        Displays only the name, synopsis, and examples. This parameter is
        effective only when the help files are installed on the computer. It
```

```
        has no effect on displays of conceptual ( About_ ) help.

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

    -Full <System.Management.Automation.SwitchParameter>
        Displays the entire help article for a cmdlet. Full includes
        parameter descriptions and attributes, examples, input and output
        object types, and additional notes.

        This parameter is effective only when the help files are installed
        on the computer. It has no effect on displays of conceptual ( About_
        ) help.

        Required?                    false
        Position?                    named
        Default value                False
        Accept pipeline input?       False
        Accept wildcard characters?  false
...
```

When you ran the previous command to display the help topic for `Get-Help`, you probably noticed the output scrolled by too quickly to read it.

If you're using the PowerShell console, Windows Terminal, or VS Code and need to view a help topic, the `help` function can be useful. It pipes `Get-Help` to `more.com`, displaying one page of help content at a time. As for the ISE, running `help` works the same way as `Get-Help`. I recommend using the `help` function instead of the `Get-Help` cmdlet because it provides a better user experience and it's less to type.

While typing less may seem beneficial, it's not always the best practice or intuitive when saving commands in a script or sharing code with others. Using full cmdlet and parameter names offers the benefit of self-documenting, making the code more easily interpreted and understandable for anyone who reviews it.

Run each of the following commands in PowerShell on your computer.

```
Get-Help -Name Get-Help -Full
help -Name Get-Help -Full
help Get-Help -Full
```

Did you observe any variations in the output when you ran the previous commands?

The only difference is the last two commands display their output one page at a time. When using the `help` function, press the *Spacebar* to navigate to the next page of content or `Q` to quit. If you need to terminate any command running interactively in PowerShell, press *Ctrl+C*.

In the previous example, the first line uses the `Get-Help` cmdlet, the second uses the `help` function, and the third line omits the **Name** parameter while using the `help` function. Since **Name** is a positional parameter, the third example takes advantage of its placement instead of explicitly stating the parameter's name.

To quickly find information about a specific parameter, use the `help` function with the **Parameter** parameter. This approach is more concise, containing only the parameter-specific help content instead of manually scanning the entire help topic for details about a parameter.

In the following example, use the `help` function with the **Parameter** parameter to return information from the help topic for the **Name** parameter of `Get-Help`.

```
help Get-Help -Parameter Name
```

Based on the following results, the **Name** parameter is positional and must be specified in position zero when used positionally.

```
-Name <System.String>
    Gets help about the specified command or concept. Enter the name of a
    cmdlet, function, provider, script, or workflow, such as `Get-Member`,
    a conceptual article name, such as `about_Objects`, or an alias, such
    as `ls`. Wildcard characters are permitted in cmdlet and provider
    names, but you can't use wildcard characters to find the names of
    function help and script help articles.

    To get help for a script that isn't located in a path that's listed in
    the `$env:Path` environment variable, type the script's path and file
```

```
    name.

    If you enter the exact name of a help article, `Get-Help` displays the
    article contents.

    If you enter a word or word pattern that appears in several help
    article titles, `Get-Help` displays a list of the matching titles.

    If you enter any text that doesn't match any help article titles,
    `Get-Help` displays a list of articles that include that text in their
    contents.

    The names of conceptual articles, such as `about_Objects`, must be
    entered in English, even in non-English versions of PowerShell.

    Required?                    false
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByPropertyName)
    Accept wildcard characters?  true
```

The **Name** parameter expects the datatype for its value to be a single string as identified by `<String>` next to the parameter name.

There are several other parameters besides **Parameter** that you can specify with `Get-Help` to return a subset of a help topic. Run the following commands on your computer to see how they work.

```
Get-Help -Name Get-Command -Full
Get-Help -Name Get-Command -Detailed
Get-Help -Name Get-Command -Examples
Get-Help -Name Get-Command -Online
Get-Help -Name Get-Command -Parameter Noun
Get-Help -Name Get-Command -ShowWindow
```

I typically use `help <command name>` with the **Full** or **Online** parameter. If you're only interested in the examples, use the **Examples** parameter; if you're only interested in a specific parameter, use the **Parameter** parameter.

When using the **ShowWindow** parameter, it opens the help topic in a separate searchable window that can be placed on a different monitor if you have multiple monitors. However, the **ShowWindow** parameter has a known bug that may prevent it from displaying the entire help topic. The **ShowWindow** parameter also requires an operating system with a Graphical User Interface (GUI). It returns an error if you attempt to use it on Windows Server installed with the server core installation option.

If you want the help topic in a separate window and have internet access, use the **Online** parameter instead. The **Online** parameter opens the help topic with the most up-to-date content in your default web browser, allowing you to search it and navigate to other help topics.

```
help Get-Command -Online
```

Finding commands with Get-Help

To find commands with `Get-Help`, specify a search term surrounded by asterisk (*) wildcard characters for the value of the **Name** parameter. The following example uses the **Name** parameter positionally.

```
help *process*
```

```
Name                                Category  Module                    Synops
----                                --------  ------                    ------
Enter-PSHostProcess                 Cmdlet    Microsoft.PowerShell.Core Con...
Exit-PSHostProcess                  Cmdlet    Microsoft.PowerShell.Core Clo...
Get-PSHostProcessInfo               Cmdlet    Microsoft.PowerShell.Core Get...
Debug-Process                       Cmdlet    Microsoft.PowerShell.M... Deb...
Get-Process                         Cmdlet    Microsoft.PowerShell.M... Get...
Start-Process                       Cmdlet    Microsoft.PowerShell.M... Sta...
Stop-Process                        Cmdlet    Microsoft.PowerShell.M... Sto...
Wait-Process                        Cmdlet    Microsoft.PowerShell.M... Wai...
Invoke-LapsPolicyProcessing         Cmdlet    LAPS                      Inv...
ConvertTo-ProcessMitigationPolicy Cmdlet    ProcessMitigations        Con...
Get-ProcessMitigation               Cmdlet    ProcessMitigations        Get...
Set-ProcessMitigation               Cmdlet    ProcessMitigations        Set...
```

In this scenario, you aren't required to add the * wildcard characters. When you omit the wildcard characters, `Get-Help` automatically adds them behind the scenes. The following example produces the same results as specifying the * wildcard character on each end of `process`.

```
help process
```

However, you should always add them since that option works consistently. Otherwise, you're required to add them in certain scenarios and not in others. When you specify a wildcard character within the value, they're no longer automatically appended behind the scenes.

The following command doesn't return any results unless you add a * wildcard character to the beginning, end, or both the beginning and end of `pr*cess`.

```
help pr*cess
```

PowerShell generates an error if you specify a value that begins with a dash without enclosing it in quotes because it interprets it as a parameter name. No such parameter name exists for the `Get-Help` cmdlet.

```
help -process
```

If you're attempting to search for commands that end with `-process`, you must add an * to the beginning of the value.

```
help *-process
```

When you search for PowerShell commands with `Get-Help`, it's better to be vague rather than too specific.

When you searched for `process` earlier, the results only returned commands that included `process` in their name. But if you use `help` to search for `processes`, it won't find any matches for command names. As previously stated, when help doesn't find any matches, it performs a comprehensive full-text search of every help topic on your system and returns those results. This type of search often produces more results than expected, probably without the information you're trying to locate.

```
help processes


Name                              Category  Module                    Synops
----                              --------  ------                    ------
Disconnect-PSSession              Cmdlet    Microsoft.PowerShell.Core Dis...
Enter-PSHostProcess               Cmdlet    Microsoft.PowerShell.Core Con...
ForEach-Object                    Cmdlet    Microsoft.PowerShell.Core Per...
Get-PSHostProcessInfo             Cmdlet    Microsoft.PowerShell.Core Get...
Get-PSSessionConfiguration        Cmdlet    Microsoft.PowerShell.Core Get...
New-PSSessionOption               Cmdlet    Microsoft.PowerShell.Core Cre...
New-PSTransportOption             Cmdlet    Microsoft.PowerShell.Core Cre...
Out-Host                          Cmdlet    Microsoft.PowerShell.Core Sen...
Start-Job                         Cmdlet    Microsoft.PowerShell.Core Sta...
Where-Object                      Cmdlet    Microsoft.PowerShell.Core Sel...
Debug-Process                     Cmdlet    Microsoft.PowerShell.M... Deb...
Get-Process                       Cmdlet    Microsoft.PowerShell.M... Get...
Get-WmiObject                     Cmdlet    Microsoft.PowerShell.M... Get...
Start-Process                     Cmdlet    Microsoft.PowerShell.M... Sta...
Stop-Process                      Cmdlet    Microsoft.PowerShell.M... Sto...
Wait-Process                      Cmdlet    Microsoft.PowerShell.M... Wai...
Clear-Variable                    Cmdlet    Microsoft.PowerShell.U... Del...
Convert-String                    Cmdlet    Microsoft.PowerShell.U... For...
ConvertFrom-Csv                   Cmdlet    Microsoft.PowerShell.U... Con...
ConvertFrom-Json                  Cmdlet    Microsoft.PowerShell.U... Con...
ConvertTo-Html                    Cmdlet    Microsoft.PowerShell.U... Con...
ConvertTo-Xml                     Cmdlet    Microsoft.PowerShell.U... Cre...
Debug-Runspace                    Cmdlet    Microsoft.PowerShell.U... Sta...
Export-Csv                        Cmdlet    Microsoft.PowerShell.U... Con...
Export-FormatData                 Cmdlet    Microsoft.PowerShell.U... Sav...
Format-List                       Cmdlet    Microsoft.PowerShell.U... For...
Format-Table                      Cmdlet    Microsoft.PowerShell.U... For...
Get-Unique                        Cmdlet    Microsoft.PowerShell.U... Ret...
Group-Object                      Cmdlet    Microsoft.PowerShell.U... Gro...
Import-Clixml                     Cmdlet    Microsoft.PowerShell.U... Imp...
Import-Csv                        Cmdlet    Microsoft.PowerShell.U... Cre...
Measure-Object                    Cmdlet    Microsoft.PowerShell.U... Cal...
Out-File                          Cmdlet    Microsoft.PowerShell.U... Sen...
Out-GridView                      Cmdlet    Microsoft.PowerShell.U... Sen...
Select-Object                     Cmdlet    Microsoft.PowerShell.U... Sel...
```

```
Set-Variable                        Cmdlet    Microsoft.PowerShell.U... Set...
Sort-Object                         Cmdlet    Microsoft.PowerShell.U... Sor...
Tee-Object                          Cmdlet    Microsoft.PowerShell.U... Sav...
Trace-Command                       Cmdlet    Microsoft.PowerShell.U... Con...
Write-Information                   Cmdlet    Microsoft.PowerShell.U... Spe...
Export-BinaryMiLog                  Cmdlet    CimCmdlets                Cre...
Get-CimAssociatedInstance           Cmdlet    CimCmdlets                Ret...
Get-CimInstance                     Cmdlet    CimCmdlets                Get...
Import-BinaryMiLog                  Cmdlet    CimCmdlets                Use...
Invoke-CimMethod                    Cmdlet    CimCmdlets                Inv...
New-CimInstance                     Cmdlet    CimCmdlets                Cre...
Remove-CimInstance                  Cmdlet    CimCmdlets                Rem...
Set-CimInstance                     Cmdlet    CimCmdlets                Mod...
Compress-Archive                    Function  Microsoft.PowerShell.A... Cre...
Get-Counter                         Cmdlet    Microsoft.PowerShell.D... Get...
Invoke-WSManAction                  Cmdlet    Microsoft.WSMan.Manage... Inv...
Remove-WSManInstance                Cmdlet    Microsoft.WSMan.Manage... Del...
Get-WSManInstance                   Cmdlet    Microsoft.WSMan.Manage... Dis...
New-WSManInstance                   Cmdlet    Microsoft.WSMan.Manage... Cre...
Set-WSManInstance                   Cmdlet    Microsoft.WSMan.Manage... Mod...
about_Arithmetic_Operators          HelpFile
about_Arrays                        HelpFile
about_Environment_Variables         HelpFile
about_Execution_Policies            HelpFile
about_Functions                     HelpFile
about_Jobs                          HelpFile
about_Logging                       HelpFile
about_Methods                       HelpFile
about_Objects                       HelpFile
about_Pipelines                     HelpFile
about_Preference_Variables          HelpFile
about_Remote                        HelpFile
about_Remote_Jobs                   HelpFile
about_Session_Configuration_Files   HelpFile
about_Simplified_Syntax             HelpFile
about_Switch                        HelpFile
about_Variables                     HelpFile
about_Variable_Provider             HelpFile
about_Windows_Powershell_5.1        HelpFile
about_WQL                           HelpFile
```

```
about_WS-Management_Cmdlets       HelpFile
about_Foreach-Parallel            HelpFile
about_Parallel                    HelpFile
about_Sequence                    HelpFile
```

When using the `help` function to search for `process`, it returned 12 results. However, when searching for `processes`, it produced 78 results. If your search only finds one match, the help topic is displayed instead of listing the results.

```
help *hotfix*


NAME
    Get-HotFix

SYNOPSIS
    Gets the hotfixes that are installed on local or remote computers.


SYNTAX
    Get-HotFix [-ComputerName <System.String[]>] [-Credential
    <System.Management.Automation.PSCredential>] [-Description
    <System.String[]>] [<CommonParameters>]

    Get-HotFix [[-Id] <System.String[]>] [-ComputerName <System.String[]>]
    [-Credential <System.Management.Automation.PSCredential>]
    [<CommonParameters>]


DESCRIPTION
    > This cmdlet is only available on the Windows platform. The
    `Get-Hotfix` cmdlet uses the Win32_QuickFixEngineering WMI class to
    list hotfixes that are installed on the local computer or specified
    remote computers.


RELATED LINKS
    Online Version: https://learn.microsoft.com/powershell/module/microsoft.
    powershell.management/get-hotfix?view=powershell-5.1&WT.mc_id=ps-gethelp
```

```
    about_Arrays
    Add-Content
    Get-ComputerRestorePoint
    Get-Credential
    Win32_QuickFixEngineering class

REMARKS
    To see the examples, type: "get-help Get-HotFix -examples".
    For more information, type: "get-help Get-HotFix -detailed".
    For technical information, type: "get-help Get-HotFix -full".
    For online help, type: "get-help Get-HotFix -online"
```

It's not commonly known that `Get-Help` can also find commands that lack help topics. The `more` function is one of the commands that doesn't have a help topic. To confirm you can find commands with `Get-Help` that don't include help topics, use the `help` function to find `more`.

```
help *more*
```

The search only found one match, so it returned the basic syntax information you'll see when a command doesn't have a help topic.

```
NAME
    more

SYNTAX
    more [[-paths] <string[]>]

ALIASES
    None

REMARKS
    None
```

The PowerShell help system also contains conceptual **About** help topics. You must update the help content on your system for the **About** help topics to exist. If the initial update of the help system fails, the **About** help topics won't be available until

you run `Update-Help` and it completes successfully. For more information, see the *Updating help* section of this chapter.

Use the following command to return a list of all **About** help topics on your system.

```
help About_*
```

When you limit the results to one **About** help topic, the content of the help topic is displayed instead of returning a list of help topics.

```
help about_Updatable_Help
```

Updating help

Earlier in this chapter, you updated the PowerShell help topics on your computer the first time you ran the `Get-Help` cmdlet. You should periodically run the `Update-Help` cmdlet on your computer to obtain any updates to the help content. `Update-Help` requires internet access by default.

> When you run `Update-Help` in Windows PowerShell 5.1, it requires running PowerShell elevated as an administrator.

In the following example, use the `Update-Help` cmdlet to update the PowerShell help content on your computer.

```
Update-Help
```

As shown in the following results, a module returned an error. Errors aren't uncommon and usually occur when the module's author doesn't configure updatable help correctly.

```
Update-Help : Failed to update Help for the module(s) 'BitsTransfer' with UI
culture(s) {en-US} : Unable to retrieve the HelpInfo XML file for UI culture
en-US. Make sure the HelpInfoUri property in the module manifest is valid or
check your network connection and then try the command again.
At line:1 char:1
+ Update-Help
+ ~~~~~~~~~~~
    + CategoryInfo          : ResourceUnavailable: (:) [Update-Help], Except
   ion
    + FullyQualifiedErrorId : UnableToRetrieveHelpInfoXml,Microsoft.PowerShe
   ll.Commands.UpdateHelpCommand
```

If your computer doesn't have internet access, use the `Save-Help` cmdlet on a computer with internet access to download and save the updated help content. Then, use the **SourcePath** parameter of `Update-Help` to specify the location of the saved updated help content.

Get-Command

`Get-Command` is another multipurpose command that helps you locate commands. You can also use `Get-Command` to learn how to use commands, but in a different and more indirect way when compared to `Get-Help`.

How do you determine the syntax for `Get-Command`? You could use `Get-Help` to display the help topic for `Get-Command`, as shown in the *Get-Help* section of this chapter. You can also use `Get-Command` with the **Syntax** parameter to view the syntax for any command. This shortcut helps you quickly determine how to use a command without navigating through its help content.

```
Get-Command -Name Get-Command -Syntax
```

Using `Get-Command` with its **Syntax** parameter offers a more programmatic view by showing the type of the parameter, without listing the specific allowable values.

```
Get-Command [[-ArgumentList] <Object[]>] [-Verb <string[]>]
[-Noun <string[]>] [-Module <string[]>]
[-FullyQualifiedModule <ModuleSpecification[]>] [-TotalCount <int>]
[-Syntax] [-ShowCommandInfo] [-All] [-ListImported]
[-ParameterName <string[]>] [-ParameterType <PSTypeName[]>]
[<CommonParameters>]

Get-Command [[-Name] <string[]>] [[-ArgumentList] <Object[]>]
[-Module <string[]>] [-FullyQualifiedModule <ModuleSpecification[]>]
[-CommandType <CommandTypes>] [-TotalCount <int>] [-Syntax]
[-ShowCommandInfo] [-All] [-ListImported] [-ParameterName <string[]>]
[-ParameterType <PSTypeName[]>] [<CommonParameters>]
```

If you need more detailed information about how to use a command, revert to using `Get-Help`.

```
help Get-Command -Full
```

The **SYNTAX** section of `Get-Help` provides a more user-friendly display by expanding enumerated values for parameters. It shows you the actual values you can use, making it easier to understand the available options.

```
...
    Get-Command [[-Name] <System.String[]>] [[-ArgumentList]
    <System.Object[]>] [-All] [-CommandType {Alias | Function | Filter |
    Cmdlet | ExternalScript | Application | Script | Workflow |
    Configuration | All}] [-FullyQualifiedModule
    <Microsoft.PowerShell.Commands.ModuleSpecification[]>] [-ListImported]
    [-Module <System.String[]>] [-ParameterName <System.String[]>]
    [-ParameterType <System.Management.Automation.PSTypeName[]>]
    [-ShowCommandInfo] [-Syntax] [-TotalCount <System.Int32>]
    [<CommonParameters>]

    Get-Command [[-ArgumentList] <System.Object[]>] [-All]
    [-FullyQualifiedModule
    <Microsoft.PowerShell.Commands.ModuleSpecification[]>] [-ListImported]
    [-Module <System.String[]>] [-Noun <System.String[]>] [-ParameterName
    <System.String[]>] [-ParameterType
```

```
    <System.Management.Automation.PSTypeName[]>] [-ShowCommandInfo]
    [-Syntax] [-TotalCount <System.Int32>] [-Verb <System.String[]>]
    [<CommonParameters>]
...
```

The **PARAMETERS** section of the help topic for `Get-Command` reveals the **Name**, **Noun**, and **Verb** parameters accept wildcard characters.

```
...
    -Name <System.String[]>
        Specifies an array of names. This cmdlet gets only commands that
        have the specified name. Enter a name or name pattern. Wildcard
        characters are permitted.

        To get commands that have the same name, use the All parameter. When
        two commands have the same name, by default, `Get-Command` gets the
        command that runs when you type the command name.

        Required?                    false
        Position?                    0
        Default value                None
        Accept pipeline input?       True (ByPropertyName, ByValue)
        Accept wildcard characters?  true

    -Noun <System.String[]>
        Specifies an array of command nouns. This cmdlet gets commands,
        which include cmdlets, functions, and aliases, that have names that
        include the specified noun. Enter one or more nouns or noun
        patterns. Wildcard characters are permitted.

        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       True (ByPropertyName)
        Accept wildcard characters?  true
    -Verb <System.String[]>
        Specifies an array of command verbs. This cmdlet gets commands,
        which include cmdlets, functions, and aliases, that have names that
        include the specified verb. Enter one or more verbs or verb
```

```
        patterns. Wildcard characters are permitted.

        Required?                    false
        Position?                    named
        Default value                None
        Accept pipeline input?       True (ByPropertyName)
        Accept wildcard characters?  true
...
```

In the following example, use * wildcard characters with the **Name** parameter of `Get-Command`.

```
Get-Command -Name *service*
```

When using wildcard characters with the **Name** parameter of `Get-Command`, it returns PowerShell commands and native commands, as shown in the following results.

```
CommandType     Name                                               Version
-----------     ----                                               -------
Function        Get-NetFirewallServiceFilter                       2.0.0.0
Function        Set-NetFirewallServiceFilter                       2.0.0.0
Cmdlet          Get-Service                                        3.1.0.0
Cmdlet          New-Service                                        3.1.0.0
Cmdlet          New-WebServiceProxy                                3.1.0.0
Cmdlet          Restart-Service                                    3.1.0.0
Cmdlet          Resume-Service                                     3.1.0.0
Cmdlet          Set-Service                                        3.1.0.0
Cmdlet          Start-Service                                      3.1.0.0
Cmdlet          Stop-Service                                       3.1.0.0
Cmdlet          Suspend-Service                                    3.1.0.0
Application     SecurityHealthService.exe                          10.0.2...
Application     SensorDataService.exe                              10.0.2...
Application     services.exe                                       10.0.2...
Application     services.msc                                       0.0.0.0
Application     TieringEngineService.exe                           10.0.2...
Application     Windows.WARP.JITService.exe                        10.0.2...
```

If you use wildcard characters with the **Name** parameter of `Get-Command`, consider limiting the results to PowerShell cmdlets, functions, and aliases with the **CommandType** parameter.

```
Get-Command -Name *service* -CommandType Cmdlet, Function, Alias
```

A better option might be to use either the **Verb** or **Noun** parameter or both since only PowerShell commands have verbs and nouns.

