100 POWERSHELL CMDLETS

By: Bijay Kumar

PowerShell is a powerful scripting language developed by Microsoft for task automation and configuration management. It consists of a command-line shell and an associated scripting language.

In this PDF, I have explained 100+ useful PowerShell cmdlets, that you should know as an administrator.

Before checking the cmdlets, check out the below tutorials:

- https://powershellfaqs.com/how-to-run-powershell-script-in-powershell-ise/
- https://powershellfaqs.com/run-powershell-script-in-visual-studio-code/
- https://powershellfaqs.com/powershell-naming-conventions/

I hope you will like these cmdlets.

Introduction to PowerShell Cmdlets

PowerShell cmdlets are essential for performing automated tasks and managing systems. They are powerful tools designed to handle single operations and make complex tasks more manageable.

Cmdlets (pronounced "command-lets") are lightweight commands in the PowerShell environment. They are built on the .NET framework and designed to perform a single function.

For instance, **Get-Process** retrieves the currently running processes on a system, while **Stop-Process** can stop a specific process by its ID or name.

Each cmdlet follows a **Verb-Noun** naming convention, making it easy to understand their purpose.

Examples include **Get-Content** for reading file contents and Set-Content for writing to a file.

Most cmdlets produce objects as output, which can be further processed using other cmdlets.

How Cmdlets Differ from Commands in PowerShell

Cmdlets are different from traditional shell commands in several ways. Unlike standard commands, cmdlets are not standalone executables. Instead, they are instances of .NET classes, offering more integrated functionality.

Here is summary:

Feature	PowerShell Cmdlets	CMD Commands
Definition	Cmdlets are lightweight commands that are part of the PowerShell environment.	CMD commands are traditional command-line instructions used in the Windows Command Prompt.
Implementation	Implemented as .NET classes, typically written in C# or other .NET languages.	Implemented as executable programs or scripts.
Syntax	Verb-Noun format (e.g., `Get-Process`, `Set-Item`).	Varies widely, often simple and less structured (e.g., `dir`, `copy`).
Output	Outputs objects, which can be piped to other cmdlets for further processing.	Outputs text, which often requires parsing for further processing.

Feature	PowerShell Cmdlets	CMD Commands
Integration	Deeply integrated with .NET, allowing for advanced scripting and automation capabilities.	Limited to basic command-line operations and scripting.
Error Handling	Provides robust error handling and debugging capabilities.	Basic error handling, often requiring manual intervention.
Extensibility	Extensible via custom cmdlets and modules.	Limited extensibility, primarily through batch files and external programs.
Environment	Designed for system administration, automation, and configuration management.	Designed for general command-line tasks and basic file operations.

PowerShell cmdlet examples

1) Get-Process: Lists running processes

The Get-Process cmdlet retrieves the processes currently running on a local computer. Without any parameters, this cmdlet lists all active processes.

Users can filter the results by specifying a process name or process ID (PID). This is useful for quickly finding specific processes.

For example, to view processes with "Calculator" in the name, use:

Get-Process -Name 'Calculator'

This command returns details like process ID, name, and memory usage.

To get the top five processes using more than 25MB RAM:

Get-Process | Where-Object { $_.WorkingSet -gt 25000*1024 } | Sort-Object -Property WorkingSet -Descending | Select-Object -First 5

Another useful example is stopping a process by its ID or name:

Stop-Process -Id 1234

2) Get-Service: Retrieves the Status of Services

The Get-Service cmdlet in PowerShell allows users to view the status of services on both local and remote computers.

By default, running Get-Service without parameters retrieves all the services on the local machine. This includes both running and stopped services.

Users can specify particular services by using service names or search strings.

For example, **Get-Service "wmi*"** retrieves all services that start with "wmi".

Get-Service can be combined with other cmdlets. By piping its output to Where-Object, users can filter services based on properties like status.

For example,

Get-Service | Where-Object {$_.Status -eq 'Running'}

The above cmdlet lists only running services.

3) Get-Help: Provides help for cmdlets

The Get-Help cmdlet in PowerShell provides detailed information about cmdlets, functions, workflows, and scripts.

To get general help about a cmdlet, type Get-Help followed by the cmdlet's name. For example, to get information about the Get-Process cmdlet, use the below cmdlet:

Get-Help Get-Process

PowerShell also supports detailed help. To see this, use the -Detailed parameter. For example, type **Get-Help Get-Process -Detailed**.

If you need examples, use the -Examples parameter. It will show you practical examples of how to use the cmdlet. For instance, Get-Help Get-Process -Examples will show examples for the Get-Process cmdlet.

More extensive information is available with the -Full parameter. It provides all available details, including syntax and parameter descriptions. Use Get-Help Get-Process -Full for this.

Conceptual help articles in PowerShell begin with "about_". To get help about comparison operators, type **Get-Help about_Comparison_Operators**.

For the most up-to-date help content, use the **Update-Help cmdlet**. This ensures that you have the latest help files installed.

4) Set-ExecutionPolicy: Changes the User Preference for PowerShell Script Execution

The Set-ExecutionPolicy cmdlet is essential for managing PowerShell script execution policies. It allows users to control the conditions under which scripts run, enhancing security.

On Windows, Set-ExecutionPolicy supports several modes. Restricted mode is the default, preventing any scripts from running. AllSigned requires all scripts to be signed by a trusted publisher.

To enable running scripts that are unsigned but originated from the local computer, set the policy to RemoteSigned. Use the command:

Set-ExecutionPolicy RemoteSigned

In Unrestricted mode, all scripts can run, though you'll receive warnings when running scripts downloaded from the internet. Set this with:

Set-ExecutionPolicy Unrestricted

For maximum script execution, Bypass allows all scripts to run without any warnings or prompts. Use this with:

Set-ExecutionPolicy Bypass

For non-Windows systems starting with PowerShell 6.0, the default execution policy is Unrestricted and cannot be changed. While the cmdlet is available, it displays a message indicating that policy changes are not supported.

To avoid prompts when changing the execution policy, use the -Force parameter:

Set-ExecutionPolicy RemoteSigned -Force

To see the current execution policy, use the Get-ExecutionPolicy cmdlet:

Get-ExecutionPolicy

Adjusting the execution policy affects only the current user by default. To apply changes to all users on the computer, use the -Scope parameter:

Set-ExecutionPolicy RemoteSigned -Scope LocalMachine

5) Get-EventLog: Displays event logs from local or remote computers

Get-EventLog is a PowerShell cmdlet that retrieves event logs. It works with Windows classic event logs like Application, System, or Security.

For basic usage, **Get-EventLog** can list all available event logs on a local computer.

For example, **Get-EventLog -List** shows a list of logs available.

To get details from a specific log, use the -LogName parameter.

For instance, Get-EventLog -LogName System fetches entries from the System log.

To see only recent events, limit the results.

The command **Get-EventLog -LogName System -Newest 10** shows the most recent 10 entries.

Reading event logs from a remote computer is also possible. Use the -ComputerName parameter.

For example, **Get-EventLog -LogName Application -ComputerName Server01** retrieves logs from a remote machine named Server01.

6) Restart-Computer: Restarts the local computer or a remote computer

The Restart-Computer cmdlet in PowerShell is used to restart the operating system on both local and remote computers. This cmdlet is available only on Windows platforms.

To restart a local computer, use:

Restart-Computer

To restart a remote computer, specify the remote computer's name:

Restart-Computer -ComputerName "RemotePC"

Including credentials for authentication:

Restart-Computer -ComputerName "RemotePC" -Credential (Get-Credential)

Use the -Force parameter to force an immediate restart:

Restart-Computer -ComputerName "RemotePC" -Force

You can conduct these operations as background jobs. For example:

Restart-Computer -ComputerName "RemotePC" -AsJob

When working with multiple remote computers, list the names separated by commas:

Restart-Computer -ComputerName "RemotePC1","RemotePC2"

For specifying authentication levels:

Restart-Computer -ComputerName "RemotePC" -Authentication Default

7) New-Item: Creates a new item

The New-Item cmdlet in PowerShell is used to create a new item in a specified path. This is useful for making new files, directories, or registry keys.

The syntax for creating a directory is straightforward. For instance, to create a new folder named "TestFolder" in the C:\ drive, use the following command:

New-Item -Path "C:\TestFolder" -ItemType "Directory"

To create a file named "Example.txt" inside the "TestFolder" directory, you'll use:

New-Item -Path "C:\TestFolder\Example.txt" -ItemType "File"

The New-Item cmdlet can be used to set the value of the items it creates.

For example, creating a text file with initial content is done by specifying the -Value parameter.

New-Item -Path "C:\TestFolder\Example.txt" -ItemType "File" -Value "Initial content"

In addition to creating files and directories, New-Item also works with the Windows registry. For example, to create a new registry key in HKEY_CURRENT_USER\Software, you can use:

New-Item -Path "HKCU:\Software\NewKey"

References:
https://powershellfaqs.com/create-file-if-not-exists-in-powershell/
https://powershellfaqs.com/create-folder-if-not-exist-in-powershell/

8) Get-Content: Gets the content of a file

The Get-Content cmdlet in PowerShell is used to read the content of a file. It retrieves data one line at a time and outputs it as an array of strings, where each element represents a line.

To use it, provide the path to the file. For example, **Get-Content -Path "C:\example.txt"** will read the file at that location.

One useful feature of Get-Content is that it can read a specified number of lines from the beginning or end of a file. To read the first five lines, use **Get-Content -Path "C:\example.txt" -TotalCount 5**.

You can also filter content. For instance, if you want to display only lines containing a specific string, pipe the output to the Select-String cmdlet.

Get-Content -Path "C:\example.txt" | Select-String "search term" will show only matching lines.

Variables can store the output of Get-Content. This is useful for further manipulation or processing. For instance, **$content = Get-Content -Path "C:\example.txt"** sets the variable $content to the file's contents.

For large files, reading the content in chunks using the -ReadCount parameter can be efficient.

Get-Content -Path "C:\example.txt" -ReadCount 10 reads 10 lines at a time.

Another practical use is reading the last few lines of a log file.

Get-Content -Path "C:\example.log" -Tail 5 reads the last five lines.