In the following example, use `Get-Command` to determine what commands exist on your computer for working with processes. Use the **Noun** parameter and specify `Process` as its value.

```
Get-Command -Noun Process
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Cmdlet          Debug-Process                                      3.1.0.0
Cmdlet          Get-Process                                        3.1.0.0
Cmdlet          Start-Process                                      3.1.0.0
Cmdlet          Stop-Process                                       3.1.0.0
Cmdlet          Wait-Process                                       3.1.0.0
```

When you run `Get-Command` without any parameters, it returns a list of all the commands on your system.

Contributing to the documentation

The help content for PowerShell is open source and available in the PowerShell-Docs[1] repository on GitHub. For more information, see Contributing to PowerShell documentation[2].

Summary

In this chapter, you've learned how to find commands with `Get-Help` and `Get-Command`. You've also learned how to use the help system to figure out how to use commands once you find them. In addition, you've learned how to update the help system on your computer when new help content is available.

[1] https://github.com/MicrosoftDocs/PowerShell-Docs
[2] https://learn.microsoft.com/powershell/scripting/community/contributing/overview

Review

1. Is the **DisplayName** parameter of `Get-Service` positional?
2. How many parameter sets does the `Get-Process` cmdlet have?
3. What PowerShell commands exist for working with event logs?
4. What's the PowerShell command for returning a list of PowerShell processes running on your computer?
5. How do you update the PowerShell help content stored on your computer?

References

To learn more about the information covered in this chapter, read the following PowerShell help topics.

- Get-Help[3]
- Get-Command[4]
- Update-Help[5]
- Save-Help[6]
- about_Updatable_Help[7]
- about_Command_Syntax[8]

Next steps

In the next chapter, you'll learn about objects, properties, methods, and the `Get-Member` cmdlet.

[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/get-help
[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/get-command
[5] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/update-help
[6] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/save-help
[7] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_updatable_help
[8] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_command_syntax

Chapter 3 - Discovering Objects, Properties, and Methods

PowerShell is an object-oriented scripting language. It represents data and system states using structured objects derived from .NET classes defined in the .NET Framework. By leveraging the .NET Framework, PowerShell offers access to various system capabilities, including file system, registry, and Windows Management Instrumentation (WMI) classes. PowerShell also has access to the .NET Framework class library, which contains a vast collection of classes that you can use to develop robust PowerShell scripts.

In PowerShell, each item or state is an instance of an object that can be explored and manipulated. The `Get-Member` cmdlet is one of the primary tools provided by PowerShell for object discovery, which reveals an object's characteristics. This chapter explores how PowerShell leverages objects and how you can discover and manipulate these objects to streamline your scripts and manage your systems more efficiently.

Prerequisites

To follow the specific examples in this chapter, ensure that your lab environment computer is part of your lab environment Active Directory domain. You must also install the Active Directory PowerShell module bundled with the Windows Remote Server Administration Tools (RSAT). If you're using Windows 10 build 1809 or later, including Windows 11, you can install RSAT as a Windows feature.

> Active Directory is unsupported for Windows Home editions.

- For information about installing RSAT, see Windows Management modules[1].

[1] https://learn.microsoft.com/powershell/scripting/whats-new/module-compatibility#windows-management-modules

- For older versions of Windows, see RSAT for Windows[2].

Get-Member

`Get-Member` provides insight into the objects, properties, and methods associated with PowerShell commands. You can pipe any PowerShell command that produces object-based output to `Get-Member`. When you pipe the output of a command to `Get-Member`, it reveals the structure of the object returned by the command, detailing its properties and methods.

- **Properties**: The attributes of an object.
- **Methods**: The actions you can perform on an object.

Consider a driver's license as an analogy to illustrate this concept. Like any object, a driver's license has properties, such as **eye color**, which typically includes `blue` and `brown` values. In contrast, methods represent actions you can perform on the object. For instance, **Revoke** is a method that the Department of Motor Vehicles can perform on a driver's license.

Properties

To retrieve details about the Windows Time service on your system using PowerShell, use the `Get-Service` cmdlet.

```
Get-Service -Name w32time
```

The results include the **Status**, **Name**, and **DisplayName** properties. The **Status** property indicates that the service is `Running`. The value for the **Name** property is `w32time`, and the value for the **DisplayName** property is `Windows Time`.

[2] https://learn.microsoft.com/troubleshoot/windows-server/system-management-components/remote-server-administration-tools

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time
```

To list all available properties and methods for `Get-Service`, pipe it to `Get-Member`.

```
Get-Service -Name w32time | Get-Member
```

The results show the first line contains one piece of significant information. **TypeName** identifies the type of object that's returned, which in this example is a **System.ServiceProcess.ServiceController** object. This is often abbreviated to the last part of the **TypeName**, such as **ServiceController**, in this example.

```
   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDepend...
Disposed                  Event         System.EventHandler Disposed(Syst...
Close                     Method        void Close()
Continue                  Method        void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef Cr...
Dispose                   Method        void Dispose(), void IDisposable....
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        void ExecuteCommand(int command)
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetimeS...
Pause                     Method        void Pause()
Refresh                   Method        void Refresh()
Start                     Method        void Start(), void Start(string[]...
Stop                      Method        void Stop()
WaitForStatus             Method        void WaitForStatus(System.Service...
CanPauseAndContinue       Property      bool CanPauseAndContinue {get;}
CanShutdown               Property      bool CanShutdown {get;}
CanStop                   Property      bool CanStop {get;}
```

```
Container                 Property      System.ComponentModel.IContainer ...
DependentServices         Property      System.ServiceProcess.ServiceCont...
DisplayName               Property      string DisplayName {get;set;}
MachineName               Property      string MachineName {get;set;}
ServiceHandle             Property      System.Runtime.InteropServices.Sa...
ServiceName               Property      string ServiceName {get;set;}
ServicesDependedOn        Property      System.ServiceProcess.ServiceCont...
ServiceType               Property      System.ServiceProcess.ServiceType...
Site                      Property      System.ComponentModel.ISite Site ...
StartType                 Property      System.ServiceProcess.ServiceStar...
Status                    Property      System.ServiceProcess.ServiceCont...
ToString                  ScriptMethod  System.Object ToString();
```

Notice when you piped `Get-Service` to `Get-Memnber`, there are more properties than are displayed by default. Although these additional properties aren't shown by default, you can select them by piping to `Select-Object` and using the **Property** parameter. The following example selects all properties by piping the results of `Get-Service` to `Select-Object` and specifying the `*` wildcard character as the value for the **Property** parameter.

```
Get-Service -Name w32time | Select-Object -Property *
```

By default, PowerShell returns four properties as a table and five or more as a list. However, some commands apply custom formatting to override the default number of properties displayed in a table. You can use `Format-Table` and `Format-List` to override these defaults manually.

```
Name                  : w32time
RequiredServices      : {}
CanPauseAndContinue   : False
CanShutdown           : True
CanStop               : True
DisplayName           : Windows Time
DependentServices     : {}
MachineName           : .
ServiceName           : w32time
ServicesDependedOn    : {}
ServiceHandle         :
```

```
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Manual
Site                :
Container           :
```

Specific properties can also be selected using a comma-separated list as the value of the **Property** parameter.

```
Get-Service -Name w32time |
    Select-Object -Property Status, Name, DisplayName, ServiceType
```

```
 Status Name    DisplayName                          ServiceType
 ------ ----    -----------                          -----------
Running w32time Windows Time Win32OwnProcess, Win32ShareProcess
```

You can use wildcard characters when specifying property names with `Select-Object`.

In the following example, use `Can*` as one of the values for the **Property** parameter to return all the properties that start with `Can`. These include **CanPauseAndContinue**, **CanShutdown**, and **CanStop**.

```
Get-Service -Name w32time |
    Select-Object -Property Status, DisplayName, Can*
```

Notice there are more properties listed than are displayed by default.

```
Status              : Running
DisplayName         : Windows Time
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
```

Methods

Methods are actions you can perform on an object. Use the **MemberType** parameter to narrow down the results of `Get-Member` to display only the methods for `Get-Service`.

```
Get-Service -Name w32time | Get-Member -MemberType Method
```

As you can see, there are many methods.

```
   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType Definition
----                      ---------- ----------
Close                     Method     void Close()
Continue                  Method     void Continue()
CreateObjRef              Method     System.Runtime.Remoting.ObjRef Creat...
Dispose                   Method     void Dispose(), void IDisposable.Dis...
Equals                    Method     bool Equals(System.Object obj)
ExecuteCommand            Method     void ExecuteCommand(int command)
GetHashCode               Method     int GetHashCode()
GetLifetimeService        Method     System.Object GetLifetimeService()
GetType                   Method     type GetType()
InitializeLifetimeService Method     System.Object InitializeLifetimeServ...
Pause                     Method     void Pause()
Refresh                   Method     void Refresh()
Start                     Method     void Start(), void Start(string[] args)
Stop                      Method     void Stop()
WaitForStatus             Method     void WaitForStatus(System.ServicePro...
```

You can use the **Stop** method to stop a Windows service. You must run this command from an elevated PowerShell session.

```
(Get-Service -Name w32time).Stop()
```

Query the status of the Windows Time service to confirm it's stopped.

```
Get-Service -Name w32time
```

```
Status   Name               DisplayName
------   ----               -----------
Stopped  w32time            Windows Time
```

You might use methods sparingly, but you should be aware of them. Sometimes, you'll find a `Get-*` command without a corresponding `Set-*` command. Often, you can find a method to perform a `Set-*` action in this scenario. The `Get-SqlAgentJob` cmdlet in the SqlServer PowerShell module is an excellent example. No corresponding `Set-*` cmdlet exists, but you can use a method to complete the same task. For more information about the SqlServer PowerShell module and installation instructions, see the SQL Server PowerShell overview[3].

Another reason to be aware of methods is many PowerShell users assume you can't make destructive changes with `Get-*` commands, but they can actually cause severe problems if misused.

A better option is to use a dedicated cmdlet if one exists to perform an action. For example, use the `Start-Service` cmdlet to start the Windows Time service.

By default, `Start-Service`, like the **Start** method of `Get-Service`, doesn't return any results. But one of the benefits of using a cmdlet is that it often provides additional capabilities that aren't available with a method.

In the following example, use the **PassThru** parameter, which causes a cmdlet that doesn't typically produce output to generate output. You must run this command from an elevated PowerShell session.

```
Get-Service -Name w32time | Start-Service -PassThru
```

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time
```

> When working with PowerShell cmdlets, it's important to avoid making assumptions about their output.

To retrieve information about the PowerShell process running on your lab environment computer, use the `Get-Process` cmdlet.

[3] https://learn.microsoft.com/sql/powershell/download-sql-server-ps-module

```
Get-Process -Name PowerShell
```

```
Handles  NPM(K)    PM(K)      WS(K)     CPU(s)      Id  SI ProcessName
-------  ------    -----      -----     ------      --  -- -----------
    710      31    55692      70580       0.72   9436   2 powershell
```

To determine the available properties, pipe `Get-Process` to `Get-Member`.

```
Get-Process -Name PowerShell | Get-Member
```

When using the `Get-Process` command, you may notice that some properties displayed by default are missing when you view the results of `Get-Member`. This is because many of the values shown by default, such as `NPM(K)`, `PM(K)`, `WS(K)`, and `CPU(s)`, are calculated properties. You must pipe commands to `Get-Member` to determine their actual property names.

```
   TypeName: System.Diagnostics.Process

Name                      MemberType     Definition
----                      ----------     ----------
Handles                   AliasProperty  Handles = Handlecount
Name                      AliasProperty  Name = ProcessName
NPM                       AliasProperty  NPM = NonpagedSystemMemorySize64
PM                        AliasProperty  PM = PagedMemorySize64
SI                        AliasProperty  SI = SessionId
VM                        AliasProperty  VM = VirtualMemorySize64
WS                        AliasProperty  WS = WorkingSet64
Disposed                  Event          System.EventHandler Disposed(Sy...
ErrorDataReceived         Event          System.Diagnostics.DataReceived...
Exited                    Event          System.EventHandler Exited(Syst...
OutputDataReceived        Event          System.Diagnostics.DataReceived...
BeginErrorReadLine        Method         void BeginErrorReadLine()
BeginOutputReadLine       Method         void BeginOutputReadLine()
CancelErrorRead           Method         void CancelErrorRead()
CancelOutputRead          Method         void CancelOutputRead()
Close                     Method         void Close()
CloseMainWindow           Method         bool CloseMainWindow()
```

```
CreateObjRef                Method         System.Runtime.Remoting.ObjRef ...
Dispose                     Method         void Dispose(), void IDisposabl...
Equals                      Method         bool Equals(System.Object obj)
GetHashCode                 Method         int GetHashCode()
GetLifetimeService          Method         System.Object GetLifetimeService()
GetType                     Method         type GetType()
InitializeLifetimeService   Method         System.Object InitializeLifetim...
Kill                        Method         void Kill()
Refresh                     Method         void Refresh()
Start                       Method         bool Start()
ToString                    Method         string ToString()
WaitForExit                 Method         bool WaitForExit(int millisecon...
WaitForInputIdle            Method         bool WaitForInputIdle(int milli...
__NounName                  NoteProperty   string __NounName=Process
BasePriority                Property       int BasePriority {get;}
Container                   Property       System.ComponentModel.IContaine...
EnableRaisingEvents         Property       bool EnableRaisingEvents {get;s...
ExitCode                    Property       int ExitCode {get;}
ExitTime                    Property       datetime ExitTime {get;}
Handle                      Property       System.IntPtr Handle {get;}
HandleCount                 Property       int HandleCount {get;}
HasExited                   Property       bool HasExited {get;}
Id                          Property       int Id {get;}
MachineName                 Property       string MachineName {get;}
MainModule                  Property       System.Diagnostics.ProcessModul...
MainWindowHandle            Property       System.IntPtr MainWindowHandle ...
MainWindowTitle             Property       string MainWindowTitle {get;}
MaxWorkingSet               Property       System.IntPtr MaxWorkingSet {ge...
MinWorkingSet               Property       System.IntPtr MinWorkingSet {ge...
Modules                     Property       System.Diagnostics.ProcessModul...
NonpagedSystemMemorySize    Property       int NonpagedSystemMemorySize {g...
NonpagedSystemMemorySize64  Property       long NonpagedSystemMemorySize64...
PagedMemorySize             Property       int PagedMemorySize {get;}
PagedMemorySize64           Property       long PagedMemorySize64 {get;}
PagedSystemMemorySize       Property       int PagedSystemMemorySize {get;}
PagedSystemMemorySize64     Property       long PagedSystemMemorySize64 {g...
PeakPagedMemorySize         Property       int PeakPagedMemorySize {get;}
PeakPagedMemorySize64       Property       long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize       Property       int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64     Property       long PeakVirtualMemorySize64 {g...
```

```
PeakWorkingSet              Property       int PeakWorkingSet {get;}
PeakWorkingSet64            Property       long PeakWorkingSet64 {get;}
PriorityBoostEnabled        Property       bool PriorityBoostEnabled {get;...
PriorityClass               Property       System.Diagnostics.ProcessPrior...
PrivateMemorySize           Property       int PrivateMemorySize {get;}
PrivateMemorySize64         Property       long PrivateMemorySize64 {get;}
PrivilegedProcessorTime     Property       timespan PrivilegedProcessorTim...
ProcessName                 Property       string ProcessName {get;}
ProcessorAffinity           Property       System.IntPtr ProcessorAffinity...
Responding                  Property       bool Responding {get;}
SafeHandle                  Property       Microsoft.Win32.SafeHandles.Saf...
SessionId                   Property       int SessionId {get;}
Site                        Property       System.ComponentModel.ISite Sit...
StandardError               Property       System.IO.StreamReader Standard...
StandardInput               Property       System.IO.StreamWriter Standard...
StandardOutput              Property       System.IO.StreamReader Standard...
StartInfo                   Property       System.Diagnostics.ProcessStart...
StartTime                   Property       datetime StartTime {get;}
SynchronizingObject         Property       System.ComponentModel.ISynchron...
Threads                     Property       System.Diagnostics.ProcessThrea...
TotalProcessorTime          Property       timespan TotalProcessorTime {get;}
UserProcessorTime           Property       timespan UserProcessorTime {get;}
VirtualMemorySize           Property       int VirtualMemorySize {get;}
VirtualMemorySize64         Property       long VirtualMemorySize64 {get;}
WorkingSet                  Property       int WorkingSet {get;}
WorkingSet64                Property       long WorkingSet64 {get;}
PSConfiguration             PropertySet    PSConfiguration {Name, Id, Prio...
PSResources                 PropertySet    PSResources {Name, Id, Handleco...
Company                     ScriptProperty System.Object Company {get=$thi...
CPU                         ScriptProperty System.Object CPU {get=$this.To...
Description                 ScriptProperty System.Object Description {get=...
FileVersion                 ScriptProperty System.Object FileVersion {get=...
Path                        ScriptProperty System.Object Path {get=$this.M...
Product                     ScriptProperty System.Object Product {get=$thi...
ProductVersion              ScriptProperty System.Object ProductVersion {g...
```

You can't pipe a command to `Get-Member` that doesn't generate output. Because `Start-Service` doesn't produce output by default, attempting to pipe it to `Get-Member` results in an error. You must run this command from an elevated PowerShell session.

```
Start-Service -Name w32time | Get-Member
```

To be piped to `Get-Member`, a command must produce object-based output.

```
Get-Member : You must specify an object for the Get-Member cmdlet.
At line:1 char:31
+ Start-Service -Name w32time | Get-Member
+                               ~~~~~~~~~~
    + CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperation
   Exception
    + FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Comma
   nds.GetMemberCommand
```

To avoid this error, specify the **PassThru** parameter with `Start-Service`. Adding the **PassThru** parameter causes a cmdlet that doesn't usually produce output to generate output. You must run this command from an elevated PowerShell session.