References:

https://powershellfaqs.com/read-a-file-line-by-line-in-powershell/

9) Set-Content: Writes or replaces the content of a file

Set-Content is a powerful cmdlet in PowerShell used to write or replace the content of a file. This cmdlet differs from Add-Content, which appends content to a file instead of replacing it. Set-Content is beneficial when you need to overwrite existing data.

For example, to write a simple string to a text file, the command would be:

Set-Content -Path "C:\example.txt" -Value "Hello, World!"

This command takes a string, "Hello, World!", and writes it to the file located at C:\example.txt.

If the file already exists, the content is replaced with the new string.

Using Set-Content can also be handy when working with large outputs. For example, if you generate a list of files in a directory and want to save this list:

Get-ChildItem -Path "C:\MyFolder" | Set-Content -Path "C:\FileList.txt"

This command gets a list of files in C:\MyFolder and writes the list to C:\FileList.txt, replacing any existing content.

References:

https://powershellfaqs.com/write-string-to-file-in-powershell/

https://powershellfaqs.com/write-variables-to-a-file-in-powershell/

10) Get-ChildItem: Gets files and folders

The Get-ChildItem cmdlet in PowerShell is used to access files and directories in a specified location.

It can be used to list files, folders, registry keys, or certificates.

A basic use of Get-ChildItem is to list items in a directory.

To do this, the command is followed by the directory path.

Get-ChildItem -Path "C:\ExampleDirectory"

For a more detailed view, the Recurse parameter allows getting all items in child directories as well.

Get-ChildItem -Path "C:\ExampleDirectory" -Recurse

Filtering the results is also possible.

For example, to list only .txt files, the -Filter parameter is used.

Get-ChildItem -Path "C:\ExampleDirectory" -Filter "*.txt"

To limit the depth of recursion, the -Depth parameter can be added.

Get-ChildItem -Path "C:\ExampleDirectory" -Recurse -Depth 2

The cmdlet can also be used to get items from other locations like the registry or certificate store.

For example, listing all registry keys under a specific path:

Get-ChildItem -Path "HKLM:\Software\Example"

When working with certificates, the command might look like this:

Get-ChildItem -Path "Cert:\LocalMachine\My"

In addition, Get-ChildItem doesn't display empty directories by default. To include them, the -Force parameter can be used.

Get-ChildItem -Path "C:\ExampleDirectory" -Force

Refereces:
https://powershellfaqs.com/count-files-in-a-folder-using-powershell/

11) Remove-Item: Deletes files and folders

The Remove-Item cmdlet is used in PowerShell to delete items such as files and folders.

It supports different item types including files, directories, and even registry keys.

To delete a single file, use the command:

Remove-Item -Path "C:\path\to\your\file.txt"

For directories, the command is similar:

Remove-Item -Path "C:\path\to\your\folder" -Recurse

Using the -Recurse parameter ensures all contents within a directory are deleted.

If the -Recurse parameter is not included, only the directory itself is deleted, leaving its contents behind.

To exclude specific files or folders from deletion, the -Exclude parameter can be used:

Remove-Item -Path "C:\path\to\your\folder*" -Exclude "file1.txt","subfolder"

This will delete everything in the folder except the specified items.

To delete only files within a directory but keep subfolders, use a wildcard:

Remove-Item -Path "C:\path\to\your\folder*.*"

This command deletes all files that have a dot in their name, thus keeping subfolders intact.

Deleting items can be risky, so adding the -WhatIf parameter allows you to preview what will be deleted:

Remove-Item -Path "C:\path\to\your\folder*" -WhatIf

This command shows what would happen if the command were executed, without actually performing the deletion.

References:

https://powershellfaqs.com/delete-contents-of-folder-in-powershell/

12) Copy-Item: Copies an item from one location to another

The Copy-Item cmdlet in PowerShell allows users to copy items from one location to another.

This cmdlet can copy files, directories, and other items within the same namespace.

It is essential to understand that Copy-Item does not perform a cut operation. The original item remains in place after copying.

For example, to copy a file named "example.txt" from the "C:\Source" folder to the "C:\Destination" folder, the command is:

Copy-Item -Path "C:\Source\example.txt" -Destination "C:\Destination"

Copy-Item can also be used to copy directories.

To copy an entire folder and its contents, use the -Recurse parameter:

Copy-Item -Path "C:\SourceFolder" -Destination "C:\DestinationFolder" -Recurse

In addition to copying files locally, Copy-Item can copy to remote locations. Here is an example of copying a file to a remote computer:

Copy-Item -Path "C:\Source\example.txt" -Destination "\\RemoteComputer\SharedFolder"

If you want to copy and rename an item simultaneously, use the -Destination parameter with the new name:

Copy-Item -Path "C:\Source\example.txt" -Destination "C:\Destination\newname.txt"

For verbose output, which provides detailed information about the copy operation, use the -Verbose parameter:

Copy-Item -Path "C:\Source\example.txt" -Destination "C:\Destination" -Verbose

To force the copy process, such as overwriting existing files, include the -Force parameter:

Copy-Item -Path "C:\Source\example.txt" -Destination "C:\Destination" -Force

References:
https://powershellfaqs.com/copy-files-from-one-folder-to-another-in-powershell/
https://powershellfaqs.com/copy-and-rename-files-in-powershell/

13) Move-Item: Moves an item from one location to another

Move-Item is a cmdlet in PowerShell used to move items like files, directories, and registry keys from one location to another.

It relocates the item and also its properties and contents.

The syntax for Move-Item is like below:

Move-Item -Path 'C:\source\file.txt' -Destination 'C:\destination\file.txt'

In this command, Move-Item moves file.txt from the source folder to the destination folder.

If a file with the specified name already exists in the destination, you need to use the -Force parameter to overwrite it. For example:

Move-Item -Path 'C:\source\data.txt' -Destination 'C:\dest\data.txt' -Force

The -Force parameter ensures that the existing file in the destination gets replaced by the new file.

You can also move entire directories. For instance:

Move-Item -Path 'C:\sourceDir' -Destination 'C:\destDir'

This command moves the directory named sourceDir to destDir.

The -LiteralPath parameter can be used if your item path includes special characters. For example:

Move-Item -LiteralPath 'C:\Folder With[Special]Chars' -Destination 'C:\NewLocation'

This command safely moves items even when the paths contain special characters.

References:

https://powershellfaqs.com/move-a-file-to-a-folder-in-powershell/

14) Rename-Item: Renames an item

The Rename-Item cmdlet in PowerShell changes the name of a specified item, such as a file or folder.

It allows users to rename items without modifying their content.

A simple example of renaming a file is:

Rename-Item -Path "C:\Example\oldname.txt" -NewName "newname.txt"

This command changes oldname.txt to newname.txt.

Rename-Item does not move items. To move and rename, use the Move-Item cmdlet instead.

Here's a basic example of moving and renaming:

Move-Item -Path "C:\Example\oldname.txt" -Destination "C:\NewFolder\newname.txt"

To rename multiple items, combine Rename-Item with Get-ChildItem. For example:

Get-ChildItem -Path "C:\Logs\" | Rename-Item -NewName {$_.name -replace "old", "new"}

This replaces "old" with "new" in the filenames within the Logs folder.

It is useful to know that Rename-Item does not accept wildcards directly.

To handle more complex renaming patterns, use the -replace operator:

Rename-Item -Path "C:\Example\name (Custom).txt" -NewName {$_.name -replace " \(Custom\)", ""}

This command removes "(Custom)" from the file name.

In cases where special characters need to be stripped from filenames, a similar approach can be followed:

Get-ChildItem -Path "C:\Files\" | Rename-Item -NewName {$_.name -replace "[!@#\$%\^&*\(\)_\+]", ""}

This removes special characters from filenames in the specified directory.

References:

https://powershellfaqs.com/copy-and-rename-files-in-powershell/

https://powershellfaqs.com/rename-files-with-date-in-powershell/

15) Import-Module: Imports a PowerShell module

The Import-Module cmdlet is used for managing modules in PowerShell. It allows users to load specific modules into their session.

This cmdlet is useful when you need to access the commands, functions, and providers within a module that is not automatically loaded.

The basic syntax for importing a module is straightforward:

Import-Module -Name ModuleName

16) Export-ModuleMember: Exports functions, aliases, variables, or cmdlets from a module

The Export-ModuleMember cmdlet specifies which parts of a PowerShell module are made available for use. This can include functions, cmdlets, variables, and aliases.

It must be used within a script module (.psm1 file) or a dynamic module created with the New-Module cmdlet.

By default, all functions in a module are visible. However, variables and aliases are not. Using Export-ModuleMember, you can control what gets exported.

For example, [**Export-ModuleMember -Function "Get-User", "Set-User"**] would export only the specified functions.

If no Export-ModuleMember command is included in a script module, only the functions are exported by default. Variables and aliases require explicit exporting using this cmdlet.

This helps in managing the scope and ensuring only necessary components are available to the user.

17) Add-Content: Appends content to a file

The Add-Content cmdlet in PowerShell is used to append content to a specified file. This is useful for adding new information to an existing text file without overwriting the current data.

For example, to add a simple string to a file, the command might look like this:

Add-Content -Path "example.txt" -Value "New content"

This command appends "New content" to the end of "example.txt."

Add-Content can also append the current date and time to a file. This can be done using:

Add-Content -Path "log.txt" -Value (Get-Date)

This command adds the current date and time to "log.txt" each time it's run.

Multiple files can be updated simultaneously. For instance:

Add-Content -Path "file1.log", "file2.log" -Value "Update"

This example appends "Update" to both "file1.log" and "file2.log."

To avoid adding a new line after the content, the -NoNewline switch can be used:

Add-Content -Path "example.txt" -Value "No newline" -NoNewline

This appends "No newline" to "example.txt" without creating a new line.

Add-Content is a powerful cmdlet for managing text files efficiently.

18) Get-Command: Finds cmdlets, functions, workflows, aliases in PowerShell

Get-Command in PowerShell helps users find various commands available on their systems. It can list cmdlets, functions, workflows, and aliases. This makes it easier for users to discover and use the tools they need.

When run without parameters, Get-Command displays all commands available in PowerShell. This includes commands from modules and those imported from other sessions.

Using the -Name parameter allows users to search for specific commands by their names. For example, **Get-Command -Name Get-Process** will find the Get-Process cmdlet. This helps narrow down the search to exact matches.

The -CommandType parameter refines the search by specifying the type of command. Options include Alias, Cmdlet, Function, or Application.

For instance, **Get-Command -CommandType** Function will list all functions available. This is useful when looking for specific types of commands.

19) Get-Alias: Retrieves aliases

The Get-Alias cmdlet in PowerShell is used to retrieve aliases in the current session.

By default, Get-Alias takes an alias and returns the command name it represents. This cmdlet also shows built-in aliases, aliases you have set or imported, and aliases added to your profile.

For instance, running **Get-Alias** without any parameters will list all aliases available in your session. This includes useful shortcuts like ls for **Get-ChildItem**.

To find out which command ls is an alias for, you can type:

Get-Alias -Name ls

This will return that ls is an alias for Get-ChildItem.

To list all aliases for a specific command such as Get-ChildItem, use:

Get-Alias -Definition Get-ChildItem

These commands can help make repeated tasks more efficient.

20) Set-Alias: Creates a new alias

The Set-Alias cmdlet in PowerShell allows users to create or change an alias for a cmdlet or a command, such as a function, script, file, or other executable. This is useful for creating shortcuts to commonly used commands.

An alias is an alternate name that can be used in place of a cmdlet or command. For example, "sal" is the alias for the Set-Alias cmdlet itself. This can help in reducing typing effort and improving efficiency.

To create an alias, use the Set-Alias cmdlet followed by the name of the alias and the command it represents. For instance, **Set-Alias -Name np -Value notepad.exe** creates an alias "np" for Notepad.

You can use existing cmdlets to create these shortcuts as well. Consider the following example**: Set-Alias -Name ls -Value Get-ChildItem**.

This example creates an alias "ls" for the Get-ChildItem cmdlet, which is similar to the "ls" command in UNIX.

21) Invoke-Command: Runs commands on local and remote computers

The Invoke-Command cmdlet in PowerShell allows users to run commands on local and remote computers. This cmdlet is useful for managing multiple systems from a single console.

Using Invoke-Command, you can run a command on a single remote computer with the -ComputerName parameter.

Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Process }

This command retrieves the list of processes from the remote computer named Server01.

To run commands on multiple computers simultaneously, list the computer names separated by commas:

Invoke-Command -ComputerName Server01, Server02 -ScriptBlock { Get-Service }

This example fetches the running services from both Server01 and Server02.

For repetitive tasks that require multiple related commands, Invoke-Command can be used with a PSSession.

$session = New-PSSession -ComputerName Server01
Invoke-Command -Session $session -ScriptBlock { Get-EventLog -LogName Security }

Credentials can be specified if the current user doesn't have permission on the remote machine. Use the -Credential parameter:

$cred = Get-Credential
Invoke-Command -ComputerName Server01 -ScriptBlock { Get-Process } -Credential $cred

This command prompts for a username and password.

To execute a local script on a remote machine, copy the script using Copy-Item and then call it via Invoke-Command. This ensures scripts run smoothly:

**Copy-Item -Path .\script.ps1 -Destination \\Server01\c$\temp
Invoke-Command -ComputerName Server01 -ScriptBlock { & "C:\temp\script.ps1" }**

22) Test-Connection: Sends ICMP echo request packets (pings) to one or more computers

The Test-Connection cmdlet is a PowerShell cmdlet that sends ICMP echo request packets, commonly known as pings, to one or more remote computers. This cmdlet is used to check network connectivity. It can help determine if a particular computer can be reached over an IP network.

Using the -Count parameter, users can specify the number of echo requests to send.

Test-Connection -ComputerName Server01 -Count 4

If users need to perform continuous monitoring, they might use a loop. However, PowerShell's Test-Connection does not have an indefinite option like the traditional ping -t. Instead, a loop structure can be utilized for this purpose:

while ($true) { Test-Connection -ComputerName Server01 -Count 1 }

Test-Connection supports asynchronous pinging using the -AsJob parameter. This allows the task to run in the background, freeing up the console for other commands:

Test-Connection -ComputerName Server01 -Count 4 -AsJob

Another useful feature is specifying different source and destination computers. This can help test connectivity from multiple points:

Test-Connection -Source Server01 -ComputerName Server02

23) Start-Process: Starts one or more processes on the local computer

The Start-Process cmdlet is used in PowerShell to start one or more processes on the local computer. It creates a new process that inherits the environment variables from the current session.

To specify the program that runs, you can enter an executable file, script file, or any file that can be opened by a program on the computer.

For example, to run a batch file without displaying a window, use:

Start-Process -FilePath "C:\temp\example.bat" -Wait -WindowStyle Hidden

This command starts the example.bat file in a hidden window and waits for it to finish.

You can also print a text file using Start-Process:

Start-Process -FilePath "myfile.txt" -WorkingDirectory "C:\PS-Test" -Verb Print

This command prints the myfile.txt file located in the C:\PS-Test directory.

24) Stop-Process: Stops one or more running processes

The Stop-Process cmdlet is used to stop one or more processes running on a computer. It can identify processes by their name or process ID (PID). This cmdlet is useful for ending unresponsive programs.

To stop a process by its name, use this format:

Stop-Process -Name "notepad"

This command stops all instances of Notepad.

Another way to stop a process is by using its PID. To do this, you need to know the PID of the process:

Stop-Process -Id 1234

To ensure safety, use the -Confirm parameter. This parameter will prompt you to confirm before stopping the process:

Stop-Process -Id 1234 -Confirm

If you're unsure whether a process is running and you want to stop it only if it is running, you can use:

ps notepad -ErrorAction SilentlyContinue | Stop-Process -PassThru

This checks if Notepad is running and stops it only if it is. If Notepad is not running, the command does nothing.

25) Get-WmiObject: Gets instances of WMI classes or information about the available classes

The Get-WmiObject cmdlet in PowerShell retrieves instances of WMI (Windows Management Instrumentation) classes or information about available WMI classes.

Get-WmiObject is useful for querying system information and managing Windows systems.

For example, to find the operating system version on a local machine, use:

Get-WmiObject -Class Win32_OperatingSystem

To target a remote computer, use the -ComputerName parameter:

Get-WmiObject -Class Win32_OperatingSystem -ComputerName "RemotePC"

Listing all available classes within a namespace can be done with:

Get-WmiObject -List -Namespace "root\cimv2"

Using Get-WmiObject with specific namespaces and filters can refine searches.

Get-WmiObject -Query "SELECT * FROM Win32_LogicalDisk WHERE DeviceID = 'C:'"

26) Set-Variable: Sets the value of a variable

Set-Variable in PowerShell allows users to set the value of a variable. This cmdlet provides different options to control the variable's scope and behavior.

To set a variable, use the syntax Set-Variable -Name "variableName" -Value "value".

Set-Variable -Name "desc" -Value "A description"

This command sets the variable desc to the value "A description".

Set-Variable has additional parameters.

Set-Variable -Name "globalVar" -Value "Global Value" -Scope Global

For instance, the -Scope parameter lets you define the variable's scope. If you want a global variable, you can specify -Scope Global.

Set-Variable -Name "constVar" -Value "Constant Value" -Option Constant

The -Option parameter controls the variable’s behavior. You can make a variable read-only or even constant, meaning it cannot be changed or deleted.

Set-Variable -Name "processes" -Value (Get-Process)

This cmdlet also handles different data types. You may set a variable to the output of a command, like so:

This command stores the current system processes in the processes variable.

Users can retrieve the value of a set variable using the Get-Variable cmdlet.

Get-Variable -Name "desc"

27) Get-Variable: Gets the value of a variable

The Get-Variable cmdlet retrieves PowerShell variables from the current session. It offers a way to access variable names and values quickly.

Using the -Name parameter, users can specify which variables to display.

For example, **Get-Variable -Name MyVar** fetches the variable named "MyVar".

A useful feature is the -ValueOnly parameter. It returns only the values of the variables.

For instance, **Get-Variable -Name MyVar -ValueOnly** shows the value stored in "MyVar".

You can also filter variables by pattern using wildcards.

For example, **Get-Variable -Name m*** retrieves variables that start with the letter "m". This is helpful for getting multiple variables at once.

Another example is retrieving variables starting with multiple letters.

By using -Include, such as **Get-Variable -Include M*, P***, you can gather variables that start with "M" and "P".

28) Export-Csv: Converts objects into a series of CSV items

The Export-Csv cmdlet in PowerShell converts objects into a series of CSV strings and saves them in a specified text file. This is useful for

exporting data in a format that can be easily opened with spreadsheet applications.

When using Export-Csv, each object is represented by a comma-separated line in the CSV file. The cmdlet includes all properties of the objects as separate columns.

To use the cmdlet, you need to pass the objects you want to export and specify the output file.

For example:

Get-Process | Export-Csv -Path "processes.csv" -NoTypeInformation

In this example, the Get-Process cmdlet retrieves all running processes. The output is then exported to "processes.csv".

The -NoTypeInformation parameter ensures that no type information is included in the file, making it cleaner.

You can also include specific properties from an object.

For example:

Get-Process | Select-Object Name, CPU | Export-Csv -Path "processes.csv" -NoTypeInformation

This command selects only the Name and CPU properties of processes, and then exports them. This makes your CSV file more concise and focused.

If you wish to append data to an existing CSV file, you can use the -Append parameter. This prevents overwriting the file:

Get-Service | Export-Csv -Path "services.csv" -Append -NoTypeInformation

This cmdlet is crucial for scripts that need to create reports or log data in a structured format.

29) Import-Csv: Creates table-like custom objects from the items in a CSV file

The Import-Csv cmdlet in PowerShell is used to read data from a CSV file and create custom objects from that data. Each column in the CSV file becomes a property of the custom object, making it easy to work with the data programmatically.

To import a CSV file, you can use a simple command.

For example, **Import-Csv -Path "C:\data\users.csv"**.

This cmdlet is extremely useful for processing data files, like user lists, configuration settings, or log files.

Each row in the CSV file becomes an individual object. You can then access the properties of these objects in your scripts.

For instance, if the CSV has columns "Name," "Email," and "Age," you can access these as properties.

Additionally, Import-Csv works seamlessly with other cmdlets. It can be piped to Where-Object, Sort-Object, or Select-Object to filter, sort, or select specific data.

You can also specify the column header row and delimiter. By default, it assumes the first row contains headers and uses a comma as a delimiter.