```
Start-Service -Name w32time -PassThru | Get-Member
```

```
   TypeName: System.ServiceProcess.ServiceController

Name                      MemberType    Definition
----                      ----------    ----------
Name                      AliasProperty Name = ServiceName
RequiredServices          AliasProperty RequiredServices = ServicesDepend...
Disposed                  Event         System.EventHandler Disposed(Syst...
Close                     Method        void Close()
Continue                  Method        void Continue()
CreateObjRef              Method        System.Runtime.Remoting.ObjRef Cr...
Dispose                   Method        void Dispose(), void IDisposable....
Equals                    Method        bool Equals(System.Object obj)
ExecuteCommand            Method        void ExecuteCommand(int command)
GetHashCode               Method        int GetHashCode()
GetLifetimeService        Method        System.Object GetLifetimeService()
GetType                   Method        type GetType()
InitializeLifetimeService Method        System.Object InitializeLifetimeS...
```

```
Pause                   Method        void Pause()
Refresh                 Method        void Refresh()
Start                   Method        void Start(), void Start(string[]...
Stop                    Method        void Stop()
WaitForStatus           Method        void WaitForStatus(System.Service...
CanPauseAndContinue     Property      bool CanPauseAndContinue {get;}
CanShutdown             Property      bool CanShutdown {get;}
CanStop                 Property      bool CanStop {get;}
Container               Property      System.ComponentModel.IContainer ...
DependentServices       Property      System.ServiceProcess.ServiceCont...
DisplayName             Property      string DisplayName {get;set;}
MachineName             Property      string MachineName {get;set;}
ServiceHandle           Property      System.Runtime.InteropServices.Sa...
ServiceName             Property      string ServiceName {get;set;}
ServicesDependedOn      Property      System.ServiceProcess.ServiceCont...
ServiceType             Property      System.ServiceProcess.ServiceType...
Site                    Property      System.ComponentModel.ISite Site ...
StartType               Property      System.ServiceProcess.ServiceStar...
Status                  Property      System.ServiceProcess.ServiceCont...
ToString                ScriptMethod  System.Object ToString();
```

`Out-Host` is designed to show output directly in the PowerShell host and doesn't produce object-based output. As a result, you can't pipe its output to `Get-Member`, which requires object-based input.

```
Get-Service -Name w32time | Out-Host | Get-Member
```

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time

Get-Member : You must specify an object for the Get-Member cmdlet.
At line:1 char:40
+ Get-Service -Name w32time | Out-Host | Get-Member
+                                        ~~~~~~~~~~
    + CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperation
   Exception
    + FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Comma
   nds.GetMemberCommand
```

Get-Command

Knowing the type of object a command produces allows you to search for commands that accept that type of object as input.

```
Get-Command -ParameterType ServiceController
```

The following commands accept a **ServiceController** object via pipeline or parameter input.

```
CommandType     Name                                               Version
-----------     ----                                               -------
Cmdlet          Get-Service                                        3.1.0.0
Cmdlet          Restart-Service                                    3.1.0.0
Cmdlet          Resume-Service                                     3.1.0.0
Cmdlet          Set-Service                                        3.1.0.0
Cmdlet          Start-Service                                      3.1.0.0
Cmdlet          Stop-Service                                       3.1.0.0
Cmdlet          Suspend-Service                                    3.1.0.0
```

Active Directory

> As mentioned in the chapter prerequisites, ensure you have RSAT installed for this section. Additionally, your lab environment computer must be a member of your lab environment Active Directory domain.

To identify the commands added to the ActiveDirectory PowerShell module after you install RSAT, use `Get-Command` combined with the **Module** parameter. The following example lists all the commands available in the ActiveDirectory module.

```
Get-Command -Module ActiveDirectory
```

The ActiveDirectory PowerShell module added a total of 147 commands.

Have you observed the naming convention of these commands? The nouns in the command names are prefixed with *AD* to avoid potential naming conflicts with commands in other modules. This prefixing is a common practice among PowerShell modules.

```
CommandType     Name                                               Version
-----------     ----                                               -------
Cmdlet          Add-ADCentralAccessPolicyMember                    1.0.1.0
Cmdlet          Add-ADComputerServiceAccount                       1.0.1.0
Cmdlet          Add-ADDomainControllerPasswordReplicationPolicy    1.0.1.0
Cmdlet          Add-ADFineGrainedPasswordPolicySubject             1.0.1.0
Cmdlet          Add-ADGroupMember                                  1.0.1.0
Cmdlet          Add-ADPrincipalGroupMembership                     1.0.1.0
Cmdlet          Add-ADResourcePropertyListMember                   1.0.1.0
Cmdlet          Clear-ADAccountExpiration                          1.0.1.0
Cmdlet          Clear-ADClaimTransformLink                         1.0.1.0
Cmdlet          Disable-ADAccount                                  1.0.1.0
...
```

By default, the `Get-ADUser` cmdlet retrieves a limited set of properties for user objects and limits its output to the first 1000 users. This constraint is a performance optimization designed to avoid overwhelming Active Directory with excessive data retrieval.

```
Get-ADUser -Identity mike | Get-Member -MemberType Properties
```

Even if you only have a basic understanding of Active Directory, you may recognize that a user account has more properties than those shown in the example.

```
   TypeName: Microsoft.ActiveDirectory.Management.ADUser

Name              MemberType Definition
----              ---------- ----------
DistinguishedName Property   System.String DistinguishedName {get;set;}
Enabled           Property   System.Boolean Enabled {get;set;}
GivenName         Property   System.String GivenName {get;set;}
Name              Property   System.String Name {get;}
ObjectClass       Property   System.String ObjectClass {get;set;}
ObjectGUID        Property   System.Nullable`1[[System.Guid, mscorlib, Ve...
SamAccountName    Property   System.String SamAccountName {get;set;}
SID               Property   System.Security.Principal.SecurityIdentifier...
Surname           Property   System.String Surname {get;set;}
UserPrincipalName Property   System.String UserPrincipalName {get;set;}
```

The `Get-ADUser` cmdlet includes a **Properties** parameter to specify additional properties beyond the defaults you want to retrieve. Use the * wildcard character as the parameter value to return all properties.

```
Get-ADUser -Identity mike -Properties * | Get-Member -MemberType Properties
```

```
   TypeName: Microsoft.ActiveDirectory.Management.ADUser

Name                                  MemberType Definition
----                                  ---------- ----------
AccountExpirationDate                 Property   System.DateTime AccountEx...
accountExpires                        Property   System.Int64 accountExpir...
AccountLockoutTime                    Property   System.DateTime AccountLo...
AccountNotDelegated                   Property   System.Boolean AccountNot...
AllowReversiblePasswordEncryption     Property   System.Boolean AllowRever...
AuthenticationPolicy                  Property   Microsoft.ActiveDirectory...
AuthenticationPolicySilo              Property   Microsoft.ActiveDirectory...
BadLogonCount                         Property   System.Int32 BadLogonCoun...
badPasswordTime                       Property   System.Int64 badPasswordT...
badPwdCount                           Property   System.Int32 badPwdCount ...
CannotChangePassword                  Property   System.Boolean CannotChan...
CanonicalName                         Property   System.String CanonicalNa...
Certificates                          Property   Microsoft.ActiveDirectory...
City                                  Property   System.String City {get;s...
CN                                    Property   System.String CN {get;}
codePage                              Property   System.Int32 codePage {ge...
Company                               Property   System.String Company {ge...
CompoundIdentitySupported             Property   Microsoft.ActiveDirectory...
Country                               Property   System.String Country {ge...
countryCode                           Property   System.Int32 countryCode ...
Created                               Property   System.DateTime Created {...
createTimeStamp                       Property   System.DateTime createTim...
Deleted                               Property   System.Boolean Deleted {g...
Department                            Property   System.String Department ...
Description                           Property   System.String Description...
DisplayName                           Property   System.String DisplayName...
DistinguishedName                     Property   System.String Distinguish...
Division                              Property   System.String Division {g...
```

```
DoesNotRequirePreAuth                Property   System.Boolean DoesNotReq...
dSCorePropagationData                Property   Microsoft.ActiveDirectory...
EmailAddress                         Property   System.String EmailAddres...
EmployeeID                           Property   System.String EmployeeID ...
EmployeeNumber                       Property   System.String EmployeeNum...
Enabled                              Property   System.Boolean Enabled {g...
Fax                                  Property   System.String Fax {get;set;}
GivenName                            Property   System.String GivenName {...
HomeDirectory                        Property   System.String HomeDirecto...
HomedirRequired                      Property   System.Boolean HomedirReq...
HomeDrive                            Property   System.String HomeDrive {...
HomePage                             Property   System.String HomePage {g...
HomePhone                            Property   System.String HomePhone {...
Initials                             Property   System.String Initials {g...
instanceType                         Property   System.Int32 instanceType...
isDeleted                            Property   System.Boolean isDeleted ...
KerberosEncryptionType               Property   Microsoft.ActiveDirectory...
LastBadPasswordAttempt               Property   System.DateTime LastBadPa...
LastKnownParent                      Property   System.String LastKnownPa...
lastLogoff                           Property   System.Int64 lastLogoff {...
lastLogon                            Property   System.Int64 lastLogon {g...
LastLogonDate                        Property   System.DateTime LastLogon...
lastLogonTimestamp                   Property   System.Int64 lastLogonTim...
LockedOut                            Property   System.Boolean LockedOut ...
logonCount                           Property   System.Int32 logonCount {...
LogonWorkstations                    Property   System.String LogonWorkst...
Manager                              Property   System.String Manager {ge...
MemberOf                             Property   Microsoft.ActiveDirectory...
MNSLogonAccount                      Property   System.Boolean MNSLogonAc...
MobilePhone                          Property   System.String MobilePhone...
Modified                             Property   System.DateTime Modified ...
modifyTimeStamp                      Property   System.DateTime modifyTim...
msDS-User-Account-Control-Computed   Property   System.Int32 msDS-User-Ac...
Name                                 Property   System.String Name {get;}
nTSecurityDescriptor                 Property   System.DirectoryServices....
ObjectCategory                       Property   System.String ObjectCateg...
ObjectClass                          Property   System.String ObjectClass...
ObjectGUID                           Property   System.Nullable`1[[System...
objectSid                            Property   System.Security.Principal...
Office                               Property   System.String Office {get...
```

```
OfficePhone                              Property   System.String OfficePhone...
Organization                             Property   System.String Organizatio...
OtherName                                Property   System.String OtherName {...
PasswordExpired                          Property   System.Boolean PasswordEx...
PasswordLastSet                          Property   System.DateTime PasswordL...
PasswordNeverExpires                     Property   System.Boolean PasswordNe...
PasswordNotRequired                      Property   System.Boolean PasswordNo...
POBox                                    Property   System.String POBox {get;...
PostalCode                               Property   System.String PostalCode ...
PrimaryGroup                             Property   System.String PrimaryGrou...
primaryGroupID                           Property   System.Int32 primaryGroup...
PrincipalsAllowedToDelegateToAccount Property   Microsoft.ActiveDirectory...
ProfilePath                              Property   System.String ProfilePath...
ProtectedFromAccidentalDeletion          Property   System.Boolean ProtectedF...
pwdLastSet                               Property   System.Int64 pwdLastSet {...
SamAccountName                           Property   System.String SamAccountN...
sAMAccountType                           Property   System.Int32 sAMAccountTy...
ScriptPath                               Property   System.String ScriptPath ...
sDRightsEffective                        Property   System.Int32 sDRightsEffe...
ServicePrincipalNames                    Property   Microsoft.ActiveDirectory...
SID                                      Property   System.Security.Principal...
SIDHistory                               Property   Microsoft.ActiveDirectory...
SmartcardLogonRequired                   Property   System.Boolean SmartcardL...
sn                                       Property   System.String sn {get;set;}
State                                    Property   System.String State {get;...
StreetAddress                            Property   System.String StreetAddre...
Surname                                  Property   System.String Surname {ge...
Title                                    Property   System.String Title {get;...
TrustedForDelegation                     Property   System.Boolean TrustedFor...
TrustedToAuthForDelegation               Property   System.Boolean TrustedToA...
UseDESKeyOnly                            Property   System.Boolean UseDESKeyO...
userAccountControl                       Property   System.Int32 userAccountC...
userCertificate                          Property   Microsoft.ActiveDirectory...
UserPrincipalName                        Property   System.String UserPrincip...
uSNChanged                               Property   System.Int64 uSNChanged {...
uSNCreated                               Property   System.Int64 uSNCreated {...
whenChanged                              Property   System.DateTime whenChang...
whenCreated                              Property   System.DateTime whenCreat...
```

The default configuration for retrieving Active Directory user account properties is

intentionally limited to avoid performance issues. Trying to return every property for every user account in your production Active Directory environment could severely degrade the performance of your domain controllers and network. In most cases, you'll only need specific properties for certain users. However, returning all properties for a single user is reasonable when identifying the available properties.

It's not uncommon to run a command multiple times when prototyping it. If you anticipate running a resource-intensive query when prototyping a command, consider executing it once and storing the results in a variable. Then, you can work with the variable's contents more efficiently than repeatedly executing a resource-intensive query.

For example, the following command retrieves all properties for a user account and stores the results in a variable named `$Users`. Work with the contents of the `$Users` variable instead of running the `Get-ADUser` command multiple times. Remember, the variable's contents don't update automatically when a user's information changes in Active Directory.

```
$Users = Get-ADUser -Identity mike -Properties *
```

You can explore the available properties by piping the `$Users` variable to `Get-Member`.

```
$Users | Get-Member -MemberType Properties
```

To view specific properties such as **Name**, **LastLogonDate**, and **LastBadPasswordAttempt**, pipe the `$Users` variable to `Select-Object`. This method displays the desired properties and their values based on the contents of the `$Users` variable, eliminating the need for multiple queries to Active Directory. It's a more resource-efficient approach than repeatedly executing the `Get-ADUser` command.

```
$Users | Select-Object -Property Name, LastLogonDate, LastBadPasswordAttempt
```

When querying Active Directory, filter the data at the source using the **Properties** parameter to return only the necessary properties.

```
Get-ADUser -Identity mike -Properties LastLogonDate, LastBadPasswordAttempt
```

```
DistinguishedName      : CN=Mike F. Robbins,CN=Users,DC=mikefrobbins,DC=com
Enabled                : True
GivenName              : Mike
LastBadPasswordAttempt :
LastLogonDate          : 11/14/2023 5:10:16 AM
Name                   : Mike F. Robbins
ObjectClass            : user
ObjectGUID             : 11c7b61f-46c3-4399-9ed0-ff4e453bc2a2
SamAccountName         : mike
SID                    : S-1-5-21-611971124-518002951-3581791498-1105
Surname                : Robbins
UserPrincipalName      : µ@mikefrobbins.com
```

Summary

In this chapter, you've learned how to determine what type of object a command produces, what properties and methods are available for a command, and how to work with commands that limit the properties returned by default.

Review

1. What type of object does the `Get-Process` cmdlet produce?
2. How do you determine what the available properties are for a command?
3. If a command exists for getting something but not for setting the same thing, what should you check for?
4. How can certain commands that don't return output by default be made to generate output?
5. What should you consider doing when prototyping a command producing a large amount of output?

References

- Get-Member[4]
- Viewing Object Structure (Get-Member)[5]
- about_Objects[6]
- about_Properties[7]
- about_Methods[8]
- No PowerShell Cmdlet to Start or Stop Something? Don't Forget to Check for Methods on the Get Cmdlets[9]

Next steps

In the next chapter, you'll learn about one-liners and the pipeline.

[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.utility/get-member
[5] https://learn.microsoft.com/powershell/scripting/samples/viewing-object-structure--get-member-
[6] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_objects
[7] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_properties
[8] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_methods
[9] https://mikefrobbins.com/2016/12/15/no-powershell-cmdlet-to-start-or-stop-something-dont-forget-to-check-for-methods-on-the-get-cmdlets/

Chapter 4 - One-Liners and the pipeline

When I started learning PowerShell, I initially relied on the Graphical User Interface (GUI) for tasks that seemed too complex for simple PowerShell commands. However, as I continued to learn, I improved my skills and moved from basic one-liners to creating scripts, functions, and modules. It's important to remember that feeling overwhelmed by advanced examples online is normal. No one starts as an expert in PowerShell; we all start as beginners.

For those who primarily use the GUI for administrative tasks, install the management tools on your administrative workstation to remotely manage your servers. Whether your server uses a GUI or the Server Core OS installation, this approach is beneficial. It's a practical way to familiarize yourself with remote server management in preparation for performing administrative tasks with PowerShell.

As with the previous chapters, try these concepts in your lab environment.

One-Liners

A PowerShell one-liner is one continuous pipeline. It's a common misconception that a command on one physical line is a PowerShell one-liner, but this is'nt always true.

For instance, consider the following example: the command extends over multiple physical lines, yet it's a PowerShell one-liner because it forms a continuous pipeline. Line-breaking a lengthy one-liner at the pipe symbol, a natural breaking point in PowerShell, is recommended to enhance readability and clarity. This strategic use of line breaks improves readability without disrupting the flow of the pipeline.

```
Get-Service |
    Where-Object CanPauseAndContinue -eq $true |
    Select-Object -Property *
```

```
Name                  : LanmanWorkstation
RequiredServices      : {NSI, MRxSmb20, Bowser}
CanPauseAndContinue   : True
CanShutdown           : False
CanStop               : True
DisplayName           : Workstation
DependentServices     : {SessionEnv, Netlogon}
MachineName           : .
ServiceName           : LanmanWorkstation
ServicesDependedOn    : {NSI, MRxSmb20, Bowser}
ServiceHandle         :
Status                : Running
ServiceType           : Win32OwnProcess, Win32ShareProcess
StartType             : Automatic
Site                  :
Container             :

Name                  : Netlogon
RequiredServices      : {LanmanWorkstation}
CanPauseAndContinue   : True
CanShutdown           : False
CanStop               : True
DisplayName           : Netlogon
DependentServices     : {}
MachineName           : .
ServiceName           : Netlogon
ServicesDependedOn    : {LanmanWorkstation}
ServiceHandle         :
Status                : Running
ServiceType           : Win32ShareProcess
StartType             : Automatic
Site                  :
Container             :

Name                  : vmicheartbeat
```

```
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Heartbeat Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmicheartbeat
ServicesDependedOn  : {}
ServiceHandle       :
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : vmickvpexchange
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Data Exchange Service
DependentServices   : {}
MachineName         : .
ServiceName         : vmickvpexchange
ServicesDependedOn  : {}
ServiceHandle       :
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : vmicrdv
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : False
CanStop             : True
DisplayName         : Hyper-V Remote Desktop Virtualization Service
DependentServices   : {}
```

```
MachineName          : .
ServiceName          : vmicrdv
ServicesDependedOn   : {}
ServiceHandle        :
Status               : Running
ServiceType          : Win32OwnProcess, Win32ShareProcess
StartType            : Manual
Site                 :
Container            :

Name                 : vmicshutdown
RequiredServices     : {}
CanPauseAndContinue  : True
CanShutdown          : False
CanStop              : True
DisplayName          : Hyper-V Guest Shutdown Service
DependentServices    : {}
MachineName          : .
ServiceName          : vmicshutdown
ServicesDependedOn   : {}
ServiceHandle        :
Status               : Running
ServiceType          : Win32OwnProcess, Win32ShareProcess
StartType            : Manual
Site                 :
Container            :

Name                 : vmicvss
RequiredServices     : {}
CanPauseAndContinue  : True
CanShutdown          : False
CanStop              : True
DisplayName          : Hyper-V Volume Shadow Copy Requestor
DependentServices    : {}
MachineName          : .
ServiceName          : vmicvss
ServicesDependedOn   : {}
ServiceHandle        :
Status               : Running
ServiceType          : Win32OwnProcess, Win32ShareProcess
```

```
StartType           : Manual
Site                :
Container           :

Name                : webthreatdefsvc
RequiredServices    : {RpcSs, wtd}
CanPauseAndContinue : True
CanShutdown         : True
CanStop             : True
DisplayName         : Web Threat Defense Service
DependentServices   : {}
MachineName         : .
ServiceName         : webthreatdefsvc
ServicesDependedOn  : {RpcSs, wtd}
ServiceHandle       :
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Manual
Site                :
Container           :

Name                : webthreatdefusersvc_644de
RequiredServices    : {}
CanPauseAndContinue : True
CanShutdown         : True
CanStop             : True
DisplayName         : Web Threat Defense User Service_644de
DependentServices   : {}
MachineName         : .
ServiceName         : webthreatdefusersvc_644de
ServicesDependedOn  : {}
ServiceHandle       :
Status              : Running
ServiceType         : 240
StartType           : Automatic
Site                :
Container           :

Name                : Winmgmt
RequiredServices    : {RPCSS}
```

```
CanPauseAndContinue : True
CanShutdown         : True
CanStop             : True
DisplayName         : Windows Management Instrumentation
DependentServices   : {}
MachineName         : .
ServiceName         : Winmgmt
ServicesDependedOn  : {RPCSS}
ServiceHandle       :
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Automatic
Site                :
Container           :
```

Natural line breaks can occur at commonly used characters, including comma (`,`) and opening brackets (`[`), braces (`{`), and parenthesis (`(`). Others that aren't so common include the semicolon (`;`), equals sign (`=`), and both opening single and double quotes (`'`,`"`).

Using the backtick (`` ` ``) or grave accent character as a line continuation is controversial. It's best to avoid it if possible. Using a backtick following a natural line break character is a common mistake. This redundancy is unnecessary and can clutter the code.

The commands in the following example execute correctly from the PowerShell console. However, attempting to run them in the console pane of the PowerShell Integrated Scripting Environment (ISE) results in an error. This difference occurs because, unlike the PowerShell console, the console pane of the ISE doesn't automatically anticipate the continuation of a command onto the next line. To prevent this issue, press `Shift+Enter` in the console pane of the ISE instead of `Enter` when you need to extend a command across multiple lines. This key combination signals to the ISE that the command is continuing on the following line, preventing the execution that leads to errors.