A common use case is importing user data.

For example, **Import-Csv -Path "C:\data\users.csv" | ForEach-Object { $_.Name }** would iterate through each user and output their names.

30) Measure-Object: Measures the properties of objects

The Measure-Object cmdlet in PowerShell helps calculate the properties of objects. This cmdlet can count objects or calculate the sum, average, minimum, and maximum values of numeric properties.

For basic counting, command Measure-Object can count the number of objects returned by other cmdlets.

For example, using **Get-ChildItem | Measure-Object** counts files and folders in a directory.

You can measure specific properties too. To get the sum of file sizes in a directory, you could use:

Get-ChildItem | Measure-Object -Property Length -Sum

To calculate average, minimum, and maximum values, you can utilize the -Average, -Minimum, and -Maximum parameters.

For example:

Get-ChildItem | Measure-Object -Property Length -Average -Minimum -Maximum

The cmdlet can also measure strings. It can count the characters, words, and lines in text files.

For instance, to count words:

Get-Content file.txt | Measure-Object -Word

31) ConvertTo-Json: Converts an object to a JSON-formatted string

The ConvertTo-Json cmdlet in PowerShell is used to convert .NET objects into JSON-formatted strings.

To use ConvertTo-Json, pass the object you want to convert.

For example:

```
$object = [PSCustomObject]@{Name="John"; Age=30}
$json = $object | ConvertTo-Json
Write-Output $json
```

This converts the object with properties Name and Age into a JSON string.

ConvertTo-Json has several parameters. The -Depth parameter sets the levels of the object to include. The default is 2. Increase this number if your objects are complex.

For instance:

ConvertTo-Json -InputObject $object -Depth 3

Use the -Compress parameter to remove spaces and line breaks, which minimizes file size:

ConvertTo-Json -InputObject $object -Compress

DateTime objects convert to JSON using the ISO 8601 format. This ensures consistent date-time representations:

```
$date = Get-Date
$jsonDate = $date | ConvertTo-Json
Write-Output $jsonDate
```

This cmdlet can handle both single objects and arrays. To ensure arrays are treated correctly use the -AsArray parameter when necessary:

```
$array = @($object)
$jsonArray = $array | ConvertTo-Json -AsArray
Write-Output $jsonArray
```

32) ConvertFrom-Json: Converts a JSON-formatted string to an object

The ConvertFrom-Json cmdlet is used in PowerShell to convert a JSON-formatted string into a PowerShell object.

When executing ConvertFrom-Json, the cmdlet reads the JSON string and creates an object. Each property in the JSON string becomes a

property in the resulting object. This makes handling and manipulating JSON data straightforward in PowerShell.

For example:

$jsonString = '{"name": "John", "age": 30, "city": "New York"}'
$personObject = $jsonString | ConvertFrom-Json

In this example, $personObject will have properties name, age, and city with values "John", 30, and "New York" respectively.

An important feature of ConvertFrom-Json is its ability to handle nested JSON structures. This enables it to work with complex data formats often seen in real-world applications.

Example with nested JSON:

$nestedJson = '{"employee": {"name": "Jane", "details": {"age": 25, "city": "Los Angeles"}}}'
$employeeObject = $nestedJson | ConvertFrom-Json

Here, $employeeObject will contain another object as a property: employee.details.

33) Out-File: Sends output to a file

Out-File is a useful cmdlet in PowerShell for sending output directly to a file. This cmdlet allows you to store the output that would normally appear on the console into a specified file.

For basic usage, you can pipe the output of a command to Out-File and specify the file path.

Get-Process | Out-File -FilePath C:\temp\processlist.txt

This command retrieves a list of processes and writes it to "processlist.txt" in the "C:\temp" directory.

When dealing with large outputs, you might want to append data to an existing file rather than overwrite it. The -Append parameter achieves this.

Get-Service | Out-File -FilePath C:\temp\services.txt -Append

Out-File also provides control over the encoding of the output file. Using the -Encoding parameter, you can specify the type of encoding, such as UTF8 or ASCII.

Get-Command | Out-File -FilePath C:\temp\commands.txt -Encoding UTF8

34) Write-Output: Sends Output to the Pipeline

Write-Output is a PowerShell cmdlet used to send output to the next command in the pipeline.

When Write-Output is the last cmdlet in the pipeline, the result is displayed in the console.

This cmdlet can handle strings, variables, and objects.

For example, **Write-Output "Hello, World!"** sends the string "Hello, World!" to the console.

35) Write-Host: Writes customized output to the console

The Write-Host cmdlet in PowerShell is used to print text to the console. This command is ideal for displaying messages, such as when prompting a user for input.

Write-Host outputs directly to the console. This makes it different from Write-Output, which sends data to the pipeline.

For example, **Write-Host "Hello, World!"** will display the text "Hello, World!" on the console.

Customization is one of the strengths of Write-Host. You can change the color of the text using the -ForegroundColor parameter.

For instance, **Write-Host "Hello, World!" -ForegroundColor Green** will display the text in green.

You can also change the background color with the -BackgroundColor parameter.

For example, **Write-Host "Hello, World!" -ForegroundColor Yellow -BackgroundColor Blue** will show the text in yellow with a blue background.

To display a variable, use it inside the Write-Host command.

For example, if $exampleVar = "PowerShell", then Write-Host "This is $exampleVar" will display "This is PowerShell".

36) Read-Host: Reads a line of input from the console

The Read-Host cmdlet in PowerShell is used to pause script execution and prompt the user for input.

It can be very helpful in scripts that require user interaction. For example, one can use it to ask for a user's name and then display a personalized greeting.

A basic usage of Read-Host looks like this:

```
$name = Read-Host "Enter your name"
Write-Output "Hello, $name!"
```

In this example, the script asks for the user's name and stores the input in the $name variable. Then it greets the user with their name.

For added security, Read-Host can mask the input, useful for passwords.

```
$password = Read-Host "Enter your password" -AsSecureString
```

While this input is hidden, it is also stored as a SecureString object. This is beneficial for keeping sensitive information secure.

For use cases where the masked password needs to be manipulated as plain text, the -MaskInput parameter can be used in PowerShell 7.1 and later versions.

$password = Read-Host "Enter your password" -MaskInput

37) Send-MailMessage: Sends an email message

The Send-MailMessage cmdlet in PowerShell is used to send email messages.

With Send-MailMessage, you need to specify an SMTP server. This is critical for the cmdlet to work. You'll also need the sender's and recipient's email addresses.

Here's a basic example:

Send-MailMessage -To "recipient@example.com" -From "sender@example.com" -Subject "Test email" -Body "Hello from PowerShell!" -SmtpServer "smtp.example.com"

You can include attachments with your email. Just use the -Attachments parameter to specify the file path.

To send an email with an attachment:

Send-MailMessage -To "recipient@example.com" -From "sender@example.com" -Subject "Report" -Body "Please find the report attached." -Attachments "C:\path\to\file.txt" -SmtpServer "smtp.example.com"

Adding credentials is often necessary, especially when sending through an authenticated SMTP server. Use the Get-Credential cmdlet to prompt for credentials.

Example with credentials:

$cred = Get-Credential
Send-MailMessage -To "recipient@example.com" -From "sender@example.com" -Subject "Secure email" -Body "Secured email with credentials" -SmtpServer "smtp.example.com" -Credential $cred

To send to multiple recipients, list the email addresses separated by commas.

Example with multiple recipients:

Send-MailMessage -To "recipient1@example.com,recipient2@example.com" -From "sender@example.com" -Subject "Group email" -Body "Hello everyone" -SmtpServer "smtp.example.com"

This cmdlet also supports HTML-formatted emails. Use the -BodyAsHtml switch to enable this.

Example with HTML:

Send-MailMessage -To "recipient@example.com" -From "sender@example.com" -Subject "HTML email" -Body "<h1>Hello</h1> This is an HTML email. " -BodyAsHtml -SmtpServer "smtp.example.com"

38) Comparison Operators: eq, ne, gt, lt, etc.

PowerShell uses comparison operators to compare values. These operators help determine if values are equal, not equal, greater than, or less than.

The -eq operator checks if two values are equal. For example, 5 -eq 5 returns True.

The -ne operator checks if two values are not equal. For instance, 6 -ne 5 returns True.

The -gt operator checks if one value is greater than another. For example, 10 -gt 5 returns True.

The -lt operator checks if one value is less than another. For instance, 3 -lt 5 returns True.

These operators can also compare strings. For example, "hello" -eq "hello" returns True.

Case sensitivity can be controlled. The -ceq and -cne operators are case-sensitive versions of -eq and -ne.

PowerShell offers more comparison operators such as -ge (greater than or equal to) and -le (less than or equal to).

Date comparisons are also possible. For example, Get-Date -eq [datetime]"2024-06-18" checks if the current date is June 18, 2024.

Using these operators with different data types, such as integers, strings, and dates, makes them very versatile.

39) ForEach-Object: Performs an operation on each item in a collection

The ForEach-Object cmdlet in PowerShell is used to apply an operation to each item in a collection of objects. It can handle input objects piped to it or specified using the InputObject parameter

A common use of ForEach-Object is to manipulate a list of files. For example, to rename all .txt files in a directory to .bak, one would use:

Get-ChildItem -Path "C:\logs" -Filter *.txt | ForEach-Object { Rename-Item $_.FullName ($_.FullName -replace '\.txt$','.bak') }

Here's another example of ForEach-Object. It updates the attribute of users in an Active Directory group:

Get-ADUser -Filter * | ForEach-Object { Set-ADUser -Identity $_ -Title "Engineer" }

40) Where-Object: Selects objects based on their property values

Where-Object is one of PowerShell's key cmdlets for filtering data. It allows users to select objects from a collection based on specified property values.

To filter objects, you can use this. For example, to get files created after a certain date, you could use:

Get-ChildItem | Where-Object { $_.CreationTime -gt '2024-01-01' }

Another way is to use the property value approach. This method provides a cleaner syntax for straightforward comparisons. For example, to select objects where a property equals a specified value:

Get-Process | Where-Object -Property ProcessName -EQ 'notepad'

Where-Object is also useful for complex filtering with multiple conditions. This can be done using -and and -or operators within the script block:

Get-EventLog -LogName System | Where-Object { $_.EventID -eq 6005 -or $_.EventID -eq 6006 }

For more advanced usage, users might want to use an alias like ? which represents Where-Object to make the command shorter:

Get-Service | ? { $_.Status -eq 'Running' }

41) Start-Sleep: Suspends the activity in a script or session for the specified period

The Start-Sleep cmdlet in PowerShell is used to pause the execution of a script or session for a specified duration. This cmdlet is helpful when you need to wait for a task to complete or pause before repeating an operation.

To pause a script for a certain amount of time, use the -Seconds parameter. For example, **Start-Sleep -Seconds 10** will pause the script for 10 seconds.

If a more precise time is needed, the -Milliseconds parameter can be used. For example, **Start-Sleep -Milliseconds 500** will suspend the script for half a second. This is useful when small delay intervals are required.

42) Select-Object: Selects specified properties of an object or set of objects

The Select-Object cmdlet is essential for working with properties in PowerShell. It allows users to select specific properties from an object or a collection of objects.

For example, when working with files, users might only be interested in the Name and Length properties. By using Select-Object, they can limit the output to these properties only.

To select properties, the -Property parameter is used. For instance, to display only the Name and CreationTime of a file, use:

Get-ChildItem | Select-Object -Property Name, CreationTime

This command pipes the output of Get-ChildItem to Select-Object, which then selects and displays only the Name and CreationTime properties.

Select-Object can also return unique objects using the -Unique parameter. This is useful when dealing with large datasets where duplicate entries need to be removed.

For example, to remove duplicates based on the Name property:

Get-ChildItem | Select-Object -Property Name -Unique

Additionally, users can select the first or last number of objects in a collection using the -First or -Last parameters

To select the first three files in a directory:

Get-ChildItem | Select-Object -First 3

Or, to select the last three files:

Get-ChildItem | Select-Object -Last 3

43) Sort-Object: Sorts Objects by Property Values

The Sort-Object cmdlet in PowerShell sorts objects based on their properties. Users can specify one or more properties for sorting.

By default, it sorts in ascending order but can be adjusted to sort in descending order if needed.

For instance, to sort an array of files by their last modified date:

Get-ChildItem | Sort-Object LastWriteTime

If the property isn't specified, Sort-Object uses any default properties of the input object.

If no default properties exist, PowerShell attempts to sort the objects directly.

To sort by multiple properties, include them in the cmdlet. An example of sorting by both "Name" and "Length":

Get-ChildItem | Sort-Object Name, Length

To sort data in descending order, use the -Descending parameter:

Get-ChildItem | Sort-Object Name -Descending

You can also sort custom objects. Suppose you have an array of custom objects with properties "Name" and "Age":

```
$people = @(
  [pscustomobject]@{Name='Alice'; Age=30},
  [pscustomobject]@{Name='Bob'; Age=25}
)
$people | Sort-Object Age
```

44) Group-Object: Groups objects that contain the same value for specified properties

The Group-Object cmdlet in PowerShell allows users to group objects based on a specified property. This cmdlet is particularly useful when you need to organize data for easier analysis.

For instance, if you have a list of processes, you can group them by their process name. The cmdlet will then create groups of processes that share the same name.

Here's a simple example:

Get-Process | Group-Object -Property ProcessName

You can also group by multiple properties. When specifying more than one property, objects are first grouped by the first property, then further grouped within those groups by the second property.

Example:

Get-Process | Group-Object -Property ProcessName, CPU

This command groups processes first by name and then by CPU usage within each name group.

The cmdlet can return the groups as a hash table if needed. The keys of the hash table will be the property values by which the objects are grouped.

Syntax for hash table:

Get-Process | Group-Object -Property ProcessName -AsHashTable

To filter groups with more than one member, you can use the Where-Object cmdlet. This technique is useful for identifying duplicate entries in a dataset.

Example:

$array | Group-Object | Where-Object { $_.Count -gt 1 }

This groups the elements of an array and then selects groups containing more than one item.

45) Get-Date: Gets the current date and time

The Get-Date cmdlet is used in PowerShell to retrieve the current date and time. It automatically uses the system’s culture settings to format the output.

For example, running Get-Date without any parameters will show the current date and time of the system. This output includes details like the day of the week, month, day, and time.

Get-Date

You can also customize the date and time format. For instance, to get just the date in a short format, you can use the -Format parameter.

Get-Date -Format "MM/dd/yyyy"

The Get-Date cmdlet can perform date arithmetic. Adding days to the current date can be done using the -AddDays parameter.

(Get-Date).AddDays(5)

For a detailed list of properties, you can pipe Get-Date to Format-List.

Get-Date | Format-List

The cmdlet is also useful for creating specific date and time objects. By setting parameters like -Year, -Month, and -Day, users can define a particular date.

Get-Date -Year 2025 -Month 12 -Day 31

46) New-Object: Creates an instance of a .NET Framework or COM object

The New-Object cmdlet in PowerShell is used to create instances of .NET Framework or COM objects.

To create a .NET Framework object, users typically provide the fully qualified name of the class. For instance, to create an event log object, one would use:

New-Object -TypeName System.Diagnostics.EventLog

COM objects can also be created with New-Object. This requires the -ComObject parameter and the ProgID of the desired COM object. For example:

New-Object -ComObject Scripting.FileSystemObject

This cmdlet simplifies object creation, making it easier for PowerShell users to work with various .NET and COM libraries.

47) Out-String: Converts objects to strings

The Out-String cmdlet in PowerShell is used to convert objects into strings. By default, it gathers the strings and returns them as a single string. This can be useful when you need to view output in a text format.

You can also use the -Stream parameter with Out-String. This makes the cmdlet return one string at a time instead of combining them all into one. This is handy when dealing with large outputs.

Here's a simple example:

Get-Process | Out-String

This command lists all running processes and converts the list to a single string.

The -Width parameter can be used to specify the width of the output:

Get-Process | Out-String -Width 50

This sets the width of each line in the output to 50 characters.

(Get-Date).ToString()

This method converts the datetime object to its string form.

48) ConvertTo-SecureString

The ConvertTo-SecureString cmdlet in PowerShell is used to convert plain text into a secure string. This secure string is encrypted and kept confidential. It is often used when handling sensitive information like passwords.

A common use of ConvertTo-SecureString is converting user input into a secure format. For example, to convert a plain text password, the user can enter:

$SecurePassword = ConvertTo-SecureString "P@ssw0rd" -AsPlainText -Force

This ensures the password is stored in an encrypted format.

The cmdlet can also handle encrypted strings. Use ConvertTo-SecureString with the -Key or -SecureKey parameters to re-encrypt data. Here's how to use it with an encryption key stored in a variable:

$key = (3..14) # As an example encryption key
$SecureString = ConvertTo-SecureString "EncryptedData" -Key $key

For interactive prompts, the cmdlet can work with Read-Host to allow users to enter passwords securely:

$SecurePassword = Read-Host "Enter password" -AsSecureString

The cmdlet can also be used inversely to obtain secure strings from encrypted files. First, encrypt the string using ConvertFrom-SecureString, then read the encrypted data back:

$EncryptedString = ConvertFrom-SecureString -Key $key
$SecurePassword = ConvertTo-SecureString $EncryptedString -Key $key

49) Clear-Host: Clears the screen

The Clear-Host cmdlet is used to clear the screen in PowerShell. It wipes all previous output and commands from the display, leaving a clean slate.

Users can enter **Clear-Host** directly in the PowerShell command prompt. This command removes text from the active console but does not affect saved results or items in the session.

There is also an alias for Clear-Host, which is cls. Typing cls achieves the same result as Clear-Host, making it a useful shortcut.

For those who want to clear the screen without any prompts, the -Force flag can be used. Adding -Force bypasses any confirmation messages, instantly clearing the screen.

To see the aliases for Clear-Host, you can use the Get-Alias command. For example, Get-Alias cls will show that cls maps to Clear-Host.

50) Write-Debug: Writes a debug message to the console

Write-Debug is a cmdlet in PowerShell that sends debug messages to the console. It is useful for troubleshooting and understanding what a script is doing at specific points.

To use Write-Debug, the script should include the cmdlet in the relevant places where you want to output a debug message. An example is:

Write-Debug "Starting script execution"

For these debug messages to appear in the console, the script must be run with the -Debug parameter. This parameter tells PowerShell to display the debug output.

.\MyScript.ps1 -Debug

51) Write-Verbose: Writes a verbose message to the console

The Write-Verbose cmdlet in PowerShell allows for the display of detailed messages during script execution. This is especially useful for debugging or tracking the progress of scripts.

By default, these verbose messages are not shown. They will only appear if the -Verbose parameter is added when running the script.

For example, to write a message saying "Searching the Application Event Log," you would use:

Write-Verbose -Message "Searching the Application Event Log."

To actually see this message, add -Verbose when calling the script:

.\MyScript.ps1 -Verbose

When executed, the console will display:

VERBOSE: Searching the Application Event Log.

Messages can also include multi-line information. For instance:

Write-Verbose -Message "Function Start: `nTime: $(Get-Date)"

52) Write-Warning: Writes a Warning Message

The Write-Warning cmdlet in PowerShell is used to display a warning message.

To use Write-Warning, simply pass the message string as an argument. For example:

Write-Warning "This is a test warning."

This command will output: "WARNING: This is a test warning."

It can receive input through the pipeline. For example, a string stored in a variable can be piped to the cmdlet:

$string = "Pipeline warning message."
$string | Write-Warning

This will display the warning message from the variable.

The Write-Warning cmdlet also supports the -Message parameter, which is explicitly used to define the warning message:

Write-Warning -Message "Explicit parameter warning."

This ensures the intention is clear in scripts where the parameters are specified.

When executing scripts that might generate warnings, the WarningAction common parameter can be used. This parameter allows control over the behavior when a warning message is issued. For example:

Write-Warning "This will inquire warning." -WarningAction Inquire

This will prompt the user with options when the warning is displayed.

53) Write-Error: Writes an error message

The Write-Error cmdlet in PowerShell sends an error message to the error stream. This is useful for logging and debugging scripts. It is often used by developers and system administrators.

The basic syntax is **Write-Error -Message "Your error message"**. The -Message parameter specifies the error message. You can also include additional details using other parameters.

For example, Write-Error -Message "File not found". This command sends "File not found" to the error stream. It's a simple but effective way to alert users or log errors.