```
Get-Service -Name w32time |
    Select-Object -Property *
```

```
Name                : w32time
RequiredServices    : {}
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
DisplayName         : Windows Time
DependentServices   : {}
MachineName         : .
ServiceName         : w32time
ServicesDependedOn  : {}
ServiceHandle       :
Status              : Running
ServiceType         : Win32OwnProcess, Win32ShareProcess
StartType           : Manual
Site                :
Container           :
```

This next example doesn't qualify as a PowerShell one-liner because it's not one continuous pipeline. Instead, it's two separate commands placed on a single line, separated by a semicolon. This semicolon indicates the end of one command and the beginning of another.

```
$Service = 'w32time'; Get-Service -Name $Service
```

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time
```

Many programming and scripting languages require a semicolon at the end of each line. However, in PowerShell, semicolons at the end of lines are unnecessary and not recommended. You should avoid them for cleaner and more readable code.

Filter Left

This chapter demonstrates how to filter the results of various commands. For instance, the `Get-Service` command is used with the **Name** parameter to display only the Windows Time service.

It's a best practice in PowerShell to filter the results as early as possible in the pipeline. Achieving this involves applying filters using parameters on the initial command, usually at the beginning of the pipeline. This is commonly referred to as *filtering left*.

To illustrate this concept, consider the following example: Use the **Name** parameter of `Get-Service` to filter the results at the beginning of the pipeline, returning only the details for the Windows Time service. This method demonstrates efficient data retrieval, ensuring you only return the necessary and relevant information.

```
Get-Service -Name w32time
```

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time
```

It's common to see online examples of a PowerShell command being piped to the `Where-Object` cmdlet to filter its results. This technique is inefficient if an earlier command in the pipeline has a parameter to perform the filtering.

```
Get-Service | Where-Object Name -eq w32time
```

```
Status   Name               DisplayName
------   ----               -----------
Running  W32Time            Windows Time
```

The first example demonstrates filtering directly at the source, returning results specifically for the Windows Time service. In contrast, the second example retrieves all services and then uses another command to filter the results. This might seem insignificant in small-scale scenarios, but consider a situation involving a large dataset, like Active Directory. It's inefficient to retrieve details for thousands of user accounts only to narrow them down to a small subset. Practice *filtering left* — applying filters as early as possible in the command sequence — even in seemingly trivial cases. This habit ensures efficiency in more complex scenarios where it becomes crucial.

Command sequencing for effective filtering

There's a misconception that the order of commands in PowerShell is inconsequential, but this is a misunderstanding. The sequence in which you arrange commands, particularly when filtering, is vital. For example, suppose you're using `Select-Object` to choose specific properties and `Where-Object` to filter. In that case, it's essential to apply the filtering first. Failing to do so means the necessary properties might not be available in the pipeline for filtering, leading to ineffective or erroneous results.

The following example fails to produce results because the **CanStopAndContinue** property is absent when `Select-Object` is piped to `Where-Object`. This is because the **CanStopAndContinue** property was not included in the selection made by `Select-Object`. Effectively, it has been excluded or filtered out.

```
Get-Service |
    Select-Object -Property DisplayName, Running, Status |
    Where-Object CanPauseAndContinue
```

Reversing the order of `Select-Object` and `Where-Object` produces the desired results.

```
Get-Service |
    Where-Object CanPauseAndContinue |
    Select-Object -Property DisplayName, Status
```

```
DisplayName                                   Status
-----------                                   ------
Workstation                                  Running
Netlogon                                     Running
Hyper-V Heartbeat Service                    Running
Hyper-V Data Exchange Service                Running
Hyper-V Remote Desktop Virtualization Service Running
Hyper-V Guest Shutdown Service               Running
Hyper-V Volume Shadow Copy Requestor         Running
Web Threat Defense Service                   Running
Web Threat Defense User Service_644de        Running
Windows Management Instrumentation           Running
```

The Pipeline

As you've seen in many examples throughout this book, you can often use the output of one command as input for another command. In Chapter 3, `Get-Member` was used to determine what type of object a command produces.

Chapter 3 also showed using the **ParameterType** parameter of `Get-Command` to determine what commands accepted that type of input. Depending on how thorough help for a command is, it may include an **INPUTS** and **OUTPUTS** section.

The **INPUTS** section indicates that you can pipe a **ServiceController** or a **String** object to the `Stop-Service` cmdlet.

```
help Stop-Service -Full
```

The following output has been abbreviated to show the relevant portion of the help.

```
...
INPUTS
    System.ServiceProcess.ServiceController
        You can pipe a service object to this cmdlet.

    System.String
        You can pipe a string that contains the name of a service to this
        cmdlet.


OUTPUTS
    None
        By default, this cmdlet returns no output.

    System.ServiceProcess.ServiceController
        When you use the PassThru parameter, this cmdlet returns a
        ServiceController object representing the service.
...
```

However, it doesn't specify which parameters accept that type of input. You can determine that information by checking the different parameters in the full version of the help for the `Stop-Service` cmdlet.

```
help Stop-Service -Full
```

Once again, only the relevant help is shown in the following results. Notice that the **DisplayName** parameter doesn't accept pipeline input. The **InputObject** parameter accepts pipeline input by value for **ServiceController** objects. The Name parameter accepts pipeline input by value for **String** objects and pipeline input **by property name**.

```
...
-DisplayName <System.String[]>
    Specifies the display names of the services to stop. Wildcard
    characters are permitted.

    Required?                    true
    Position?                    named
    Default value                None
    Accept pipeline input?       False
    Accept wildcard characters?  true

-InputObject <System.ServiceProcess.ServiceController[]>
    Specifies ServiceController objects that represent the services to
    stop. Enter a variable that contains the objects, or type a command
    or expression that gets the objects.

    Required?                    true
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByValue)
    Accept wildcard characters?  false

-Name <System.String[]>
    Specifies the service names of the services to stop. Wildcard
    characters are permitted.

    Required?                    true
    Position?                    0
    Default value                None
    Accept pipeline input?       True (ByPropertyName, ByValue)
```

```
    Accept wildcard characters?  true
...
```

When handling pipeline input, a parameter that accepts pipeline input both **by property name** and **by value** prioritizes **by value** binding first. If this method fails, it attempts to process pipeline input **by property name**. However, the term **by value** can be misleading. A more accurate description is **by type**.

For instance, if you pipe the output of a command that generates a **ServiceController** object to `Stop-Service`, this output is bound to the **InputObject** parameter. If the piped command produces a **String** object, it associates the output with the **Name** parameter. If you pipe output from a command that doesn't produce a **ServiceController** or **String** object, but does include a property named **Name**, `Stop-Service` binds the value of the **Name** property to its **Name** parameter.

Determine what type of output the `Get-Service` command produces.

```
Get-Service -Name w32time | Get-Member
```

```
   TypeName: System.ServiceProcess.ServiceController
```

`Get-Service` produces a **ServiceController** object type.

As shown in the help for `Stop-Service` cmdlet, the **InputObject** parameter accepts **ServiceController** objects through the pipeline **by value**. This implies that when you pipe the output of the `Get-Service` cmdlet to `Stop-Service`, the **ServiceController** objects produced by `Get-Service` bind to the **InputObject** parameter of `Stop-Service`.

```
Get-Service -Name w32time | Stop-Service
```

Now try string input. Pipe `w32time` to `Get-Member` to confirm that it's a string.

```
'w32time' | Get-Member
```

```
   TypeName: System.String
```

The PowerShell help documentation illustrates that when you pipe a string to `Stop-Service`, it binds to the **Name** parameter **by value**. Conduct a practical test to see this in action: pipe the string `w32time` to `Stop-Service`. This example demonstrates how `Stop-Service` processes the string `w32time` as the name of the service to stop. Execute the following command to observe this binding and command execution in action.

Notice that `w32time` is enclosed in single quotes. In PowerShell, it's a best practice to use single quotes for static strings, reserving double quotes for situations where the string contains variables that require expansion. Single quotes tell PowerShell to treat the content literally without parsing for variables. This approach not only ensures accuracy in how your script interprets the string but also enhances performance, as PowerShell expends less processing effort on strings within single quotes.

```
'w32time' | Stop-Service
```

Create a custom object to test pipeline input by property name for the **Name** parameter of `Stop-Service`.

```
$customObject = [pscustomobject]@{
    Name = 'w32time'
}
```

The contents of the **CustomObject** variable is a **PSCustomObject** object type and it contains a property named **Name**.

```
$customObject | Get-Member
```

```
   TypeName: System.Management.Automation.PSCustomObject

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
Name        NoteProperty string Name=w32time
```

When working with variables in PowerShell, such as `$customObject` in this example, it's important to use double quotes if you need to enclose the variable in quotes. Double quotes allow for variable expansion — PowerShell evaluates the variable and uses its value. For example, if you enclose `$customObject` in double quotes and pipe it to `Get-Member`, PowerShell processes the value of `$customObject`. In contrast, using single quotes would result in piping the literal string `$customObject` to `Get-Member`, not the value of the variable. This distinction is crucial for scenarios where you need to evaluate the value of variables.

When piping the contents of the `$customObject` variable to the `Stop-Service` cmdlet, the binding to the **Name** parameter occurs by **property name** rather than **by value**. This is because `$customObject` is an object that contains a property named **Name**. In this scenario, PowerShell identifies the **Name** property within `$customObject` and uses its value for the **Name** parameter of `Stop-Service`.

Create another custom object using a different property name, such as **Service**.

```
$customObject = [pscustomobject]@{
    Service = 'w32time'
}
```

An error occurs while trying to stop the `w32time` service by piping `$customObject` to `Stop-Service`. The pipeline binding fails because `$customObject` doesn't produce a **ServiceController** or **String** object and doesn't contain a **Name** property.

```
$customObject | Stop-Service
```

```
Stop-Service : Cannot find any service with service name
'@{Service=w32time}'.
At line:1 char:17
+ $customObject | Stop-Service
+                 ~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (@{Service=w32time}:String) [
   Stop-Service], ServiceCommandException
    + FullyQualifiedErrorId : NoServiceFoundForGivenName,Microsoft.PowerShe
   ll.Commands.StopServiceCommand
```

When the output property names of one command don't match the pipeline input requirements of another command, you can use `Select-Object` to rename the property names so they line up correctly.

In the following example, use `Select-Object` to rename the **Service** property to a property named **Name**.

At first glance, the syntax of this example might appear complex. However, it's essential to understand that more than copying and pasting code is required to learn the syntax. Instead, take the time to type out the code manually. This hands-on practice helps you remember the syntax, and it becomes more intuitive with repeated effort. Utilizing multiple monitors or split screen can also aid in the learning process. Display the example code on one screen while actively typing and experimenting with it on another. This setup makes it easier to follow along and enhances your understanding and retention of the syntax.

```
$customObject |
    Select-Object -Property @{name='Name';expression={$_.Service}} |
    Stop-Service
```

There are instances where you might need to use a parameter that doesn't accept pipeline input. In such cases, you can still use the output of one command as the input for another. First, capture and save the display names of a few specific Windows services into a text file. This step allows you to use the saved data as input for another command.

```
'Background Intelligent Transfer Service', 'Windows Time' |
    Out-File -FilePath $env:TEMP\services.txt
```

You can use parentheses to pass the output of one command as input for a parameter to another command.

```
Stop-Service -DisplayName (Get-Content -Path $env:TEMP\services.txt)
```

This concept is like the order of operations in Algebra. Just as mathematical operations within parentheses are computed first, the command enclosed in parentheses is executed before the outer command.

PowerShellGet

PowerShellGet, a module included with PowerShell version 5.0 and above, provides commands to discover, install, publish, and update PowerShell modules and other items in a NuGet repository. For those using PowerShell version 3.0 and above, PowerShellGet is also available as a separate download.

The PowerShell Gallery[1] is an online repository hosted by Microsoft, designed as a central hub for sharing PowerShell modules, scripts, and other resources. While Microsoft hosts the PowerShell Gallery, the PowerShell community contributes most of the available modules and scripts. Given the source of these modules and scripts, exercise caution before integrating any code from the PowerShell Gallery into your environment. Review and test downloads from the PowerShell Gallery in an isolated test environment. This process ensures the code is secure and reliable, functions as expected, and safeguards your environment from potential issues or vulnerabilities arising from unvetted code.

Many organizations opt to establish their own internal, private NuGet repository. This repository serves a dual purpose. First, it acts as a secure location for storing modules developed in-house, intended solely for internal use. Secondly, it provides a vetted collection of modules sourced externally, including those from public repositories. Companies typically undertake a thorough validation process before adding these external modules to the internal repository. This process is crucial to

[1] https://www.powershellgallery.com/

ensure the modules are free from malicious content and align with the security and operational standards of the company.

Use the `Find-Module` cmdlet that's part of the PowerShellGet module to find a module in the PowerShell Gallery that I wrote named MrToolkit.

```
Find-Module -Name MrToolkit
```

```
NuGet provider is required to continue
PowerShellGet requires NuGet provider version '2.8.5.201' or newer to
interact with NuGet-based repositories. The NuGet provider must be available
 in 'C:\Program Files\PackageManagement\ProviderAssemblies' or
'C:\Users\mikefrobbins\AppData\Local\PackageManagement\ProviderAssemblies'.
You can also install the NuGet provider by running 'Install-PackageProvider
-Name NuGet -MinimumVersion 2.8.5.201 -Force'. Do you want PowerShellGet to
install and import the NuGet provider now?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):

Version    Name                Repository         Description
-------    ----                ----------         -----------
1.3        MrToolkit           PSGallery          Misc PowerShell Tools
```

The first time you use one of the commands from the PowerShellGet module, you'll be prompted to install the NuGet provider.

To install the MrToolkit module, pipe the previous command to `Install-Module`.

```
Find-Module -Name MrToolkit | Install-Module -Scope CurrentUser
```

```
Untrusted repository
You are installing the modules from an untrusted repository. If you trust
this repository, change its InstallationPolicy value by running the
Set-PSRepository cmdlet. Are you sure you want to install the modules from
'https://www.powershellgallery.com/api/v2'?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "N"):y
```

Since the PowerShell Gallery is an untrusted repository, it prompts you to approve the installation of the module.

Finding pipeline input the easy way

The MrToolkit module includes a function named `Get-MrPipelineInput`. This cmdlet is designed to provide users with a convenient method for identifying the parameters of a command capable of accepting pipeline input. Specifically, it reveals three key aspects:

- Which parameters of a command can receive pipeline input
- The type of object each parameter accepts
- Whether they accept pipeline input **by value** or **by property name**

This capability dramatically simplifies the process of understanding and utilizing the pipeline capabilities of PowerShell commands.

The information previously obtained by analyzing the help documentation can be determined using this function.

```
Get-MrPipelineInput -Name Stop-Service | Format-List
```

```
ParameterName                   : InputObject
ParameterType                   : System.ServiceProcess.ServiceController[]
ValueFromPipeline               : True
ValueFromPipelineByPropertyName : False

ParameterName                   : Name
ParameterType                   : System.String[]
ValueFromPipeline               : True
ValueFromPipelineByPropertyName : True
```

Summary

In this chapter, you've learned about the intricacies of PowerShell one-liners. You've also learned that the physical line count of a command is irrelevant to its classification as a PowerShell one-liner. Additionally, you learned about key concepts such as filtering left, the pipeline, and PowerShellGet.

Review

1. What is a PowerShell one-liner?
2. What are some of the characters where natural line breaks can occur in PowerShell?
3. Why should you filter left?
4. What are the two ways that a PowerShell command can accept pipeline input?
5. Why shouldn't you trust commands found in the PowerShell Gallery?

References

- about_Pipelines[2]
- about_Command_Syntax[3]

[2] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_pipelines
[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_command_syntax

- about_Parameters[4]
- PowerShellGet: The BIG EASY way to discover, install, and update PowerShell modules[5]

Next steps

In the next chapter, you'll learn about formatting, aliases, providers, and comparison operators.

[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_parameters
[5] https://mikefrobbins.com/2015/04/23/powershellget-the-big-easy-way-to-discover-install-and-update-powershell-modules/

Chapter 5 - Formatting, aliases, providers, comparison

Prerequisites

The SQL Server PowerShell module is required by some of the examples shown in this chapter. For more information about the SqlServer PowerShell module and installation instructions, see the SQL Server PowerShell overview[1]. It's also used in subsequent chapters. Download and install it on your Windows lab environment computer.

Format Right

In Chapter 4, you learned to filter as far to the left as possible. The rule for manually formatting a command's output is similar to that rule except it needs to occur as far to the right as possible.

The most common format commands are `Format-Table` and `Format-List`. `Format-Wide` and `Format-Custom` can also be used, but are less common.

As mentioned in Chapter 3, a command that returns more than four properties defaults to a list unless custom formatting is used.

```
Get-Service -Name w32time |
    Select-Object -Property Status, DisplayName, Can*
```

[1] https://learn.microsoft.com/sql/powershell/download-sql-server-ps-module

```
Status              : Running
DisplayName         : Windows Time
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
```

Use the `Format-Table` cmdlet to manually override the formatting and show the output in a table instead of a list.

```
Get-Service -Name w32time |
    Select-Object -Property Status, DisplayName, Can* |
    Format-Table
```

```
Status  DisplayName  CanPauseAndContinue CanShutdown CanStop
------  -----------  ------------------- ----------- -------
Running Windows Time               False        True    True
```

The default output for `Get-Service` is three properties in a table.

```
Get-Service -Name w32time
```

```
Status   Name               DisplayName
------   ----               -----------
Running  w32time            Windows Time
```

Use the `Format-List` cmdlet to override the default formatting and return the results in a list.

```
Get-Service -Name w32time | Format-List
```

Notice that simply piping `Get-Service` to `Format-List` made it return additional properties. This doesn't occur with every command because of the way the formatting for that particular command is set up behind the scenes.

```
Name                : w32time
DisplayName         : Windows Time
Status              : Running
DependentServices   : {}
ServicesDependedOn  : {}
CanPauseAndContinue : False
CanShutdown         : True
CanStop             : True
ServiceType         : Win32OwnProcess, Win32ShareProcess
```

The number one thing to be aware of with the format cmdlets is they produce format objects that are different than normal objects in PowerShell.

```
Get-Service -Name w32time | Format-List | Get-Member
```

```
   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatStartData

Name                                MemberType Definition
----                                ---------- ----------
Equals                              Method     bool Equals(System.Obj...
GetHashCode                         Method     int GetHashCode()
GetType                             Method     type GetType()
ToString                            Method     string ToString()
autosizeInfo                        Property   Microsoft.PowerShell.C...
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef...
groupingEntry                       Property   Microsoft.PowerShell.C...
pageFooterEntry                     Property   Microsoft.PowerShell.C...
pageHeaderEntry                     Property   Microsoft.PowerShell.C...
shapeInfo                           Property   Microsoft.PowerShell.C...