You can add more detail by using Write-Error -Message "File not found" -Category NotSpecified -ErrorId 404. The -Category and -ErrorId parameters provide extra context, which can be helpful in larger scripts.

In some cases, you might want to terminate the script after writing an error. You can do this with the Stop parameter: Write-Error -Message "Critical error" -Stop. This will stop the script execution immediately.

54) Remove-Variable: Removes a variable and its value

The Remove-Variable cmdlet in PowerShell is used to delete a variable and its value from the current session. This cmdlet is handy for cleaning up variables that are no longer needed. It ensures that variables do not take up memory unnecessarily.

To use Remove-Variable, simply specify the name of the variable you want to delete. For example:

Remove-Variable -Name myVariable

This command removes the variable myVariable and its associated value from the session.

It is important to note that Remove-Variable cannot delete variables that are set as constants or those owned by the system.

You can also remove multiple variables at once. Use the cmdlet with an array of variable names:

Remove-Variable -Name var1, var2, var3

This command deletes var1, var2, and var3 in a single execution.

Another option is to prompt for confirmation before deleting a variable. Adding the -Confirm parameter ensures you are prompted:

Remove-Variable -Name myVariable -Confirm

For those who prefer shorthand, Remove-Variable has an alias rv:

rv -Name myVariable

Using Remove-Variable is an efficient way to manage variables, especially in scripts with many temporary variables.

55) Format-Table: Display information in tabular format

The Format-Table cmdlet in PowerShell helps display command output in a table format. This cmdlet is useful for organizing data into rows and columns.

To create a table, use the Format-Table cmdlet followed by the properties you want to include. For instance, **Get-Process | Format-Table Name, CPU, ID** will display a table of processes showing just the Name, CPU, and ID.

The AutoSize parameter adjusts the column width to fit the data. This ensures that the table is easy to read and that no information is hidden. For example, **Get-Process | Format-Table Name, CPU, ID -AutoSize** formats the same table with adjusted column widths.

Using the Property parameter, you can select specific properties to display in the table. For example, **Get-Service | Format-Table -Property Name, Status** shows only the Name and Status properties of services.

The GroupBy parameter is useful when you want to group rows by a common value. For example, **Get-Process | Format-Table Name, CPU -GroupBy Company** groups processes by the company that created them.

To format table output in shorter columns, the Wrap parameter can be used. **Get-Process | Format-Table Name, CPU, -Wrap** ensures that text wraps within columns, making it easier to read.

56) Format-List: Format Output as a list

The Format-List cmdlet in PowerShell formats the output of a command as a list of properties, with each property shown on a separate line. This is especially useful for objects containing many fields.

To display all properties of an object, use **Format-List -Property ***. This command is helpful when you need a complete view of the data.

Another common use involves displaying only selected properties. You can specify the properties you want to see using **Format-List -Property Name, Status, CreatedDate**.

One of the advantages of this cmdlet is its ability to handle wildcard characters. This lets users display properties that match certain patterns. For example, **Format-List -Property *date*** would show all properties containing the word “date.”

57) Compare-Object: Compare two sets of obejcts

The Compare-Object cmdlet in PowerShell lets users compare two sets of objects. This is useful for finding differences between arrays, lists, or object arrays.

Compare-Object works by identifying which properties exist only in each set. For instance, it can show what's missing, added, or changed between two data sets.

To compare files, services, or processes, users can use Compare-Object. This cmdlet handles various types, making it versatile for different tasks.

When comparing, the results indicate where properties appear. A property in the reference set is marked with "=>", while a property in the difference set is marked with "<=".

Here's a basic example:

Compare-Object -ReferenceObject $Set1 -DifferenceObject $Set2

In this example, $Set1 is compared to $Set2, and the output reveals the differences. This makes it clear which items only appear in one of the sets.

58) Get-Job: Gets Windows PowerShell background jobs

The **Get-Job** cmdlet is essential for managing background jobs in PowerShell. Users often employ background jobs to run complex commands that might take a long time.

By using **Get-Job**, one can check on the status of these jobs. They can be running, completed, or even failed.

Background jobs run commands asynchronously. This means you can run other commands while a job is processing.

Starting with PowerShell 3.0, **Get-Job** can handle custom job types. This includes workflow jobs and scheduled jobs.

To use **Get-Job**, simply type Get-Job in the PowerShell console. This command will list all the current background jobs.

For more detailed information on a specific job, you can use the -Id or -Name parameter. For example**, Get-Job -Id 1** will give detailed info about job with ID 1.

59) Start-Job: Starts a Windows PowerShell background job

The Start-Job cmdlet is essential for tasks requiring background execution on a local computer. This cmdlet runs commands without interrupting the current session, allowing users to continue other tasks.

A background job starts with a Start-Job command, returning a job object. This job object provides details about the running job, including its status and results.

To start a simple background job, you can use:

Start-Job -ScriptBlock { Get-Content C:\largefile.txt }

This command retrieves the contents of a large file without blocking the PowerShell session.

You can also run scripts as background jobs:

Start-Job -FilePath "C:\Scripts\MyScript.ps1"

This command executes MyScript.ps1 in the background.

Jobs can be controlled using various parameters. The -Name parameter assigns a name to the job:

Start-Job -ScriptBlock { Get-Process } -Name "GetProcesses"

This is useful for identifying jobs later.

To check on running jobs, use:

Get-Job

To retrieve job results, use:

Receive-Job -Id <JobId>

Replace <JobId> with the actual job ID.

After a job completes, it's a best practice to remove it:

Remove-Job -Id <JobId>

Proper cleanup ensures that system resources are available.

60) Stop-Job: Stops a Windows PowerShell background job

The Stop-Job cmdlet ends PowerShell background jobs that are currently running. These jobs might have been started with the Start-Job cmdlet.

When using Stop-Job, you can specify jobs by their name, ID, instance ID, or state. A job object can be passed directly to the Stop-Job cmdlet.

Stopping a job does not remove it. To delete a job after stopping it, use the Remove-Job cmdlet. This separation ensures that users can review job outcomes before deletion.

For instance, to stop a specific job by ID:

Stop-Job -Id 1

Stopping all jobs:

Get-Job | Stop-Job

To stop jobs with a specific state like “Failed”:

Get-Job | Where-Object { $_.State -eq 'Failed' } | Stop-Job

Users should always ensure they stop unnecessary jobs to free up resources. Proper management of background jobs helps maintain system stability and performance.

61) Add-History: Add commands to session history

The **Add-History** cmdlet in PowerShell adds commands to the session history. Each command added to the session history includes its order of execution, status, and start and end times.

Users can leverage **Add-History** to append custom commands to their session history. This can be particularly useful for automation scripts or custom workflows.

Syntax

Add-History -InputObject <PSObject>

To add commands imported from an XML file to the session history:

$commands = Import-Clixml "C:\path\to\commands.xml"

Add-History -InputObject $commands

61) Get-History: Gets a list of the commands entered during the current session

The Get-History cmdlet in PowerShell retrieves a list of commands entered during the current session.

Using Get-History displays details including the command, its status, and execution times. This cmdlet can be particularly useful for debugging and verifying the sequence of executed commands.

To use Get-History, simply type the cmdlet with no parameters:

Get-History

This returns a list of commands executed in the current session. If you need only the most recent commands, you can use the -Count parameter:

Get-History -Count 5

This command displays the five most recent entries. For more detailed tracking, you can use the -ID parameter to fetch commands by their ID numbers:

Get-History -ID 12

This retrieves the command with ID 12 from the history.

62) Invoke-History: Runs a command or commands from the session history

The Invoke-History cmdlet allows users to re-execute commands that have been run previously without retyping them.

To use Invoke-History, users can pass objects from the Get-History cmdlet, or identify commands by their ID number. For instance, to re-run the last command, simply type **Invoke-History -Id 1**.

If you want to run multiple commands, separate their IDs with commas: Invoke-History 1, 3, 5.

The Get-History cmdlet helps find the ID numbers of commands in the session. Just type Get-History and you will see a list of previous commands along with their IDs.

63) Clear-History: Deletes the session history

The Clear-History cmdlet in PowerShell deletes command history from the current session.

To use it, simply type Clear-History at the command prompt. This will remove all commands entered during the current session.

If you need to delete specific commands, you can use strings or wildcards. For example, **Clear-History -CommandLine "Get-*"**. This deletes all commands starting with “Get”.

Another option is to specify how many recent commands to delete. For instance, **Clear-History -Count 5** will remove the last five commands.

To clear commands with spaces, use single quotes. For example, Clear-History -CommandLine 'New-Item'. This ensures the command is interpreted correctly.

64) Get-PSDrive: Gets the drives in the current session

The Get-PSDrive cmdlet retrieves the drives available in the current PowerShell session. You can use this cmdlet to get a specific drive or all drives at once.

This cmdlet provides details on Windows logical drives, including drives mapped to network shares. It also includes drives exposed by PowerShell providers, such as the Certificate, Function, and Alias drives.

To get a list of all drives, simply run:

Get-PSDrive

If you need information about a specific drive, you can specify the drive’s name. For example:

Get-PSDrive -Name C

PowerShell providers also expose other drives, like the HKLM and HKCU registry drives. This can be helpful for accessing and managing registry hives directly within your PowerShell session.

Drives mapped to network shares display their UNC path. For example, a drive mapped to \\server\shares\dir1 can be accessed as:

New-PSDrive -Name Z -PSProvider FileSystem -Root \\server\shares\dir1

Get-PSDrive -Name Z

By default, Get-PSDrive retrieves all drives created in the session, giving a comprehensive view of what's currently mounted and accessible.

If you wish to filter by a specific PowerShell provider, use the -PSProvider parameter:

Get-PSDrive -PSProvider FileSystem

65) New-PSDrive: Creates a Windows PowerShell drive in the session

The New-PSDrive cmdlet in PowerShell is used for creating new drives. These drives can point to different locations like network shares, directories, or registry keys. The cmdlet allows for both temporary and persistent drive creation.

When creating a drive, you can specify the name, provider, and root. For example, **New-PSDrive -Name X -PSProvider FileSystem -Root \\Server\Share** maps a network share to drive X.