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupStartData

Name                                MemberType Definition
----                                ---------- ----------
Equals                              Method     bool Equals(System.Obj...
GetHashCode                         Method     int GetHashCode()
GetType                             Method     type GetType()
```

```
ToString                                  Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef...
groupingEntry                             Property   Microsoft.PowerShell.C...
shapeInfo                                 Property   Microsoft.PowerShell.C...


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData

Name                                      MemberType Definition
----                                      ---------- ----------
Equals                                    Method     bool Equals(System.Obj...
GetHashCode                               Method     int GetHashCode()
GetType                                   Method     type GetType()
ToString                                  Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef...
formatEntryInfo                           Property   Microsoft.PowerShell.C...
outOfBand                                 Property   bool outOfBand {get;set;}
writeStream                               Property   Microsoft.PowerShell.C...


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.GroupEndData

Name                                      MemberType Definition
----                                      ---------- ----------
Equals                                    Method     bool Equals(System.Obj...
GetHashCode                               Method     int GetHashCode()
GetType                                   Method     type GetType()
ToString                                  Method     string ToString()
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef...
groupingEntry                             Property   Microsoft.PowerShell.C...


   TypeName: Microsoft.PowerShell.Commands.Internal.Format.FormatEndData

Name                                      MemberType Definition
----                                      ---------- ----------
Equals                                    Method     bool Equals(System.Obj...
GetHashCode                               Method     int GetHashCode()
GetType                                   Method     type GetType()
ToString                                  Method     string ToString()
```

```
ClassId2e4f51ef21dd47e99d3c952918aff9cd Property   string ClassId2e4f51ef...
groupingEntry                           Property   Microsoft.PowerShell.C...
```

What this means is format commands can't be piped to most other commands. They can be piped to some of the `Out-*` commands, but that's about it. This is why you want to perform any formatting at the very end of the line (format right).

Aliases

An alias in PowerShell is a shorter name for a command. PowerShell includes a set of built-in aliases and you can also define your own aliases.

The `Get-Alias` cmdlet is used to find aliases. If you already know the alias for a command, the **Name** parameter is used to determine what command the alias is associated with.

```
Get-Alias -Name gcm
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Alias           gcm -> Get-Command
```

Multiple aliases can be specified for the value of the **Name** parameter.

```
Get-Alias -Name gcm, gm
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Alias           gcm -> Get-Command
Alias           gm -> Get-Member
```

You'll often see the **Name** parameter omitted since it's a positional parameter.

```
Get-Alias gm
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Alias           gm -> Get-Member
```

If you want to find aliases for a command, you'll need to use the **Definition** parameter.

```
Get-Alias -Definition Get-Command, Get-Member
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Alias           gcm -> Get-Command
Alias           gm -> Get-Member
```

The **Definition** parameter can't be used positionally so it must be specified.

Aliases can save you a few keystrokes and they're fine when you're typing commands into the console. They shouldn't be used in scripts or any code that you're saving or sharing with others. As mentioned earlier in this book, using full cmdlet and parameter names is self-documenting and easier to understand.

Use caution when creating your own aliases because they'll only exist in your current PowerShell session on your computer.

Providers

A provider in PowerShell is an interface that allows file system like access to a datastore. There are a number of built-in providers in PowerShell.

```
Get-PSProvider
```

As you can see in the following results, there are built-in providers for the registry, aliases, environment variables, the file system, functions, variables, certificates, and WSMan.

```
Name                Capabilities                Drives
----                ------------                ------
Registry            ShouldProcess, Transactions {HKLM, HKCU}
Alias               ShouldProcess               {Alias}
Environment         ShouldProcess               {Env}
FileSystem          Filter, ShouldProcess, Cr... {C, D}
Function            ShouldProcess               {Function}
Variable            ShouldProcess               {Variable}
```

The actual drives that these providers use to expose their datastore can be determined with the `Get-PSDrive` cmdlet. The `Get-PSDrive` cmdlet not only displays drives exposed by providers, but it also displays Windows logical drives including drives mapped to network shares.

```
Get-PSDrive
```

```
Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
Alias                                  Alias
C                 18.56        107.62 FileSystem    C:\
Cert                                   Certificate   \
D                                      FileSystem    D:\
Env                                    Environment
Function                               Function
HKCU                                   Registry      HKEY_CURRENT_USER
HKLM                                   Registry      HKEY_LOCAL_MACHINE
Variable                               Variable
WSMan                                  WSMan
```

Third-party modules such as the Active Directory PowerShell module and the SQLServer PowerShell module both add their own PowerShell provider and PSDrive.

Import the Active Directory and SQL Server PowerShell modules.

```
Import-Module -Name ActiveDirectory, SQLServer
```

Check to see if any additional PowerShell providers were added.

```
Get-PSProvider
```

Notice that in the following set of results, two new PowerShell providers now exist, one for Active Directory and another one for SQL Server.

```
Name                 Capabilities                        Drives
----                 ------------                        ------
Registry             ShouldProcess, Transactions         {HKLM, HKCU}
Alias                ShouldProcess                       {Alias}
Environment          ShouldProcess                       {Env}
FileSystem           Filter, ShouldProcess, Credentials  {C, A, D}
Function             ShouldProcess                       {Function}
Variable             ShouldProcess                       {Variable}
ActiveDirectory      Include, Exclude, Filter, Shoul...  {AD}
SqlServer            Credentials                         {SQLSERVER}
```

A PSDrive for each of those modules was also added.

```
Get-PSDrive
```

```
Name           Used (GB)     Free (GB) Provider      Root
----           ---------     --------- --------      ----
A                                      FileSystem    A:\
AD                                     ActiveDire... //RootDSE/
Alias                                  Alias
C                 19.38        107.13 FileSystem    C:\
Cert                                   Certificate   \
D                                      FileSystem    D:\
Env                                    Environment
Function                               Function
HKCU                                   Registry      HKEY_CURRENT_USER
HKLM                                   Registry      HKEY_LOCAL_MACHINE
SQLSERVER                              SqlServer     SQLSERVER:\
Variable                               Variable
WSMan                                  WSMan
```

PSDrives can be accessed just like a traditional file system.

```
Get-ChildItem -Path Cert:\LocalMachine\CA
```

```
   PSParentPath: Microsoft.PowerShell.Security\Certificate::LocalMachine\CA

Thumbprint                                Subject
----------                                -------
FEE449EE0E3965A5246F000E87FDE2A065FD89D4  CN=Root Agency
D559A586669B08F46A30A133F8A9ED3D038E2EA8  OU=www.verisign.com/CPS Incorp....
109F1CAED645BB78B3EA2B94C0697C740733031C  CN=Microsoft Windows Hardware C...
```

Comparison Operators

PowerShell contains a number of comparison operators that are used to compare values or find values that match certain patterns. The following table contains a list of comparison operators in PowerShell.

All of the operators listed below are case-insensitive. Place a `c` in front of the operator to make it case-sensitive. For example, `-ceq` is the case-sensitive version of the `-eq` comparison operator.

Operator	Definition
`-eq`	Equal to
`-ne`	Not equal to
`-gt`	Greater than
`-ge`	Greater than or equal to
`-lt`	Less than
`-le`	Less than or equal to
`-Like`	Match using the * wildcard character
`-NotLike`	Does not match using the * wildcard character
`-Match`	Matches the specified regular expression
`-NotMatch`	Does not match the specified regular expression
`-Contains`	Determines if a collection contains a specified value

Operator	Definition
`-NotContains`	Determines if a collection does not contain a specific value
`-In`	Determines if a specified value is in a collection
`-NotIn`	Determines if a specified value is not in a collection
`-Replace`	Replaces the specified value

Proper case “PowerShell” is equal to lower case “powershell” using the equals comparison operator.

```
'PowerShell' -eq 'powershell'
```

```
True
```

It’s not equal using the case-sensitive version of the equals comparison operator.

```
'PowerShell' -ceq 'powershell'
```

```
False
```

The not equal comparison operator reverses the condition.

```
'PowerShell' -ne 'powershell'
```

```
False
```

Greater than, greater than or equal to, less than, and less than or equal all work with string or numeric values.

```
5 -gt 5
```

```
False
```

Using greater than or equal to instead of greater than with the previous example returns the **Boolean** true since five is equal to five.

```
5 -ge 5
```

```
True
```

Based on the results from the previous two examples, you can probably guess how both less than and less than or equal to work.

```
5 -lt 10
```

```
True
```

The `-Like` and `-Match` operators can be confusing, even for experienced PowerShell users. `-Like` is used with wildcard the characters `*` and `?` to perform "like" matches.

```
'PowerShell' -like '*shell'
```

```
True
```

`-Match` uses a regular expression to perform the matching.

```
'PowerShell' -match '^*.shell$'
```

```
True
```

Use the range operator to store the numbers 1 through 10 in a variable.

```
$Numbers = 1..10
```

Determine if the `$Numbers` variable includes 15.

```
$Numbers -contains 15
```

```
False
```

Determine if it includes the number 10.

```
$Numbers -contains 10
```

```
True
```

`-NotContains` reverses the logic to see if the `$Numbers` variable doesn't contain a value.

```
$Numbers -notcontains 15
```

```
True
```

The previous example returns the **Boolean** true because it's true that the `$Numbers` variable doesn't contain 15. It does however contain the number 10 so it's false when it's tested.

```
$Numbers -notcontains 10
```

```
False
```

The `-in` comparison operator was first introduced in PowerShell version 3.0. It's used to determine if a value is *in* an array. The `$Numbers` variable is an array since it contains multiple values.

```
15 -in $Numbers
```

```
False
```

In other words, `-in` performs the same test as the contains comparison operator except from the opposite direction.

```
10 -in $Numbers
```

```
True
```

15 isn't in the `$Numbers` array so false is returned in the following example.

```
15 -in $Numbers
```

```
False
```

Just like the `-contains` operator, `not` reverses the logic for the `-in` operator.

```
10 -notin $Numbers
```

```
False
```

The previous example returns false because the `$Numbers` array does include 10 and the condition was testing to determine if it didn't contain 10.

15 is "not in" the `$Numbers` array so it returns the **Boolean** true.

```
15 -notin $Numbers
```

```
True
```

The `-replace` operator does just want you would think. It's used to replace something. Specifying one value replaces that value with nothing. In the following example, I replace "Shell" with nothing.

```
'PowerShell' -replace 'Shell'
```

```
Power
```

If you want to replace a value with a different value, specify the new value after the pattern you want to replace. SQL Saturday in Baton Rouge is an event that I try to speak at every year. In the following example, I replace the word "Saturday" with the abbreviation "Sat".

```
'SQL Saturday - Baton Rouge' -Replace 'saturday','Sat'
```

```
SQL Sat - Baton Rouge
```

There are also methods like **Replace()** that can be used to replace things similar to the way the replace operator works. However, the `-Replace` operator is case-insensitive by default, and the **Replace()** method is case-sensitive.

```
'SQL Saturday - Baton Rouge'.Replace('saturday','Sat')
```

```
SQL Saturday - Baton Rouge
```

Notice that the word "Saturday" wasn't replaced in the previous example. This is because it was specified in a different case than the original. When the word "Saturday" is specified in the same case as the original, the **Replace()** method does replace it as expected.

```
'SQL Saturday - Baton Rouge'.Replace('Saturday','Sat')
```

```
SQL Sat - Baton Rouge
```

Be careful when using methods to transform data because you can run into unforeseen problems, such as failing the *Turkey Test*. For an example, see the blog article titled Using Pester to Test PowerShell Code with Other Cultures[2]. My recommendation is to use an operator instead of a method whenever possible to avoid these types of problems.

While the comparison operators can be used as shown in the previous examples, I normally find myself using them with the `Where-Object` cmdlet to perform some type of filtering.

Summary

In this chapter, you've learned a number of different topics to include Formatting Right, Aliases, Providers, and Comparison Operators.

Review

1. Why is it necessary to perform Formatting as far to the right as possible?
2. How do you determine what the actual cmdlet is for the `%` alias?
3. Why shouldn't you use aliases in scripts you save or code you share with others?
4. Perform a directory listing on the drives that are associated with one of the registry providers.
5. What's one of the main benefits of using the replace operator instead of the replace method?

[2] https://mikefrobbins.com/2015/10/22/using-pester-to-test-powershell-code-with-other-cultures/

References

- format-table[3]
- format-list[4]
- format-wide[5]
- about_Aliases[6]
- about_Providers[7]
- about_Comparison_Operators[8]
- about_Arrays[9]

Next steps

In the next chapter, you'll learn about flow control, scripting, loops, and conditional logic.

[3] https://learn.microsoft.com//powershell/module/microsoft.powershell.utility/format-table
[4] https://learn.microsoft.com//powershell/module/microsoft.powershell.utility/format-list
[5] https://learn.microsoft.com//powershell/module/microsoft.powershell.utility/format-wide
[6] https://learn.microsoft.com//powershell/module/microsoft.powershell.core/about/about_aliases
[7] https://learn.microsoft.com//powershell/module/microsoft.powershell.core/about/about_providers
[8] https://learn.microsoft.com//powershell/module/microsoft.powershell.core/about/about_comparison_operators
[9] https://learn.microsoft.com//powershell/module/microsoft.powershell.core/about/about_arrays

Chapter 6 - Flow control

Scripting

When you move from writing PowerShell one-liners to writing scripts, it sounds a lot more complicated than it really is. A script is nothing more than the same or similar commands that you would run interactively in the PowerShell console, except they're saved as a `.PS1` file. There are some scripting constructs that you may use such as a `foreach` loop instead of the `ForEach-Object` cmdlet. To beginners, the differences can be confusing especially when you consider that `foreach` is both a scripting construct and an alias for the `ForEach-Object` cmdlet.

Looping

One of the great things about PowerShell is, once you figure out how to do something for one item, it's almost as easy to do the same task for hundreds of items. Simply loop through the items using one of the many different types of loops in PowerShell.

ForEach-Object

`ForEach-Object` is a cmdlet for iterating through items in a pipeline such as with PowerShell one-liners. `ForEach-Object` streams the objects through the pipeline.

Although the **Module** parameter of `Get-Command` accepts multiple values that are strings, it only accepts them via pipeline input by property name or via parameter input. In the following scenario, if I want to pipe two strings by value to `Get-Command` for use with the **Module** parameter, I would need to use the `ForEach-Object` cmdlet.

```
'ActiveDirectory', 'SQLServer' |
    ForEach-Object {Get-Command -Module $_} |
    Group-Object -Property ModuleName -NoElement |
    Sort-Object -Property Count -Descending
```

```
Count Name
----- ----
  147 ActiveDirectory
   82 SqlServer
```

In the previous example, `$_` is the current object. Beginning with PowerShell version 3.0, `$PSItem` can be used instead of `$_`. But I find that most experienced PowerShell users still prefer using `$_` since it's backward compatible and less to type.

When using the `foreach` keyword, you must store all of the items in memory before iterating through them, which could be difficult if you don't know how many items you're working with.

```
$ComputerName = 'DC01', 'WEB01'
foreach ($Computer in $ComputerName) {
    Get-ADComputer -Identity $Computer
}
```

```
DistinguishedName : CN=DC01,OU=Domain Controllers,DC=mikefrobbins,DC=com
DNSHostName       : dc01.mikefrobbins.com
Enabled           : True
Name              : DC01
ObjectClass       : computer
ObjectGUID        : c38da20c-a484-469d-ba4c-bab3fb71ae8e
SamAccountName    : DC01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1001
UserPrincipalName :

DistinguishedName : CN=WEB01,CN=Computers,DC=mikefrobbins,DC=com
DNSHostName       : web01.mikefrobbins.com
Enabled           : True
Name              : WEB01
```

```
ObjectClass       : computer
ObjectGUID        : 33aa530e-1e31-40d8-8c78-76a18b673c33
SamAccountName    : WEB01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1107
UserPrincipalName :
```

Many times a loop such as `foreach` or `ForEach-Object` is necessary. Otherwise you'll receive an error message.

```
Get-ADComputer -Identity 'DC01', 'WEB01'
```

```
Get-ADComputer : Cannot convert 'System.Object[]' to the type
'Microsoft.ActiveDirectory.Management.ADComputer' required by parameter
'Identity'. Specified method is not supported.
At line:1 char:26
+ Get-ADComputer -Identity 'DC01', 'WEB01'
+                          ~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Get-ADComputer], Parame
   terBindingException
    + FullyQualifiedErrorId : CannotConvertArgument,Microsoft.ActiveDirecto
   ry.Management.Commands.GetADComputer
```

Other times, you can get the same results while eliminating the loop altogether. Consult the cmdlet help to understand your options.

```
'DC01', 'WEB01' | Get-ADComputer
```

```
DistinguishedName : CN=DC01,OU=Domain Controllers,DC=mikefrobbins,DC=com
DNSHostName       : dc01.mikefrobbins.com
Enabled           : True
Name              : DC01
ObjectClass       : computer
ObjectGUID        : c38da20c-a484-469d-ba4c-bab3fb71ae8e
SamAccountName    : DC01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1001
UserPrincipalName :

DistinguishedName : CN=WEB01,CN=Computers,DC=mikefrobbins,DC=com
DNSHostName       : web01.mikefrobbins.com
Enabled           : True
Name              : WEB01
ObjectClass       : computer
ObjectGUID        : 33aa530e-1e31-40d8-8c78-76a18b673c33
SamAccountName    : WEB01$
SID               : S-1-5-21-2989741381-570885089-3319121794-1107
UserPrincipalName :
```

As you can see in the previous examples, the **Identity** parameter for `Get-ADComputer` only accepts a single value when provided via parameter input, but it allows for multiple items when the input is provided via pipeline input.

For

A `for` loop iterates while a specified condition is true. The `for` loop is not something that I use often, but it does have its uses.

```
for ($i = 1; $i -lt 5; $i++) {
    Write-Output "Sleeping for $i seconds"
    Start-Sleep -Seconds $i
}
```

```
Sleeping for 1 seconds
Sleeping for 2 seconds
Sleeping for 3 seconds
Sleeping for 4 seconds
```

In the previous example, the loop will iterate four times by starting off with the number one and continue as long as the counter variable `$i` is less than 5. It will sleep for a total of 10 seconds.

Do

There are two different `do` loops in PowerShell. `Do Until` runs while the specified condition is false.

```
$number = Get-Random -Minimum 1 -Maximum 10
do {
    $guess = Read-Host -Prompt "What's your guess?"
    if ($guess -lt $number) {
        Write-Output 'Too low!'
    } elseif ($guess -gt $number) {
        Write-Output 'Too high!'
    }
}
until ($guess -eq $number)
```

```
What's your guess?: 1
Too low!
What's your guess?: 2
Too low!
What's your guess?: 3
```

The previous example is a numbers game that continues until the value you guess equals the same number that the `Get-Random` cmdlet generated.

`Do While` is just the opposite. It runs as long as the specified condition evaluates to true.

```
$number = Get-Random -Minimum 1 -Maximum 10
do {
    $guess = Read-Host -Prompt "What's your guess?"
    if ($guess -lt $number) {
        Write-Output 'Too low!'
    } elseif ($guess -gt $number) {
        Write-Output 'Too high!'
    }
}
while ($guess -ne $number)
```

```
What's your guess?: 1
Too low!
What's your guess?: 2
Too low!
What's your guess?: 3
Too low!
What's your guess?: 4
```

The same results are achieved with a `Do While` loop by reversing the test condition to not equals.

`Do` loops always run at least once because the condition is evaluated at the end of the loop.

While

Similar to the `Do While` loop, a `While` loop runs as long as the specified condition is true. The difference however, is that a `While` loop evaluates the condition at the top of the loop before any code is run. So it doesn't run if the condition evaluates to false.

```
$date = Get-Date -Date 'November 22'
while ($date.DayOfWeek -ne 'Thursday') {
    $date = $date.AddDays(1)
}
Write-Output $date
```

```
Thursday, November 23, 2017 12:00:00 AM
```

The previous example calculates what day Thanksgiving Day is on in the United States. It's always on the fourth Thursday of November. So the loop starts with the 22nd day of November and adds a day while the day of the week isn't equal to Thursday. If the 22nd is a Thursday, the loop doesn't run at all.

Break, Continue, and Return

`Break` is designed to break out of a loop. It's also commonly used with the `switch` statement.

```
for ($i = 1; $i -lt 5; $i++) {
    Write-Output "Sleeping for $i seconds"
    Start-Sleep -Seconds $i
    break
}
```

```
Sleeping for 1 seconds
```

The `break` statement shown in the previous example causes the loop to exit on the first iteration.

Continue is designed to skip to the next iteration of a loop.

```
while ($i -lt 5) {
    $i += 1
    if ($i -eq 3) {
        continue
    }
    Write-Output $i
}
```

```
1
2
4
5
```

The previous example will output the numbers 1, 2, 4, and 5. It skips number 3 and continues with the next iteration of the loop. Similar to `break`, `continue` breaks out of the loop except only for the current iteration. Execution continues with the next iteration instead of breaking out of the loop and stopping.

Return is designed to exit out of the existing scope.

```
$number = 1..10
foreach ($n in $number) {
    if ($n -ge 4) {
        Return $n
    }
}
```

```
4
```

Notice that in the previous example, return outputs the first result and then exits out of the loop. A more thorough explanation of the result statement can be found in one of my blog articles: "The PowerShell return keyword"[1].

Summary

In this chapter, you've learned about the different types of loops that exist in PowerShell.

[1] https://mikefrobbins.com/2015/07/23/the-powershell-return-keyword/

Review

1. What is the difference in the `ForEach-Object` cmdlet and the foreach scripting construct?
2. What is the primary advantage of using a While loop instead of a Do While or Do Until loop.
3. How do the break and continue statements differ?

References

- ForEach-Object[2]
- about_ForEach[3]
- about_For[4]
- about_Do[5]
- about_While[6]
- about_Break[7]
- about_Continue[8]
- about_Return[9]

[2] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/foreach-object
[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_foreach
[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_for
[5] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_do
[6] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_while
[7] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_break
[8] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_continue
[9] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_return

Chapter 7 - Working with WMI

WMI and CIM

Windows PowerShell ships by default with cmdlets for working with other technologies such as Windows Management Instrumentation (WMI). The WMI cmdlets are deprecated and are not available in PowerShell 6+, but are covered here as you may encounter them in older scripts running on Windows PowerShell. For new development, use the CIM cmdlets instead.

There are several native WMI cmdlets that exist in PowerShell without having to install any additional software or modules. `Get-Command` can be used to determine what WMI cmdlets exist in Windows PowerShell. The following results are from my Windows 11 lab environment computer that is running PowerShell version 5.1. Your results may differ depending on what PowerShell version you're running.

```
Get-Command -Noun WMI*
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Cmdlet          Get-WmiObject                                      3.1.0.0
Cmdlet          Invoke-WmiMethod                                   3.1.0.0
Cmdlet          Register-WmiEvent                                  3.1.0.0
Cmdlet          Remove-WmiObject                                   3.1.0.0
Cmdlet          Set-WmiInstance                                    3.1.0.0
```

Common Information Model (CIM) cmdlets were introduced in PowerShell version 3.0. The CIM cmdlets are designed so they can be used on both Windows and non-Windows machines.

The CIM cmdlets are all contained within a module. To obtain a list of the CIM cmdlets, use `Get-Command` with the **Module** parameter as shown in the following example.

```
Get-Command -Module CimCmdlets
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Cmdlet          Export-BinaryMiLog                                 1.0.0.0
Cmdlet          Get-CimAssociatedInstance                          1.0.0.0
Cmdlet          Get-CimClass                                       1.0.0.0
Cmdlet          Get-CimInstance                                    1.0.0.0
Cmdlet          Get-CimSession                                     1.0.0.0
Cmdlet          Import-BinaryMiLog                                 1.0.0.0
Cmdlet          Invoke-CimMethod                                   1.0.0.0
Cmdlet          New-CimInstance                                    1.0.0.0
Cmdlet          New-CimSession                                     1.0.0.0
Cmdlet          New-CimSessionOption                               1.0.0.0
Cmdlet          Register-CimIndicationEvent                        1.0.0.0
Cmdlet          Remove-CimInstance                                 1.0.0.0
Cmdlet          Remove-CimSession                                  1.0.0.0
Cmdlet          Set-CimInstance                                    1.0.0.0
```

The CIM cmdlets still allow you to work with WMI so don't be confused when someone makes the statement "When I query WMI with the PowerShell CIM cmdlets..."

As I previously mentioned, WMI is a separate technology from PowerShell and you're just using the CIM cmdlets for accessing WMI. You may find an old VBScript that uses WMI Query Language (WQL) to query WMI such as in the following example.

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colBIOS = objWMIService.ExecQuery _
    ("Select * from Win32_BIOS")

For each objBIOS in colBIOS
    Wscript.Echo "Manufacturer: " & objBIOS.Manufacturer
    Wscript.Echo "Name: " & objBIOS.Name
```

```
    Wscript.Echo "Serial Number: " & objBIOS.SerialNumber
    Wscript.Echo "SMBIOS Version: " & objBIOS.SMBIOSBIOSVersion
    Wscript.Echo "Version: " & objBIOS.Version
Next
```

You can take the WQL query from that VBScript and use it with the `Get-CimInstance` cmdlet without any modifications.

```
Get-CimInstance -Query 'Select * from Win32_BIOS'
```

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 3810-1995-1654-4615-2295-2755-89
Version           : VRTUAL - 4001628
```

That's not how I typically query WMI with PowerShell. But it does work and allows you to easily migrate existing VBScripts to PowerShell. When I start out writing a one-liner to query WMI, I use the following syntax.

```
Get-CimInstance -ClassName Win32_BIOS
```

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 3810-1995-1654-4615-2295-2755-89
Version           : VRTUAL - 4001628
```

If I only want the serial number, I can pipe the output to `Select-Object` and specify only the **SerialNumber** property.

```
Get-CimInstance -ClassName Win32_BIOS |
    Select-Object -Property SerialNumber
```

```
SerialNumber
------------
3810-1995-1654-4615-2295-2755-89
```

By default, there are several properties that are retrieved behind the scenes that are never used. It may not matter much when querying WMI on the local computer. But once you start querying remote computers, it's not only additional processing time to return that information, but also additional unnecessary information to have to pull across the network. `Get-CimInstance` has a **Property** parameter that limits the information that's retrieved. This makes the query to WMI more efficient.

```
Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber |
    Select-Object -Property SerialNumber
```

```
SerialNumber
------------
3810-1995-1654-4615-2295-2755-89
```

The previous results returned an object. To return a simple string, use the **ExpandProperty** parameter.

```
Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber |
    Select-Object -ExpandProperty SerialNumber
```

```
3810-1995-1654-4615-2295-2755-89
```

You could also use the dotted style of syntax to return a simple string. This eliminates the need to pipe to `Select-Object`.

```
(Get-CimInstance -ClassName Win32_BIOS -Property SerialNumber).SerialNumber
```

```
3810-1995-1654-4615-2295-2755-89
```

Query Remote Computers with the CIM cmdlets

I'm still running PowerShell as a local admin who is a domain user. When I try to query information from a remote computer using the `Get-CimInstance` cmdlet, I receive an access denied error message.

```
Get-CimInstance -ComputerName dc01 -ClassName Win32_BIOS
```

```
Get-CimInstance : Access is denied.
At line:1 char:1
+ Get-CimInstance -ComputerName dc01 -ClassName Win32_BIOS
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : PermissionDenied: (root\cimv2:Win32_BIOS:Stri
   ng) [Get-CimInstance], CimException
    + FullyQualifiedErrorId : HRESULT 0x80070005,Microsoft.Management.Infra
   structure.CimCmdlets.GetCimInstanceCommand
    + PSComputerName        : dc01
```

Many people have security concerns when it comes to PowerShell, but the truth is you have exactly the same permissions in PowerShell as you do in the GUI. No more and no less. The problem in the previous example is that the user running PowerShell doesn't have rights to query WMI information from the DC01 server. I could relaunch PowerShell as a domain administrator since `Get-CimInstance` doesn't have a **Credential** parameter. But, trust me, that isn't a good idea because then anything that I run from PowerShell would be running as a domain admin. That could be dangerous from a security standpoint depending on the situation.

Using the principle of least privilege, I elevate to my domain admin account on a per command basis using the **Credential** parameter, if a command has one. `Get-CimInstance` doesn't have a **Credential** parameter so the solution in this scenario is to create a **CimSession** first. Then I use the **CimSession** instead of a computer name to query WMI on the remote computer.

```
$CimSession = New-CimSession -ComputerName dc01 -Credential (Get-Credential)


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

The CIM session was stored in a variable named `$CimSession`. Notice that I also specified the `Get-Credential` cmdlet in parentheses so that it executes first, prompting me for alternate credentials, before creating the new session. I'll show you another more efficient way to specify alternate credentials later in this chapter, but it's important to understand this basic concept before making it more complicated.

The CIM session created in the previous example can now be used with the `Get-CimInstance` cmdlet to query the BIOS information from WMI on the remote computer.

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS


SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 0986-6980-3916-0512-6608-8243-13
Version           : VRTUAL - 4001628
PSComputerName    : dc01
```

There are several additional benefits to using CIM sessions instead of just specifying a computer name. When running multiple queries to the same computer, using a CIM session is more efficient than using the computer name for each query. Creating a CIM session only sets up the connection once. Then, multiple queries use that same session to retrieve information. Using the computer name requires the cmdlets to set up and tear down the connection with each individual query.

The `Get-CimInstance` cmdlet uses the WSMan protocol by default, which means the remote computer needs PowerShell version 3.0 or higher to connect. It's actually not the PowerShell version that matters, it's the stack version. The stack version can be determined using the `Test-WSMan` cmdlet. It needs to be version 3.0. That's the version you'll find with PowerShell version 3.0 and higher.

```
Test-WSMan -ComputerName dc01
```

```
wsmid           : http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentit
                  y.xsd
ProtocolVersion : http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd
ProductVendor   : Microsoft Corporation
ProductVersion  : OS: 0.0.0 SP: 0.0 Stack: 3.0
```

The older WMI cmdlets use the DCOM protocol, which is compatible with older versions of Windows. But DCOM is typically blocked by the firewall on newer versions of Windows. The `New-CimSessionOption` cmdlet allows you to create a DCOM protocol connection for use with `New-CimSession`. This allows the `Get-CimInstance` cmdlet to be used to communicate with versions of Windows as old as Windows Server 2000. This also means that PowerShell is not required on the remote computer when using the `Get-CimInstance` cmdlet with a CimSession that's configured to use the DCOM protocol.

Create the DCOM protocol option using the `New-CimSessionOption` cmdlet and store it in a variable.

```
$DCOM = New-CimSessionOption -Protocol Dcom
```

For efficiency, you can store your domain administrator or elevated credentials in a variable so you don't have to constantly enter them for each command.

```
$Cred = Get-Credential
```

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

I have a server named SQL03 that runs Windows Server 2008 (non-R2). It's the newest Windows Server operating system that doesn't have PowerShell installed by default.

Create a **CimSession** to SQL03 using the DCOM protocol.

```
$CimSession = New-CimSession -ComputerName sql03 -SessionOption $DCOM -Credenti\
al $Cred
```

Notice in the previous command, this time I specified the variable named `$Cred` as the value for the **Credential** parameter instead of having to enter them manually again.

The output of the query is the same regardless of the underlying protocol being used.

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS
```

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 7237-7483-8873-8926-7271-5004-86
Version           : VRTUAL - 4001628
PSComputerName    : sql03
```

The `Get-CimSession` cmdlet is used to see what **CimSessions** are currently connected and what protocols they're using.

```
Get-CimSession
```

```
Id           : 1
Name         : CimSession1
InstanceId   : 80742787-e38e-41b1-a7d7-fa1369cf1402
ComputerName : dc01
Protocol     : WSMAN

Id           : 2
Name         : CimSession2
InstanceId   : 8fcabd81-43cf-4682-bd53-ccce1e24aecb
ComputerName : sql03
Protocol     : DCOM
```

Retrieve and store both of the previously created **CimSessions** in a variable named `$CimSession`.

```
$CimSession = Get-CimSession
```

Query both of the computers with one command, one using the WSMan protocol and the other one with DCOM.

```
Get-CimInstance -CimSession $CimSession -ClassName Win32_BIOS
```

```
SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 0986-6980-3916-0512-6608-8243-13
Version           : VRTUAL - 4001628
PSComputerName    : dc01

SMBIOSBIOSVersion : 090006
Manufacturer      : American Megatrends Inc.
Name              : Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz
SerialNumber      : 7237-7483-8873-8926-7271-5004-86
Version           : VRTUAL - 4001628
PSComputerName    : sql03
```

I've written numerous blog articles about the WMI and CIM cmdlets. One of the most useful ones is about a function that I created to automatically determine if WSMan or DCOM should be used and set up the CIM session automatically without having to figure out which one manually. That blog article is titled PowerShell Function to Create CimSessions to Remote Computers with Fallback to Dcom[1].

When you're finished with the CIM sessions, you should remove them with the `Remove-CimSession` cmdlet. To remove all CIM sessions, simply pipe `Get-CimSession` to `Remove-CimSession`.

```
Get-CimSession | Remove-CimSession
```

[1] https://mikefrobbins.com/2014/08/28/powershell-function-to-create-cimsessions-to-remote-computers-with-fallback-to-dcom/

Summary

In this chapter, you've learned about using PowerShell to work with WMI on both local and remote computers. You've also learned how to use the CIM cmdlets to work with remote computers with both the WSMan or DCOM protocol.

Review

1. What is the difference in the WMI and CIM cmdlets?
2. By default, what protocol does the `Get-CimInstance` cmdlet use?
3. What are some of the benefits of using a CIM session instead of specifying a computer name with `Get-CimInstance`?
4. How do you specify an alternate protocol other than the default one for use with `Get-CimInstance`?
5. How do you close or remove CIM sessions?

References

- about_WMI[2]
- about_WMI_Cmdlets[3]
- about_WQL[4]
- CimCmdlets Module[5]
- Video: Using CIM Cmdlets and CIM Sessions[6]

[2] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_wmi
[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_wmi_cmdlets
[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_wql
[5] https://learn.microsoft.com/powershell/module/cimcmdlets/
[6] https://mikefrobbins.com/2013/09/12/phillyposh-user-group-meeting-presentation-follow-up-powershell-second-hop-problem-with-cimsessions/

Chapter 8 - PowerShell Remoting

PowerShell has many different ways to run commands against remote computers. In the last chapter, you saw how to remotely query WMI using the CIM cmdlets. PowerShell also includes several cmdlets that have a built-in **ComputerName** parameter.

As shown in the following example, `Get-Command` can be used with the **ParameterName** parameter to determine what commands have a **ComputerName** parameter.

```
Get-Command -ParameterName ComputerName
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Cmdlet          Add-Computer                                       3.1.0.0
Cmdlet          Clear-EventLog                                     3.1.0.0
Cmdlet          Connect-PSSession                                  3.0.0.0
Cmdlet          Enter-PSSession                                    3.0.0.0
Cmdlet          Get-CimAssociatedInstance                          1.0.0.0
Cmdlet          Get-CimClass                                       1.0.0.0
Cmdlet          Get-CimInstance                                    1.0.0.0
Cmdlet          Get-CimSession                                     1.0.0.0
Cmdlet          Get-EventLog                                       3.1.0.0
Cmdlet          Get-HotFix                                         3.1.0.0
Cmdlet          Get-Process                                        3.1.0.0
Cmdlet          Get-PSSession                                      3.0.0.0
Cmdlet          Get-Service                                        3.1.0.0
Cmdlet          Get-WmiObject                                      3.1.0.0
Cmdlet          Invoke-CimMethod                                   1.0.0.0
Cmdlet          Invoke-Command                                     3.0.0.0
Cmdlet          Invoke-WmiMethod                                   3.1.0.0
Cmdlet          Limit-EventLog                                     3.1.0.0
```

```
Cmdlet          New-CimInstance                                    1.0.0.0
Cmdlet          New-CimSession                                     1.0.0.0
Cmdlet          New-EventLog                                       3.1.0.0
Cmdlet          New-PSSession                                      3.0.0.0
Cmdlet          Receive-Job                                        3.0.0.0
Cmdlet          Receive-PSSession                                  3.0.0.0
Cmdlet          Register-CimIndicationEvent                        1.0.0.0
Cmdlet          Register-WmiEvent                                  3.1.0.0
Cmdlet          Remove-CimInstance                                 1.0.0.0
Cmdlet          Remove-CimSession                                  1.0.0.0
Cmdlet          Remove-Computer                                    3.1.0.0
Cmdlet          Remove-EventLog                                    3.1.0.0
Cmdlet          Remove-PSSession                                   3.0.0.0
Cmdlet          Remove-WmiObject                                   3.1.0.0
Cmdlet          Rename-Computer                                    3.1.0.0
Cmdlet          Restart-Computer                                   3.1.0.0
Cmdlet          Send-MailMessage                                   3.1.0.0
Cmdlet          Set-CimInstance                                    1.0.0.0
Cmdlet          Set-Service                                        3.1.0.0
Cmdlet          Set-WmiInstance                                    3.1.0.0
Cmdlet          Show-EventLog                                      3.1.0.0
Cmdlet          Stop-Computer                                      3.1.0.0
Cmdlet          Test-Connection                                    3.1.0.0
Cmdlet          Write-EventLog                                     3.1.0.0
```

Commands such as `Get-Process` and `Get-Hotfix` have a **ComputerName** parameter. This isn't the long-term direction that Microsoft is heading for running commands against remote computers. Even if you find a command that has a **ComputerName** parameter, chances are that you'll need to specify alternate credentials and it won't have a **Credential** parameter. And if you decided to run PowerShell from an elevated account, a firewall between you and the remote computer can block the request.

To use the PowerShell remoting commands that are demonstrated in this chapter, PowerShell remoting must be enabled on the remote computer. Use the `Enable-PSRemoting` cmdlet to enable PowerShell remoting.

```
Enable-PSRemoting
```

```
WinRM has been updated to receive requests.
WinRM service type changed successfully.
WinRM service started.

WinRM has been updated for remote management.
WinRM firewall exception enabled.
```

One-To-One Remoting

If you want your remote session to be interactive, then one-to-one remoting is what you want. This type of remoting is provided via the `Enter-PSSession` cmdlet.

In the last chapter, I stored my domain admin credentials in a variable named `$Cred`. If you haven't already done so, go ahead and store your domain admin credentials in the `$Cred` variable.

This allows you to enter the credentials once and use them on a per command basis as long as your current PowerShell session is active.

```
$Cred = Get-Credential
```

Create a one-to-one PowerShell remoting session to the domain controller named dc01.

```
Enter-PSSession -ComputerName dc01 -Credential $Cred
```

```
[dc01]: PS C:\Users\Administrator\Documents>
```

Notice that in the previous example that the PowerShell prompt is preceded by `[dc01]`. This means you're in an interactive PowerShell session to the remote computer named dc01. Any commands you execute run on dc01, not on your local computer. Also, keep in mind that you only have access to the PowerShell commands that exist on the remote computer and not the ones on your local computer. In other words, if you've installed additional modules on your computer, they aren't accessible on the remote computer.