Temporary drives exist only in the current session. If you need the drive to be available in Windows Explorer, use the -Persist switch. This makes the drive mapping persistent across all sessions. An example would be **New-PSDrive -Name V -PSProvider FileSystem -Root \\VBoxSvr\Win11 -Persist**.

66) Remove-PSDrive: Deletes a Windows PowerShell drive from the session

The Remove-PSDrive cmdlet is used to delete a PowerShell drive from a session. It can remove drives created by the New-PSDrive cmdlet. This cmdlet is useful for cleaning up temporary drives.

Remove-PSDrive cannot delete Windows physical or logical drives. This cmdlet works only on PowerShell drives.

To use it, specify the name of the drive you want to remove. For example, to remove a drive named “S:”, the command would be:

Remove-PSDrive -Name S

In this case, “S:” is the name of the drive you wish to remove.

67) Get-EventSubscriber: Gets the event subscribers in the current session

The Get-EventSubscriber cmdlet retrieves all event subscribers in the current PowerShell session. An event subscriber is created when you use a Register event cmdlet like Register-ObjectEvent.

When you subscribe to an event, it gets added to your event queue. The Get-EventSubscriber cmdlet displays these events, allowing you to see which events are queued. This is useful for debugging and managing event-driven scripts.

To list all event subscribers in your current session, simply run:

Get-EventSubscriber

You can also filter events based on criteria such as name or source identifier. For example:

Get-EventSubscriber -SourceIdentifier "MyEvent"

If you want to unsubscribe from an event, use the Unregister-Event cmdlet. For instance:

Unregister-Event -SourceIdentifier "MyEvent"

This cmdlet works well for monitoring asynchronous events and helps maintain a clean event queue.

68) Start-Transcript: Starts recording a transcript of the session

The Start-Transcript cmdlet in PowerShell creates a record of commands typed by the user and the output displayed on the console.

When you run Start-Transcript, it begins recording everything you do in the PowerShell session.

To start recording a transcript, run the following command:

Start-Transcript -Path "C:\path\to\transcript.txt"

An important option is -Append, which adds new session logs to the existing file instead of overwriting it:

Start-Transcript -Path "C:\path\to\transcript.txt" -Append

PowerShell 5.0 and later versions can include the hostname in the transcript filename, adding more context to the log file.

To stop the transcript, run:

Stop-Transcript

This command finishes recording and closes the transcript file.

Ensure you run PowerShell as an administrator to capture all commands in the session:

Start-Transcript -Path "C:\path\to\admin-transcript.txt"

69) Stop-Transcript: Stops recording and saves the transcript

The Stop-Transcript cmdlet stops the recording of a PowerShell session transcript.

To use this cmdlet, you must first initiate a transcript with the Start-Transcript cmdlet. Once finished, run Stop-Transcript to save the transcript file.

70) Debug-Process: Attaches a Debugger to a Process

The Debug-Process cmdlet in PowerShell is used to attach a debugger to one or more running processes on a local computer.

To use this cmdlet, the process can be specified by its name or Process ID (PID).

For example, to attach a debugger to the PowerShell processes on the computer, the following command can be used:

Get-Process -Name "powershell" | Debug-Process

A specific process can be targeted by providing its PID:

Debug-Process -Id 1234

The Debug-Process cmdlet will attach the currently registered debugger to the specified process.

71) Checkpoint-Computer: Creates a system restore point

The Checkpoint-Computer cmdlet is used to create a system restore point on a local computer. This is useful for saving the system's current state before making significant changes to the system.

Using this cmdlet can help users revert their system back to a stable state if something goes wrong during or after modifications.

For example, the command **Checkpoint-Computer -Description "PreUpdate" -RestorePointType MODIFY_SETTINGS** will create a restore point called “PreUpdate”. This is especially helpful when installing new software or changing system settings.

System restore points created with this cmdlet are only supported on client versions such as Windows 10 and Windows 8. On Windows 10, users cannot create more than one restore point each day.

To check all current restore points on the system, users can utilize the **Get-ComputerRestorePoint** cmdlet. This can display details about each restore point, helping users decide which point to revert to if needed.

For performing the actual rollback to a previous state, the Restore-Computer cmdlet can be used along with a specific sequence number obtained from Get-ComputerRestorePoint.

72) Undo-Transaction: Undoes the active transaction

The Undo-Transaction cmdlet rolls back an active transaction in PowerShell. This cmdlet is useful when you need to discard changes made during a transaction and restore the data to its original state.

When executing Undo-Transaction, any modifications performed within the transaction are undone. This ensures that no unintended changes are left in the system.

Transactions in PowerShell can include multiple operations. These might involve creating, modifying, or removing items. By rolling back the transaction, all these activities are effectively canceled.

For example, a user might start a transaction, create a new item, and then decide to roll back the change. The command sequence would look like this:

Start-Transaction

New-Item -Path "HKCU:\Software\NewKey" -ItemType Directory

Undo-Transaction

In this example, the New-Item cmdlet creates a new directory within the registry. However, the Undo-Transaction cmdlet cancels this creation, leaving the registry unchanged.

73) ConvertTo-Html: Converts .NET objects into HTML

The ConvertTo-Html cmdlet in PowerShell converts .NET objects into HTML code. This cmdlet is useful for generating HTML representations of various data or objects, which can then be displayed in a web browser.

To use ConvertTo-Html, you simply pipe the output of a command to this cmdlet. For example, **Get-Process | ConvertTo-Html** converts the list of running processes into an HTML table.

It's also possible to customize the HTML output. You can specify properties to include, set the HTML page title, and choose between table or list formats. These customization options make it a versatile tool for various reporting needs.

For example, **Get-Service | ConvertTo-Html -Property Name, Status -Title "Services List"** generates an HTML page showing only the Name and Status of services, with the title "Services List".

In addition, ConvertTo-Html returns the HTML as a string. This output can be saved to a file using Out-File or other cmdlets. This makes it easy to create static web pages or integrate HTML content into other applications.

For instance, **Get-Process | ConvertTo-Html | Out-File "processes.html"** saves the process list as an HTML file named

“processes.html”. This can then be opened in any web browser to view the formatted information.

74) Save-Help: Downloads the newest help files from the Internet and saves them on a file share

The **Save-Help** cmdlet is used to download the latest help files for PowerShell modules and save them to a specified directory.

To use the **Save-Help** cmdlet, specify the destination folder where you want to save the help files. The following command saves help files to the folder C:\PowerShellHelp:

Save-Help -DestinationPath C:\PowerShellHelp

If you want to save the help files to a network location, specify the network path instead. For example, to save help files to a network share \\Server\HelpFiles, use the following command:

Save-Help -DestinationPath \\Server\HelpFiles

You can also update help files for specific modules by using the -Module parameter. For instance, to save help files for the Microsoft.PowerShell.Management module, you would use:

Save-Help -Module Microsoft.PowerShell.Management -DestinationPath \\Server\HelpFiles

The help files are saved in cabinet (.cab) format. When you want to update the help on a local computer, use the saved files with the -SourcePath parameter of the Update-Help cmdlet:

Update-Help -SourcePath \\Server\HelpFiles

75) Update-Help: Downloads and installs the newest help files

The Update-Help cmdlet in PowerShell is used to download and install the latest help files. This keeps the help documentation up-to-

date with the latest information. It targets specific PowerShell modules that may need updated help content.

To use Update-Help, launch PowerShell as an administrator.
Type Update-Help and press Enter. This command fetches the latest help files from the internet and installs them on your local machine. In some environments, internet access may be restricted. In such cases, help files can be downloaded using Save-Help and then distributed manually.

The cmdlet supports HTTP (Port 80) and HTTPS (Port 443) for downloading help files. This ensures secure and efficient updates. For specific modules, use the -Module parameter followed by the module names, like this: Update-Help -Module ModuleName.

76) Get-ComputerInfo: Get details about a computer system

The Get-ComputerInfo cmdlet in PowerShell provides comprehensive details about a computer's system and operating system properties.

To get all computer properties:

Get-ComputerInfo

This command retrieves all available system and operating system properties from the computer.

To get specific details like the BIOS version, use:

Get-ComputerInfo | Select-Object -Property BIOSVersion

The cmdlet provides hardware details such as:

- **Manufacturer**
- **Model**
- **Number of processors**
- **Network adapters**

Fetching OS Information:

It also retrieves operating system information including:

- **OS Manufacturer**
- **OS Installed Date**
- **OS Version**

77) Reset-ComputerMachinePassword: Reset machine account password

The Reset-ComputerMachinePassword cmdlet in PowerShell is used to reset the machine account password for the local computer. This can be necessary for various reasons, such as fixing trust issues between the computer and its domain.

Basic Syntax:

Reset-ComputerMachinePassword [-Server <string>] [-Credential <PSCredential>] [-Confirm] [-WhatIf]

Parameters:

- -Server: Specifies the name of a domain controller to use during the reset.
- -Credential: Defines the user account credentials that have permission to reset the password.
- -Confirm: Prompts for confirmation before executing.
- -WhatIf: Shows what would happen if the cmdlet runs without actual execution.

Examples:

1. **Resetting Local Computer Password:**Reset-ComputerMachinePassword This command resets the machine password for the local computer using default credentials.
2. **Using a Specific Domain Controller:**Reset-ComputerMachinePassword -Server "DC01" -Credential Domain01\Admin01 This command performs the reset

using DC01 as the domain controller and Admin01 for credentials.

3. **Reset on a Remote Computer:**Invoke-Command -ComputerName "Server01" -ScriptBlock { Reset-ComputerMachinePassword } This example uses Invoke-Command to reset the password on a remote computer named Server01.

78) Set-StrictMode: Enforce coding rules

The Set-StrictMode cmdlet in PowerShell is used to enforce coding rules in scripts. This cmdlet helps catch common mistakes and improve script reliability.

Versions and Settings

PowerShell supports different strict mode versions. The versions range from 1.0 to 3.0. The command syntax is:

Set-StrictMode -Version <version>

Here are the behaviors for each version:

- **Version 1.0:** Uninitialized variables cause errors.
- **Version 2.0:** Prohibits some deprecated practices.
- **Version 3.0:** Disallows writing to read-only properties and uninitialized variables.