When you're connected to a remote computer via a one-to-one interactive PowerShell remoting session, you're effectively sitting at the remote computer. The objects are normal objects just like the ones you've been working with throughout this entire book.

```
[dc01]: Get-Process | Get-Member


   TypeName: System.Diagnostics.Process

Name                      MemberType     Definition
----                      ----------     ----------
Handles                   AliasProperty  Handles = Handlecount
Name                      AliasProperty  Name = ProcessName
NPM                       AliasProperty  NPM = NonpagedSystemMemorySize64
PM                        AliasProperty  PM = PagedMemorySize64
SI                        AliasProperty  SI = SessionId
VM                        AliasProperty  VM = VirtualMemorySize64
WS                        AliasProperty  WS = WorkingSet64
Disposed                  Event          System.EventHandler Disposed(Sy...
ErrorDataReceived         Event          System.Diagnostics.DataReceived...
Exited                    Event          System.EventHandler Exited(Syst...
OutputDataReceived        Event          System.Diagnostics.DataReceived...
BeginErrorReadLine        Method         void BeginErrorReadLine()
BeginOutputReadLine       Method         void BeginOutputReadLine()
CancelErrorRead           Method         void CancelErrorRead()
CancelOutputRead          Method         void CancelOutputRead()
Close                     Method         void Close()
CloseMainWindow           Method         bool CloseMainWindow()
CreateObjRef              Method         System.Runtime.Remoting.ObjRef ...
Dispose                   Method         void Dispose(), void IDisposabl...
Equals                    Method         bool Equals(System.Object obj)
GetHashCode               Method         int GetHashCode()
GetLifetimeService        Method         System.Object GetLifetimeService()
GetType                   Method         type GetType()
InitializeLifetimeService Method         System.Object InitializeLifetim...
Kill                      Method         void Kill()
Refresh                   Method         void Refresh()
Start                     Method         bool Start()
```

```
ToString                  Method         string ToString()
WaitForExit               Method         bool WaitForExit(int millisecon...
WaitForInputIdle          Method         bool WaitForInputIdle(int milli...
__NounName                NoteProperty   string __NounName=Process
BasePriority              Property       int BasePriority {get;}
Container                 Property       System.ComponentModel.IContaine...
EnableRaisingEvents       Property       bool EnableRaisingEvents {get;s...
ExitCode                  Property       int ExitCode {get;}
ExitTime                  Property       datetime ExitTime {get;}
Handle                    Property       System.IntPtr Handle {get;}
HandleCount               Property       int HandleCount {get;}
HasExited                 Property       bool HasExited {get;}
Id                        Property       int Id {get;}
MachineName               Property       string MachineName {get;}
MainModule                Property       System.Diagnostics.ProcessModul...
MainWindowHandle          Property       System.IntPtr MainWindowHandle ...
MainWindowTitle           Property       string MainWindowTitle {get;}
MaxWorkingSet             Property       System.IntPtr MaxWorkingSet {ge...
MinWorkingSet             Property       System.IntPtr MinWorkingSet {ge...
Modules                   Property       System.Diagnostics.ProcessModul...
NonpagedSystemMemorySize  Property       int NonpagedSystemMemorySize {g...
NonpagedSystemMemorySize64 Property      long NonpagedSystemMemorySize64...
PagedMemorySize           Property       int PagedMemorySize {get;}
PagedMemorySize64         Property       long PagedMemorySize64 {get;}
PagedSystemMemorySize     Property       int PagedSystemMemorySize {get;}
PagedSystemMemorySize64   Property       long PagedSystemMemorySize64 {g...
PeakPagedMemorySize       Property       int PeakPagedMemorySize {get;}
PeakPagedMemorySize64     Property       long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize     Property       int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64   Property       long PeakVirtualMemorySize64 {g...
PeakWorkingSet            Property       int PeakWorkingSet {get;}
PeakWorkingSet64          Property       long PeakWorkingSet64 {get;}
PriorityBoostEnabled      Property       bool PriorityBoostEnabled {get;...
PriorityClass             Property       System.Diagnostics.ProcessPrior...
PrivateMemorySize         Property       int PrivateMemorySize {get;}
PrivateMemorySize64       Property       long PrivateMemorySize64 {get;}
PrivilegedProcessorTime   Property       timespan PrivilegedProcessorTim...
ProcessName               Property       string ProcessName {get;}
ProcessorAffinity         Property       System.IntPtr ProcessorAffinity...
Responding                Property       bool Responding {get;}
```

```
SafeHandle                Property       Microsoft.Win32.SafeHandles.Saf...
SessionId                 Property       int SessionId {get;}
Site                      Property       System.ComponentModel.ISite Sit...
StandardError             Property       System.IO.StreamReader Standard...
StandardInput             Property       System.IO.StreamWriter Standard...
StandardOutput            Property       System.IO.StreamReader Standard...
StartInfo                 Property       System.Diagnostics.ProcessStart...
StartTime                 Property       datetime StartTime {get;}
SynchronizingObject       Property       System.ComponentModel.ISynchron...
Threads                   Property       System.Diagnostics.ProcessThrea...
TotalProcessorTime        Property       timespan TotalProcessorTime {get;}
UserProcessorTime         Property       timespan UserProcessorTime {get;}
VirtualMemorySize         Property       int VirtualMemorySize {get;}
VirtualMemorySize64       Property       long VirtualMemorySize64 {get;}
WorkingSet                Property       int WorkingSet {get;}
WorkingSet64              Property       long WorkingSet64 {get;}
PSConfiguration           PropertySet    PSConfiguration {Name, Id, Prio...
PSResources               PropertySet    PSResources {Name, Id, Handleco...
Company                   ScriptProperty System.Object Company {get=$thi...
CPU                       ScriptProperty System.Object CPU {get=$this.To...
Description               ScriptProperty System.Object Description {get=...
FileVersion               ScriptProperty System.Object FileVersion {get=...
Path                      ScriptProperty System.Object Path {get=$this.M...
Product                   ScriptProperty System.Object Product {get=$thi...
ProductVersion            ScriptProperty System.Object ProductVersion {g...
```

When you're done working with the remote computer, exit the one-to-one remoting session by using the `Exit-PSSession` cmdlet.

```
[dc01]:  Exit-PSSession
```

One-To-Many Remoting

Sometimes you may need to perform a task interactively on a remote computer. But remoting is much more powerful when performing a task on multiple remote computers at the same time. Use the `Invoke-Command` cmdlet to run a command against one or more remote computers at the same time.

```
Invoke-Command -ComputerName dc01, sql02, web01 {
    Get-Service -Name W32time
} -Credential $Cred
```

```
Status   Name        DisplayName       PSComputerName
------   ----        -----------       --------------
Running  W32time     Windows Time      web01
Start... W32time     Windows Time      dc01
Running  W32time     Windows Time      sql02
```

In the previous example, three servers were queried for the status of the Windows Time service. The `Get-Service` cmdlet was placed inside the script block of `Invoke-Command`. `Get-Service` actually runs on the remote computer and the results are returned to your local computer as deserialized objects.

Piping the previous command to `Get-Member` shows that the results are indeed deserialized objects.

```
Invoke-Command -ComputerName dc01, sql02, web01 {
    Get-Service -Name W32time
} -Credential $Cred | Get-Member
```

```
   TypeName: Deserialized.System.ServiceProcess.ServiceController

Name                MemberType   Definition
----                ----------   ----------
GetType             Method       type GetType()
ToString            Method       string ToString(), string ToString(strin...
Name                NoteProperty string Name=W32time
PSComputerName      NoteProperty string PSComputerName=dc01
PSShowComputerName  NoteProperty bool PSShowComputerName=True
RequiredServices    NoteProperty Deserialized.System.ServiceProcess.Servi...
RunspaceId          NoteProperty guid RunspaceId=5ed06925-8037-43ef-9072-...
CanPauseAndContinue Property     System.Boolean {get;set;}
CanShutdown         Property     System.Boolean {get;set;}
CanStop             Property     System.Boolean {get;set;}
Container           Property      {get;set;}
```

```
DependentServices   Property     Deserialized.System.ServiceProcess.Servi...
DisplayName         Property     System.String {get;set;}
MachineName         Property     System.String {get;set;}
ServiceHandle       Property     System.String {get;set;}
ServiceName         Property     System.String {get;set;}
ServicesDependedOn  Property     Deserialized.System.ServiceProcess.Servi...
ServiceType         Property     System.String {get;set;}
Site                Property      {get;set;}
StartType           Property     System.String {get;set;}
Status              Property     System.String {get;set;}
```

Notice that the majority of the methods are missing on deserialized objects. This means they're not live objects; they're inert. You can't start or stop a service using a deserialized object because it's a snapshot of the state of that object the point when the command ran on the remote computer.

That doesn't mean you can't start or stop a service using a method with `Invoke-Command` though. It just means that the method has to be called in the remote session.

I'll stop the Windows Time service on all three of those remote servers using the **Stop()** method to prove this point.

```
Invoke-Command -ComputerName dc01, sql02, web01 {
    (Get-Service -Name W32time).Stop()
} -Credential $Cred

Invoke-Command -ComputerName dc01, sql02, web01 {
    Get-Service -Name W32time
} -Credential $Cred


Status   Name        DisplayName       PSComputerName
------   ----        -----------       --------------
Stopped  W32time     Windows Time      web01
Stopped  W32time     Windows Time      dc01
Stopped  W32time     Windows Time      sql02
```

As mentioned in a previous chapter, if a cmdlet exists for accomplishing a task, I recommend using it instead of using a method. In the previous scenario, I

recommend using the `Stop-Service` cmdlet instead of the stop method. I chose to use the **Stop()** method to prove a point since many people are under the misconception that methods can't be called when using PowerShell remoting. They can't be called on the object that's returned because it's deserialized, but they can be called in the remote session itself.

PowerShell Sessions

In the last example in the previous section, I ran two commands using the `Invoke-Command` cmdlet. That means two separate sessions had to be set up and torn down to run those two commands.

Similar to the CIM sessions discussed in Chapter 7, a PowerShell session to a remote computer can be used to run multiple commands against the remote computer without the overhead of a new session for each individual command.

Create a PowerShell session to each of the three computers we've been working with in this chapter, DC01, SQL02, and WEB01.

```
$Session = New-PSSession -ComputerName dc01, sql02, web01 -Credential $Cred
```

Now use the variable named `$Session` to start the Windows Time service using a method and check the status of the service.

```
Invoke-Command -Session $Session {(Get-Service -Name W32time).Start()}
Invoke-Command -Session $Session {Get-Service -Name W32time}
```

```
Status   Name        DisplayName       PSComputerName
------   ----        -----------       --------------
Running  W32time     Windows Time      web01
Start... W32time     Windows Time      dc01
Running  W32time     Windows Time      sql02
```

Once the session is created using alternate credentials, it's no longer necessary to specify the credentials each time a command is run.

When you're finished using the sessions, be sure to remove them.

```
Get-PSSession | Remove-PSSession
```

Summary

In this chapter you've learned about PowerShell remoting, how to run commands in an interactive session with one remote computer, and how to run commands against multiple computers using one-to-many remoting. You've also learned the benefits of using a PowerShell session when running multiple commands against the same remote computer.

Review

1. How do you enable PowerShell remoting?
2. What is the PowerShell command for starting an interactive session with a remote computer?
3. What is a benefit of using a PowerShell remoting session versus just specifying the computer name with each command?
4. Can a PowerShell remoting session be used with a one-to-one remoting session?
5. What is the difference in the type of objects that are returned by cmdlets versus those returned when running those same cmdlets against remote computers with `Invoke-Command`?

References

- about_Remote[1]
- about_Remote_Output[2]
- about_Remote_Requirements[3]
- about_Remote_Troubleshooting[4]
- about_Remote_Variables[5]
- PowerShell Remoting FAQ[6]

[1] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_remote
[2] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_remote_output
[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_remote_requirements
[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_remote_troubleshooting
[5] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_remote_variables
[6] https://learn.microsoft.com/powershell/scripting/learn/remoting/powershell-remoting-faq

Chapter 9 - Functions

If you're writing PowerShell one-liners or scripts and find yourself often having to modify them for different scenarios, there's a good chance that it's a good candidate to be turned into a function that can be reused.

Whenever possible, I prefer to write functions because they are more tool oriented. I can put the functions in a script module, put that module in the `$env:PSModulePath`, and call the functions without needing to physically locate where they're saved. Using the PowerShellGet module, it's easy to share those modules in a NuGet repository. PowerShellGet ships with PowerShell version 5.0 and higher. It is available as a separate download for PowerShell version 3.0 and higher.

Don't over complicate things. Keep it simple and use the most straight forward way to accomplish a task. Avoid aliases and positional parameters in any code that you reuse. Format your code for readability. Don't hardcode values; use parameters and variables. Don't write unnecessary code even if it doesn't hurt anything. It adds unnecessary complexity. Attention to detail goes a long way when writing any PowerShell code.

Naming

When naming your functions in PowerShell, use a Pascal case[1] name with an approved verb and a singular noun. I also recommend prefixing the noun. For example: `<ApprovedVerb>-<Prefix><SingularNoun>`.

In PowerShell, there's a specific list of approved verbs that can be obtained by running `Get-Verb`.

```
Get-Verb | Sort-Object -Property Verb
```

[1] https://learn.microsoft.com/dotnet/standard/design-guidelines/capitalization-conventions

```
Verb        Group
----        -----
Add         Common
Approve     Lifecycle
Assert      Lifecycle
Backup      Data
Block       Security
Checkpoint  Data
Clear       Common
Close       Common
Compare     Data
Complete    Lifecycle
Compress    Data
Confirm     Lifecycle
Connect     Communications
Convert     Data
ConvertFrom Data
ConvertTo   Data
Copy        Common
Debug       Diagnostic
Deny        Lifecycle
Disable     Lifecycle
Disconnect  Communications
Dismount    Data
Edit        Data
Enable      Lifecycle
Enter       Common
Exit        Common
Expand      Data
Export      Data
Find        Common
Format      Common
Get         Common
Grant       Security
Group       Data
Hide        Common
Import      Data
Initialize  Data
Install     Lifecycle
Invoke      Lifecycle
```

```
Join        Common
Limit       Data
Lock        Common
Measure     Diagnostic
Merge       Data
Mount       Data
Move        Common
New         Common
Open        Common
Optimize    Common
Out         Data
Ping        Diagnostic
Pop         Common
Protect     Security
Publish     Data
Push        Common
Read        Communications
Receive     Communications
Redo        Common
Register    Lifecycle
Remove      Common
Rename      Common
Repair      Diagnostic
Request     Lifecycle
Reset       Common
Resize      Common
Resolve     Diagnostic
Restart     Lifecycle
Restore     Data
Resume      Lifecycle
Revoke      Security
Save        Data
Search      Common
Select      Common
Send        Communications
Set         Common
Show        Common
Skip        Common
Split       Common
Start       Lifecycle
```

```
Step        Common
Stop        Lifecycle
Submit      Lifecycle
Suspend     Lifecycle
Switch      Common
Sync        Data
Test        Diagnostic
Trace       Diagnostic
Unblock     Security
Undo        Common
Uninstall   Lifecycle
Unlock      Common
Unprotect   Security
Unpublish   Data
Unregister  Lifecycle
Update      Data
Use         Other
Wait        Lifecycle
Watch       Common
Write       Communications
```

In the previous example, I've sorted the results by the **Verb** column. The **Group** column gives you an idea of how these verbs are used. It's important to choose an approved verb in PowerShell when functions are added to a module. The module generates a warning message at load time if you choose an unapproved verb. That warning message makes your functions look unprofessional. Unapproved verbs also limit the discoverability of your functions.

A simple function

A function in PowerShell is declared with the function keyword followed by the function name and then an open and closing curly brace. The code that the function will execute is contained within those curly braces.

```
function Get-Version {
    $PSVersionTable.PSVersion
}
```

The function shown is a simple example that returns the version of PowerShell.

```
Get-Version
```

```
Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693
```

There's a good chance of name conflict with functions named something like `Get-Version` and default commands in PowerShell or commands that others may write. This is why I recommend prefixing the noun portion of your functions to help prevent naming conflicts. In the following example, I'll use the prefix "PS".

```
function Get-PSVersion {
    $PSVersionTable.PSVersion
}
```

Other than the name, this function is identical to the previous one.

```
Get-PSVersion
```

```
Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693
```

Even when prefixing the noun with something like PS, there's still a good chance of having a name conflict. I typically prefix my function nouns with my initials. Develop a standard and stick to it.

```
function Get-MrPSVersion {
    $PSVersionTable.PSVersion
}
```

This function is no different than the previous two other than using a more sensible name to try to prevent naming conflicts with other PowerShell commands.

```
Get-MrPSVersion
```

```
Major  Minor  Build  Revision
-----  -----  -----  --------
5      1      14393  693
```

Once loaded into memory, you can see functions on the **Function** PSDrive.

```
Get-ChildItem -Path Function:\Get-*Version
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Function        Get-Version
Function        Get-PSVersion
Function        Get-MrPSVersion
```

If you want to remove these functions from your current session, you'll have to remove them from the **Function** PSDrive or close and reopen PowerShell.

```
Get-ChildItem -Path Function:\Get-*Version | Remove-Item
```

Verify that the functions were indeed removed.

```
Get-ChildItem -Path Function:\Get-*Version
```

If the functions were loaded as part of a module, the module can be unloaded to remove them.

```
Remove-Module -Name <ModuleName>
```

The `Remove-Module` cmdlet removes modules from memory in your current PowerShell session, it doesn't remove them from your system or from disk.

Parameters

Don't statically assign values! Use parameters and variables. When it comes to naming your parameters, use the same name as the default cmdlets for your parameter names whenever possible.

```
function Test-MrParameter {

    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

Why did I use **ComputerName** and not **Computer**, **ServerName**, or **Host** for my parameter name? It's because I wanted my function standardized like the default cmdlets.

I'll create a function to query all of the commands on a system and return the number of them that have specific parameter names.

```
function Get-MrParameterCount {
    param (
        [string[]]$ParameterName
    )

    foreach ($Parameter in $ParameterName) {
        $Results = Get-Command -ParameterName $Parameter -ErrorAction SilentlyC\
ontinue

        [pscustomobject]@{
            ParameterName   = $Parameter
            NumberOfCmdlets = $Results.Count
        }
    }
}
```

As you can see in the results shown below, 39 commands that have a **ComputerName** parameter. There aren't any cmdlets that have parameters such as **Computer**, **ServerName**, **Host**, or **Machine**.

```
Get-MrParameterCount -ParameterName ComputerName, Computer, ServerName,
    Host, Machine


ParameterName NumberOfCmdlets
------------- ---------------
ComputerName               39
Computer                    0
ServerName                  0
Host                        0
Machine                     0
```

I also recommend using the same case for your parameter names as the default cmdlets. Use `ComputerName`, not `computername`. This makes your functions look and feel like the default cmdlets. People who are already familiar with PowerShell will feel right at home.

The `param` statement allows you to define one or more parameters. The parameter definitions are separated by a comma (`,`). For more information, see about_Functions_Advanced_Parameters[2].

Advanced Functions

Turning a function in PowerShell into an advanced function is really simple. One of the differences between a function and an advanced function is that advanced functions have a number of common parameters that are added to the function automatically. These common parameters include parameters such as **Verbose** and **Debug**.

I'll start out with the `Test-MrParameter` function that was used in the previous section.

```
function Test-MrParameter {

    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

What I want you to notice is that the `Test-MrParameter` function doesn't have any common parameters. There are a couple of different ways to see the common parameters. One is by viewing the syntax using `Get-Command`.

```
Get-Command -Name Test-MrParameter -Syntax


Test-MrParameter [[-ComputerName] <Object>]
```

Another is to drill down into the parameters with `Get-Command`.

[2] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters

```
(Get-Command -Name Test-MrParameter).Parameters.Keys
```

```
ComputerName
```

Add `CmdletBinding` to turn the function into an advanced function.

```
function Test-MrCmdletBinding {

    [CmdletBinding()] # Turns a regular function into an advanced function
    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

Adding `CmdletBinding` adds the common parameters automatically. `CmdletBinding` requires a `param` block, but the `param` block can be empty.

```
Get-Command -Name Test-MrCmdletBinding -Syntax
```

```
Test-MrCmdletBinding [[-ComputerName] <Object>] [<CommonParameters>]
```

Drilling down into the parameters with `Get-Command` shows the actual parameter names including the common ones.

```
(Get-Command -Name Test-MrCmdletBinding).Parameters.Keys
```

```
ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
```

SupportsShouldProcess

`SupportsShouldProcess` adds **WhatIf** and **Confirm** parameters. These are only needed for commands that make changes.

```
function Test-MrSupportsShouldProcess {

    [CmdletBinding(SupportsShouldProcess)]
    param (
        $ComputerName
    )

    Write-Output $ComputerName

}
```

Notice that there are now **WhatIf** and **Confirm** parameters.

```
Get-Command -Name Test-MrSupportsShouldProcess -Syntax
```

```
Test-MrSupportsShouldProcess [[-ComputerName] <Object>] [-WhatIf] [-Confirm]
[<CommonParameters>]
```

Once again, you can also use `Get-Command` to return a list of the actual parameter names including the common ones along with WhatIf and Confirm.