Examples

Enable Strict Mode Version 1.0:
Set-StrictMode -Version 1.0

Enable Strict Mode Version 3.0:

Set-StrictMode -Version 3.0

Use in Function:

function TestStrict {

```
    Set-StrictMode -Version 2.0

    $a = "Hello World"

    Write-Output $a

}
```

79) Write-Progress: Display progress bar

The Write-Progress cmdlet in PowerShell is an effective way to display progress bars in the command window.

This cmdlet also allows customization of the progress bar and text displayed. Users can specify the activity, status message, and percentage complete.

Here's a basic example of how to use Write-Progress:

```
for ($i = 1; $i -le 100; $i++) {

    Write-Progress -Activity "Processing" -Status "$i% Complete" -PercentComplete $i

    Start-Sleep -Milliseconds 100

}
```

Nested Progress Bars

Users can also create nested progress bars to show multiple layers of information. This can be done by specifying unique ID values.

Example:

```
for ($i = 1; $i -le 100; $i++) {
```

```
   Write-Progress -Activity "Overall Progress" -Status "$i% Overall"
-PercentComplete $i -Id 1

   for ($j = 1; $j -le 10; $j++) {

      Write-Progress -ParentId 1 -Activity "Task $i" -Status "$j0%
Task Complete" -PercentComplete ($j * 10) -Id 2

      Start-Sleep -Milliseconds 50

   }

}
```

80) Test-Path: Check if path exists

The Test-Path cmdlet is a powerful tool in PowerShell used to check if a path exists. It can validate files, directories, registry keys, and other paths. This cmdlet returns True if the path exists and False if it does not.

To check if a file exists:

```
Test-Path -Path "C:\example\file.txt"
```

For checking a directory:

```
Test-Path -Path "C:\example\folder"
```

- **-Path**: Specifies the path to check.
- **-IsValid**: Checks if the path format is valid, not its existence.
- **-Exclude**: Excludes items matching a pattern.
- **-Include**: Includes only items matching a pattern.

Below are some practical use cases of the Test-Path cmdlet:

- Check if a file exists:Test-Path -Path "C:\Users\Public\Documents\report.docx"

- Verify a registry key:Test-Path -Path "HKLM:\Software\Microsoft\Windows\CurrentVersion"
- Validate a path format without checking existence:Test-Path -Path "C:\Invalid|Path" -IsValid

81) New-Event: Create custom events

The New-Event cmdlet in PowerShell allows users to create custom events. These events can notify users about various state changes in programs, such as hardware conditions, application status, and more.

New-Event -SourceIdentifier <String> [-EventArguments <Object[]>] [-MessageData <Object>] [-Sender <Object>] [<CommonParameters>]

Parameters

- -SourceIdentifier (Mandatory): A string that uniquely identifies the event.
- -EventArguments (Optional): An array of arguments associated with the event.
- -MessageData (Optional): Any additional data associated with the event.
- -Sender (Optional): The object that is sending the event.

To create a simple custom event, use:

New-Event -SourceIdentifier "MyCustomEvent"

You can create events to log specific actions or state changes. For instance, logging network status changes:

New-Event -SourceIdentifier "NetworkStatusChange" -MessageData "Network connected"

82) View Events: View added events

To view added events, use the Get-Event cmdlet:

Get-Event

The Get-Event cmdlet retrieves events that are in the event queue in PowerShell. This cmdlet is essential for handling events triggered by various system activities and user interactions.

To retrieve all events currently in the queue, use:

Get-Event

Filter by SourceIdentifier:

Get-Event -SourceIdentifier "ProcessStarted"

Retrieve Specific Number of Events:

Get-Event -MaxEvents 5

Combined Filters:

Get-Event -SourceIdentifier "ProcessStarted" -MaxEvents 3

The Get-Event cmdlet pairs well with Register-EngineEvent. First, use Register-EngineEvent to specify the event you want to monitor, then use Get-Event to retrieve these events later.

For example:

Register-EngineEvent -SourceIdentifier "ProcessStarted" -Action { Write-Host "A process has started." }

$events = Get-Event -SourceIdentifier "ProcessStarted"

83) Remove-Event: Delete events

Remove-Event is a PowerShell cmdlet used to delete events from the event queue in the current session. It focuses specifically on the events presently in the queue.

Syntax:

Remove-Event -SourceIdentifier "ProcessStarted"

This example removes a specific event identified by “ProcessStarted”.

Removing Events by Identifier Remove-Event -SourceIdentifier "UserLogin" This command removes all events with the source identifier “UserLogin” from the queue.

Removing All Current Events Get-Event | Remove-Event This will remove all events currently in the event queue.

84) Wait-Event: Suspends the execution of script

The **Wait-Event** cmdlet suspends the execution of a script or function until a specific event occurs. Execution then resumes once the event is detected.

Wait for a Specific Event: **Wait-Event -SourceIdentifier "ProcessStarted"** This waits for an event with the source identifier “ProcessStarted”.

85) Set-Location: Change current working directory

The Set-Location cmdlet in PowerShell is used to change the current working directory.

To change the current directory to the root of the C: drive, use the following command:

Set-Location -Path C:

86) Get-Module: Get information about modules

Get-Module is a key cmdlet in PowerShell used to retrieve information about modules. It displays the modules currently imported or available on the system.

Get-Module

This command lists all modules imported in the current session. To view all available modules, use:

Get-Module -ListAvailable

This shows both locally installed and available but not yet imported modules. If you want to check a specific module, apply the -Name parameter:

Get-Module -Name <ModuleName>

87) Update-Module: Update installed modules to latest version

The Update-Module cmdlet in PowerShell updates installed modules to their latest versions from an online gallery.

To update a specific module by name, use:

Update-Module -Name SpeculationControl

To update all installed modules, you can run:

Update-Module -Name *

88) Save-Module: Save a module

The Save-Module cmdlet in PowerShell is designed to download and save a module alongside any dependencies from a registered repository to a specified path on the local computer.

Syntax:

Save-Module -Name <ModuleName> -Path <DestinationPath>

Example:

Save-Module -Name Azure -Path C:\Modules

In this example, the Azure module is downloaded and saved to the C:\Modules directory.

89) Find-Module: Find modules

Find-Module is a cmdlet in PowerShell used to find modules in a repository. It is part of the **PowerShellGet** module, which simplifies the process of discovering and installing modules from online repositories.

To find a module using its name, use the following command:

Find-Module -Name ModuleName

Search with Wildcards

The *Name* parameter can include wildcards to broaden the search:

Find-Module -Name *PowerShell*

This command searches for all modules containing the term “PowerShell”.

Find-Module can search for a module with a specific minimum version:

Find-Module -Name ModuleName -MinimumVersion 2.0.0

If the repository includes a newer version, the cmdlet returns the latest version.

Example 1: Find a specific module

Find-Module -Name Az

Looks for the Az module.

Example 2: Find multiple modules with a wildcard

Find-Module -Name *Azure*

Searches for all modules related to Azure.

Example 3: Find a module by minimum version

Find-Module -Name Az -MinimumVersion 3.0.0

Searches for the Az module with at least version 3.0.0.

Finding and Installing Modules

To find and install a module in one step, use:

Find-Module -Name Az | Install-Module

This locates the Az module and pipes it to Install-Module, installing it directly.

Use Find-Command to find commands and related modules. It helps ensure you install the correct module:

Find-Command -Name CommandName

90) Get-ScheduledTask: Get scheduled tasks

The Get-ScheduledTask cmdlet in PowerShell retrieves scheduled tasks from the local or a remote computer.

Basic Usage:

Get-ScheduledTask

This command fetches all scheduled tasks on the local computer.

Filter by Task Name:

To get a specific task, use the -TaskName parameter:

Get-ScheduledTask -TaskName "MyTask"

This command retrieves the scheduled task named “MyTask”.

91) Register-ScheduledTask: create new scheduled tasks

Register-ScheduledTask is a cmdlet in PowerShell used to create a new scheduled task. This tasks allows scripts or commands to run at specified times or events.

To use Register-ScheduledTask, you generally need to specify a trigger and an action.

Register-ScheduledTask -TaskName "MyTask" -Trigger $trigger -Action $action

Here’s a simple example that creates a scheduled task to run a script daily at 2 PM:

$trigger = New-ScheduledTaskTrigger -Daily -At 2PM

$action = New-ScheduledTaskAction -Execute "Powershell.exe" -Argument "C:\Scripts\MyScript.ps1"

Register-ScheduledTask -TaskName "DailyScript" -Trigger $trigger -Action $action.

92) Unregister-ScheduledTask: remove a scheduled task

The Unregister-ScheduledTask cmdlet in PowerShell helps remove scheduled tasks from the Windows Task Scheduler. When a task is no longer needed, this cmdlet can clean up resources.

To remove a scheduled task, execute the following command:

Unregister-ScheduledTask -TaskName "TaskName"

By default, Unregister-ScheduledTask asks for confirmation before deleting a task. To skip this prompt, add the -Confirm:$false parameter:

Unregister-ScheduledTask -TaskName "TaskName" -Confirm:$false

Example 1: Unregister a task named “HardwareInventory”:

Unregister-ScheduledTask -TaskName "HardwareInventory"

This command removes the “HardwareInventory” task from the root folder.

Example 2: Unregister without confirmation:

Unregister-ScheduledTask -TaskName "UserTask" -Confirm:$false

This command deletes the “UserTask” without asking for confirmation.

93) Invoke-WebRequest: Make an HTTPs request

Invoke-WebRequest is a powerful cmdlet in PowerShell used for making HTTP and HTTPS requests. It’s incredibly useful for web scraping, downloading files, and interacting with web APIs.

To perform a simple web request:

Invoke-WebRequest -Uri "https://powershellfaqs.com"

This sends a GET request to the specified URL and returns the content of the webpage.

You can use Invoke-WebRequest to download files:

Invoke-WebRequest -Uri "https://powershellfaqs.com/file.zip" -OutFile "C:\path\to\download\file.zip"

The -OutFile parameter specifies the path to save the downloaded file.

You can extract titles and URLs from a Google search result:

```
$response = Invoke-WebRequest -Uri "https://www.google.com/search?q=powershell"

$titles = $response.ParsedHtml.getElementsByTagName("h3")

$urls = $response.Links | Where-Object { $_.href -match "/url?q=" } | Select-Object href
```

Conclusion

I hope you enjoyed these PowerShell cmdlets. Do let me know your feedback.