```
(Get-Command -Name Test-MrSupportsShouldProcess).Parameters.Keys
```

```
ComputerName
Verbose
Debug
ErrorAction
WarningAction
InformationAction
ErrorVariable
WarningVariable
InformationVariable
OutVariable
OutBuffer
PipelineVariable
WhatIf
Confirm
```

Parameter Validation

Validate input early on. Why allow your code to continue on a path when it's not possible to run without valid input?

Always type the variables that are being used for your parameters (specify a datatype).

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [string]$ComputerName
    )

    Write-Output $ComputerName

}
```

In the previous example, I've specified **String** as the datatype for the **ComputerName** parameter. This causes it to allow only a single computer name to be specified. If more than one computer name is specified via a comma-separated list, an error is generated.

```
Test-MrParameterValidation -ComputerName Server01, Server02
```

```
Test-MrParameterValidation : Cannot process argument transformation on
parameter 'ComputerName'. Cannot convert value to type System.String.
At line:1 char:42
+ Test-MrParameterValidation -ComputerName Server01, Server02
+                                          ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidData: (:) [Test-MrParameterValidation]
   , ParameterBindingArgumentTransformationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,Test-MrP
   arameterValidation
```

The problem with the current definition is that it's valid to omit the value of the **ComputerName** parameter, but a value is required for the function to complete successfully. This is where the `Mandatory` parameter attribute comes in handy.

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string]$ComputerName
    )

    Write-Output $ComputerName

}
```

The syntax used in the previous example is PowerShell version 3.0 and higher compatible. `[Parameter(Mandatory=$true)]` could be specified instead to make the function compatible with PowerShell version 2.0 and higher. Now that the **ComputerName** is required, if one isn't specified, the function will prompt for one.

```
Test-MrParameterValidation


cmdlet Test-MrParameterValidation at command pipeline position 1
Supply values for the following parameters:
ComputerName:
```

If you want to allow for more than one value for the **ComputerName** parameter, use the **String** datatype but add open and closed square brackets to the datatype to allow for an array of strings.

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory)]
        [string[]]$ComputerName
    )

    Write-Output $ComputerName

}
```

Maybe you want to specify a default value for the **ComputerName** parameter if one isn't specified. The problem is that default values can't be used with mandatory parameters. Instead, you'll need to use the `ValidateNotNullOrEmpty` parameter validation attribute with a default value.

```
function Test-MrParameterValidation {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    Write-Output $ComputerName

}
```

Even when setting a default value, try not to use static values. In the previous example, `$env:COMPUTERNAME` is used as the default value, which is automatically translated into the local computer name if a value is not provided.

Verbose Output

While inline comments are useful, especially if you're writing some complex code, they never get seen by users unless they look into the code itself.

The function shown in the following example has an inline comment in the `foreach` loop. While this particular comment may not be that difficult to locate, imagine if the function included hundreds of lines of code.

```
function Test-MrVerboseOutput {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    foreach ($Computer in $ComputerName) {
        #Attempting to perform an action on $Computer <<-- Don't use
        #inline comments like this, use write verbose instead.
        Write-Output $Computer
    }

}
```

A better option is to use `Write-Verbose` instead of inline comments.

```
function Test-MrVerboseOutput {

    [CmdletBinding()]
    param (
        [ValidateNotNullOrEmpty()]
        [string[]]$ComputerName = $env:COMPUTERNAME
    )

    foreach ($Computer in $ComputerName) {
        Write-Verbose -Message "Attempting to perform an action on $Computer"
        Write-Output $Computer
    }

}
```

When the function is called without the **Verbose** parameter, the verbose output won't be displayed.

```
Test-MrVerboseOutput -ComputerName Server01, Server02
```

When it's called with the **Verbose** parameter, the verbose output will be displayed.

```
Test-MrVerboseOutput -ComputerName Server01, Server02 -Verbose
```

Pipeline Input

When you want your function to accept pipeline input, some additional coding is necessary. As mentioned earlier in this book, commands can accept pipeline input **by value** (by type) or **by property name**. You can write your functions just like the native commands so that they accept either one or both of these types of input.

To accept pipeline input **by value**, specified the `ValueFromPipeline` parameter attribute for that particular parameter. Keep in mind that you can only accept pipeline input **by value** from one of each datatype. For example, if you have two parameters that accept string input, only one of those can accept pipeline input **by value** because if you specified it for both of the string parameters, the pipeline input wouldn't know which one to bind to. This is another reason I call this type of pipeline input *by type* instead of **by value**.

Pipeline input comes in one item at a time similar to the way items are handled in a `foreach` loop. At a minimum, a `process` block is required to process each of these items if you're accepting an array as input. If you're only accepting a single value as input, a `process` block isn't necessary, but I still recommend specifying it for consistency.

```
function Test-MrPipelineInput {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline)]
        [string[]]$ComputerName
    )

    PROCESS {
        Write-Output $ComputerName
    }

}
```

Accepting pipeline input **by property name** is similar except it's specified with the `ValueFromPipelineByPropertyName` parameter attribute and it can be specified for any number of parameters regardless of datatype. The key is that the output of the command that's being piped in has to have a property name that matches the name of the parameter or a parameter alias of your function.

```
function Test-MrPipelineInput {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
            Write-Output $ComputerName
    }

}
```

`BEGIN` and `END` blocks are optional. `BEGIN` would be specified before the `PROCESS` block and is used to perform any initial work prior to the items being received from the pipeline. This is important to understand. Values that are piped in are not accessible in the `BEGIN` block. The `END` block would be specified after the `PROCESS` block and is used for cleanup once all of the items that are piped in have been processed.

Error Handling

The function shown in the following example generates an unhandled exception when a computer can't be contacted.

```
function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            Test-WSMan -ComputerName $Computer
        }
    }

}
```

There are a couple of different ways to handle errors in PowerShell. `Try/Catch` is the more modern way to handle errors.

```
function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer: $Compute\
r"
```

```
            }
        }
    }

}
```

Although the function shown in the previous example uses error handling, it also generates an unhandled exception because the command doesn't generate a terminating error. This is also important to understand. Only terminating errors are caught. Specify the **ErrorAction** parameter with **Stop** as the value to turn a non-terminating error into a terminating one.

```
function Test-MrErrorHandling {

    [CmdletBinding()]
    param (
        [Parameter(Mandatory,
                   ValueFromPipeline,
                   ValueFromPipelineByPropertyName)]
        [string[]]$ComputerName
    )

    PROCESS {
        foreach ($Computer in $ComputerName) {
            try {
                Test-WSMan -ComputerName $Computer -ErrorAction Stop
            }
            catch {
                Write-Warning -Message "Unable to connect to Computer: $Compute\
r"
            }
        }
    }

}
```

Don't modify the global `$ErrorActionPreference` variable unless absolutely necessary. If you're using something like .NET directly from within your PowerShell function,

you can't specify the **ErrorAction** on the command itself. In that scenario, you might need to change the global `$ErrorActionPreference` variable, but if you do change it, change it back immediately after trying the command.

Comment-Based Help

It's considered to be a best practice to add comment based help to your functions so the people you're sharing them with will know how to use them.

```
function Get-MrAutoStoppedService {

<#
.SYNOPSIS
    Returns a list of services that are set to start automatically, are not
    currently running, excluding the services that are set to delayed start.

.DESCRIPTION
    Get-MrAutoStoppedService is a function that returns a list of services
    from the specified remote computer(s) that are set to start
    automatically, are not currently running, and it excludes the services
    that are set to start automatically with a delayed startup.

.PARAMETER ComputerName
    The remote computer(s) to check the status of the services on.

.PARAMETER Credential
    Specifies a user account that has permission to perform this action. The
    default is the current user.

.EXAMPLE
     Get-MrAutoStoppedService -ComputerName 'Server1', 'Server2'

.EXAMPLE
     'Server1', 'Server2' | Get-MrAutoStoppedService

.EXAMPLE
     Get-MrAutoStoppedService -ComputerName 'Server1' -Credential (Get-Credenti\
al)
```

```
.INPUTS
    String

.OUTPUTS
    PSCustomObject

.NOTES
    Author:  Mike F. Robbins
    Website: https://mikefrobbins.com
    Twitter: @mikefrobbins
#>

    [CmdletBinding()]
    param (

    )

    #Function Body

}
```

When you add comment based help to your functions, help can be retrieved for them just like the default built-in commands.

All of the syntax for writing a function in PowerShell can seem overwhelming especially for someone who is just getting started. Often times if I can't remember the syntax for something, I'll open a second copy of the ISE on a separate monitor and view the "Cmdlet (advanced function) - Complete" snippet while typing in the code for my function. Snippets can be accessed in the PowerShell ISE using the *Ctrl + J* key combination.

Summary

In this chapter you've learned the basics of writing functions in PowerShell to include how to turn a function into an advanced function and some of the more important elements that you should consider when writing PowerShell functions

such as parameter validation, verbose output, pipeline input, error handling, and comment based help.

Review

1. How do you obtain a list of approved verbs in PowerShell?
2. How do you turn a PowerShell function into an advanced function?
3. When should **WhatIf** and **Confirm** parameters be added to your PowerShell functions?
4. How do you turn a non-terminating error into a terminating one?
5. Why should you add comment based help to your functions?

References

- about_Functions[3]
- about_Functions_Advanced_Parameters[4]
- about_CommonParameters[5]
- about_Functions_CmdletBindingAttribute[6]
- about_Functions_Advanced[7]
- about_Try_Catch_Finally[8]
- about_Comment_Based_Help[9]
- Video: PowerShell Toolmaking with Advanced Functions and Script Modules[10]

[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_functions
[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_functions_advanced_parameters
[5] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_commonparameters
[6] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_functions_cmdletbindingattribute
[7] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_functions_advanced
[8] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_try_catch_finally
[9] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_comment_based_help
[10] https://mikefrobbins.com/2016/05/26/video-powershell-toolmaking-with-advanced-functions-and-script-modules/

Chapter 10 - Script modules

Turning your one-liners and scripts in PowerShell into reusable tools becomes even more important if it's something that you're going to use frequently. Packaging your functions in a script module makes them look and feel more professional and makes them easier to share.

Dot-Sourcing Functions

Something that we didn't talk about in the previous chapter is dot-sourcing functions. When a function in a script isn't part of a module, the only way to load it into memory is to dot-source the `.PS1` file that it's saved in.

The following function has been saved as `Get-MrPSVersion.ps1`.

```
function Get-MrPSVersion {
    $PSVersionTable
}
```

When you run the script, nothing happens.

```
.\Get-MrPSVersion.ps1
```

If you try to call the function, it generates an error message.

```
Get-MrPSVersion
```

```
Get-MrPSVersion : The term 'Get-MrPSVersion' 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
+ Get-MrPSVersion
+ ~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Get-MrPSVersion:String) [],
   CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

You can determine if functions are loaded into memory by checking to see if they exist on the **Function** PSDrive.

```
Get-ChildItem -Path Function:\Get-MrPSVersion
```

```
Get-ChildItem : Cannot find path 'Get-MrPSVersion' because it does not
exist.
At line:1 char:1
+ Get-ChildItem -Path Function:\Get-MrPSVersion
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Get-MrPSVersion:String) [Get
   -ChildItem], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.Ge
   tChildItemCommand
```

The problem with calling the script that contains the function is that the functions are loaded in the **Script** scope. When the script completes, that scope is removed and the function is removed with it.

The function needs to be loaded into the **Global** scope. That can be accomplished by dot-sourcing the script that contains the function. The relative path can be used.

```
. .\Get-MrPSVersion.ps1
```

The fully qualified path can also be used.

```
. C:\Demo\Get-MrPSVersion.ps1
```

If a portion of the path is stored in a variable, it can be combined with the remainder of the path. There's no reason to use string concatenation to combine the variable together with the remainder of the path.

```
$Path = 'C:\'
. $Path\Get-MrPSVersion.ps1
```

Now when I check the **Function** PSDrive, the `Get-MrPSVersion` function exists.

```
Get-ChildItem -Path Function:\Get-MrPSVersion
```

```
CommandType     Name                                               Version
-----------     ----                                               -------
Function        Get-MrPSVersion
```

Script Modules

A script module in PowerShell is simply a file containing one or more functions that's saved as a `.PSM1` file instead of a `.PS1` file.

How do you create a script module? You're probably guessing with a command named something like `New-Module`. Your assumption would be wrong. While there is a command in PowerShell named `New-Module`, that command creates a dynamic module, not a script module. Always be sure to read the help for a command even when you think you've found the command you need.

```
help New-Module
```

```
NAME
    New-Module

SYNOPSIS
    Creates a new dynamic module that exists only in memory.


SYNTAX
    New-Module [-Name] <System.String> [-ScriptBlock]
    <System.Management.Automation.ScriptBlock> [-ArgumentList
    <System.Object[]>] [-AsCustomObject] [-Cmdlet <System.String[]>]
    [-Function <System.String[]>] [-ReturnResult] [<CommonParameters>]


DESCRIPTION
    The `New-Module` cmdlet creates a dynamic module from a script block.
    The members of the dynamic module, such as functions and variables, are
    immediately available in the session and remain available until you
    close the session.

    Like static modules, by default, the cmdlets and functions in a dynamic
    module are exported and the variables and aliases are not. However, you
    can use the Export-ModuleMember cmdlet and the parameters of
    `New-Module` to override the defaults.

    You can also use the AsCustomObject parameter of `New-Module` to return
    the dynamic module as a custom object. The members of the modules, such
    as functions, are implemented as script methods of the custom object
    instead of being imported into the session.

    Dynamic modules exist only in memory, not on disk. Like all modules,
    the members of dynamic modules run in a private module scope that is a
    child of the global scope. Get-Module cannot get a dynamic module, but
    Get-Command can get the exported members.

    To make a dynamic module available to `Get-Module`, pipe a `New-Module`
    command to Import-Module, or pipe the module object that `New-Module`
    returns to `Import-Module`. This action adds the dynamic module to the
    `Get-Module` list, but it does not save the module to disk or make it
    persistent.
```

```
RELATED LINKS
    Online Version: https://learn.microsoft.com/powershell/module/microsoft.
    powershell.core/new-module?view=powershell-5.1&WT.mc_id=ps-gethelp
    Export-ModuleMember
    Get-Module
    Import-Module
    Remove-Module
    about_Modules

REMARKS
    To see the examples, type: "get-help New-Module -examples".
    For more information, type: "get-help New-Module -detailed".
    For technical information, type: "get-help New-Module -full".
    For online help, type: "get-help New-Module -online"
```

In the previous chapter, I mentioned that functions should use approved verbs otherwise they'll generate a warning message when the module is imported. The following code uses the `New-Module` cmdlet to create a dynamic module in memory. This module demonstrates the unapproved verb warning.

```
New-Module -Name MyModule -ScriptBlock {

    function Return-MrOsVersion {
        Get-CimInstance -ClassName Win32_OperatingSystem |
        Select-Object -Property @{label='OperatingSystem';expression={$_.Captio\
n}}
    }

    Export-ModuleMember -Function Return-MrOsVersion

} | Import-Module
```

```
WARNING: The names of some imported commands from the module 'MyModule' include
unapproved verbs that might make them less discoverable. To find the commands w\
ith
unapproved verbs, run the Import-Module command again with the Verbose paramete\
r. For a
list of approved verbs, type Get-Verb.
```

Just to reiterate, although the `New-Module` cmdlet was used in the previous example, that's not the command for creating script modules in PowerShell.

Save the following two functions in a file named `MyScriptModule.psm1`.

```
function Get-MrPSVersion {
    $PSVersionTable
}

function Get-MrComputerName {
    $env:COMPUTERNAME
}
```

Try to call one of the functions.

```
Get-MrComputerName
```

```
Get-MrComputerName : The term 'Get-MrComputerName' 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
+ Get-MrComputerName
+ ~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (Get-MrComputerName:String) [
   ], CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

An error message is generated saying the function can't be found. You could also check the **Function** PSDrive just like before and you'll find that it doesn't exist there either.

You could manually import the file with the `Import-Module` cmdlet.

```
Import-Module C:\MyScriptModule.psm1
```

The module autoloading feature was introduced in PowerShell version 3. To take advantage of module autoloading, a script module needs to be saved in a folder with the same base name as the `.PSM1` file and in a location specified in `$env:PSModulePath`.

```
$env:PSModulePath
```

```
C:\Users\mike-ladm\Documents\WindowsPowerShell\Modules;C:\Program Files\Wind
owsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules;C:\
Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\
```

The results are difficult to read. Since the paths are separated by a semicolon, you can split the results to return each path on a separate line. This makes them easier to read.

```
$env:PSModulePath -split ';'
```

```
C:\Users\mike-ladm\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules
C:\Program Files (x86)\Microsoft SQL Server\130\Tools\PowerShell\Modules\
```

The first three paths in the list are the default. When SQL Server Management Studio was installed, it added the last path. For module autoloading to work, the `MyScriptModule.psm1` file needs to be located in a folder named `MyScriptModule` directly inside one of those paths.

Not so fast. For me, my current user path isn't the first one in the list. I almost never use that path since I log into Windows with a different user than the one I use to run PowerShell. That means it's not located in my normal Documents folder.

The second path is the **AllUsers** path. This is the location where I store all of my modules.

The third path is underneath `C:\Windows\System32`. Only Microsoft should be storing modules in that location since it resides within the operating systems folder.

Once the `.PSM1` file is located in the correct path, the module will load automatically when one of its commands is called.

Module Manifests

All modules should have a module manifest. A module manifest contains metadata about your module. The file extension for a module manifest file is `.PSD1`. Not all files with a `.PSD1` extension are module manifests. They can also be used for things such as storing the environmental portion of a DSC configuration. `New-ModuleManifest` is used to create a module manifest. **Path** is the only value that's required. However, the module won't work if **RootModule** isn't specified. It's a good idea to specify **Author** and **Description** in case you decide to upload your module to a NuGet repository with PowerShellGet since those values are required in that scenario.

The version of a module without a manifest is 0.0. This is a dead giveaway that the module doesn't have a manifest.

```
Get-Module -Name MyScriptModule
```

```
ModuleType Version    Name                                ExportedCommands
---------- -------    ----                                ----------------
Script     0.0        MyScriptModule                      {Get-MrComputer...
```

The module manifest can be created with all of the recommended information.

```
$moduleManifestParams = @{
    Path = "$env:ProgramFiles\WindowsPowerShell\Modules\MyScriptModule\MyScript\
Module.psd1"
    RootModule = 'MyScriptModule'
    Author = 'Mike F. Robbins'
    Description = 'MyScriptModule'
    CompanyName = 'mikefrobbins.com'
}

New-ModuleManifest @moduleManifestParams
```

If any of this information is missed during the initial creation of the module manifest, it can be added or updated later using `Update-ModuleManifest`. Don't recreate the manifest using `New-ModuleManifest` once it's already created because the GUID will change.

Defining Public and Private Functions

You may have helper functions that you may want to be private and only accessible by other functions within the module. They are not intended to be accessible to users of your module. There are a couple of different ways to accomplish this.

If you're not following the best practices and only have a `.PSM1` file, then your only option is to use the `Export-ModuleMember` cmdlet.

```
function Get-MrPSVersion {
    $PSVersionTable
}

function Get-MrComputerName {
    $env:COMPUTERNAME
}

Export-ModuleMember -Function Get-MrPSVersion
```

In the previous example, only the `Get-MrPSVersion` function is available to the users of your module, but the `Get-MrComputerName` function is available to other functions within the module itself.

```
Get-Command -Module MyScriptModule


CommandType     Name                                               Version
-----------     ----                                               -------
Function        Get-MrPSVersion                                    1.0
```

If you've added a module manifest to your module (and you should), then I recommend specifying the individual functions you want to export in the **FunctionsToExport** section of the module manifest.

```
FunctionsToExport = 'Get-MrPSVersion'
```

It's not necessary to use both `Export-ModuleMember` in the `.PSM1` file and the **FunctionsToExport** section of the module manifest. One or the other is sufficient.

Summary

In this chapter you've learned how to turn your functions into a script module in PowerShell. You've also learned some of the best practices for creating script modules such as creating a module manifest for your script module.

Review

1. How do you create a script module in PowerShell?
2. Why is it important for your functions to use an approved verb?
3. How do you create a module manifest in PowerShell?
4. What are the two options for exporting only certain functions from your module?
5. What is required for your modules to load automatically when a command is called?

References

- How to Create PowerShell Script Modules and Module Manifests[1]
- about_Modules[2]
- New-ModuleManifest[3]
- Export-ModuleMember[4]

[1] https://mikefrobbins.com/2013/07/04/how-to-create-powershell-script-modules-and-module-manifests/
[2] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/about/about_modules
[3] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/new-modulemanifest
[4] https://learn.microsoft.com/powershell/module/microsoft.powershell.core/export-modulemember