PowerShell Commands Every Developer Should Know: 50+ Cmdlets for Getting Things Done, Monitoring Performance, Debugging

Many developers love PowerShell, and for good reason: it adds power, functionality, and flexibility to the Windows Command Prompt, where many of us spend a good deal of time. It does, however, come with a bit of a learning curve, but once you've mastered the essential commands you need to know, it's productivity on steroids.

PowerShell commands are known as cmdlets, and these cmdlets are the driving force behind its functional capabilities. From commands that improve the overall Windows experience to commands useful for development work, there are dozens of important commands developers should know. We've put together this list to serve as a handy reference guide for those who are just beginning to tap into the power of PowerShell as well as those who want to level-up their PowerShell experience, including:

In addition to cmdlets, there are dozens of parameters and methods that help you get the most out of your scripts. The WhatIf parameter is particularly useful for testing PowerShell scripts without actually running them (@pluralsight). There are typically several parameters and methods available for each command. The commands listed below are a good starting point for any developer, but to fully realize the benefits, you should master parameters and other methods as well.

Basic PowerShell Cmdlets

These basic PowerShell commands are helpful for getting information in various formats, configuring security, and basic reporting.

1. Get-Command

Get-Command is an easy-to-use reference cmdlet that brings up all the commands available for use in your current session.

Simply type in this command:

```
get-command
```

The output will look something like this (@MS_ITPro):

```
CommandType     Name                                               Defi
nition
-----------     ----                                               ----
------
Cmdlet          Add-Content                                        Add-
Content [-Path] <String[...
Cmdlet          Add-History                                        Add-
History [[-InputObject] ...
Cmdlet          Add-Member                                         Add-
Member [-MemberType]
```

2. Get-Help

The Get-Help command is essential for anyone using PowerShell, providing quick access to the information you need to run and work with all of the available commands.

If you wanted some examples, for instance, you'd enter the following (@jp_jofre):

```
Get-Help [[-Name] <String>] [-Path <String>] [-Categ
ory <String[]>] [-Component <String[]>]
[-Functionality <String[]>] [-Role <String[]>] [-Exa
mples] [<CommonParameters>]
```

Check out this tutorial for more info on how Get-Help works. (@varonis)

3. Set-ExecutionPolicy

Microsoft disables scripting by default to prevent malicious scripts from executing in the PowerShell environment. Developers want to be able to write and execute scripts, however, so the Set-ExecutionPolicy command enables you to control the level of security surrounding PowerShell scripts. You can set one of four security levels:

- Restricted: This is the default security level which blocks PowerShell scripts from running. In this security level, you can only enter commands interactively.
- All Signed: This security level allows scripts to run only if they are signed by a trustworthy publisher.
- Remote Signed: In this security level, any PowerShell scripts that were created locally are permitted to run. Scripts created remotely are permitted to run only if they've been signed by a reputable publisher.
- Unrestricted: As its name suggests, the unrestricted security level permits all scripts to run by removing all restrictions from the execution policy.

Similarly, if you're working in an unfamiliar environment, you can easily find out what the current execution policy is using this command:

```
Get-ExecutionPolicy
```

Check out this threadon SuperUser for more information on configuring the execution policy in PowerShell. (@StackExchange)

4. Get-Service

It's also helpful to know what services are installed on the system. You can easily access this information with the following command:

```
Get-Service
```

The output will look something like the following (@MS_ITPro):

```
Status Name DisplayName

------ ---- -----------

Running AdobeActiveFile... Adobe Active File Monitor

V4

Stopped Alerter Alerter

Running ALG Application Layer Gateway Service

Stopped AppMgmt Application Management

Running ASChannel Local Communication Channel
```

If you need to know if a specific service is installed, you can append the -Name switch and the name of the service, and Windows will show the state of the service. Additionally, you can leverage filtering capabilities to return a specific subset of currently installed services. The following example will result in an output of data from the Get-Service command that's been piped to the Where-Object cmdlet, which then filters out everything other than the services that have been stopped:

```
Get-Service | Where-Object {$_.status -eq "stopped"}
```

Check out this postfor some additional examples of Get-Service in action.

5. ConvertTo-HTML

If you need to extract data that you can use in a report or send to someone else, the ConvertTo-HTML is one simple way to do so. To use it, pipe the output from another command to the ConvertTo-HTML command and use the -Property switch to specify which output properties you want in the HTML file. You'll also need to provide a file name.

For example, the following code creates an HTML page that lists the PowerShell aliases in the current console:

```
PS C:\> get-alias | convertto-html > aliases.htm
PS C:\> invoke-item aliases.htm
```

Bonus: The

```
Export-CSV
```

cmdlet functions in much the same way, but exports data to a .CSV file rather than HTML. Use

```
Select-Object
```

to specify which properties you want to be included in the output.

Check out this article from 4sysops for more information on using ConvertTo-HTML effectively (@adbertram).

6. Get-EventLog

You can actually use PowerShell to parse your machine's event logs using the Get-EventLog cmdlet. There are several parameters available. Use the -Log switch followed by the name of the log file to view a specific log. You'd use the following command, for example, to view the Application log:

```
Get-EventLog -Log "Application"
```

Check out a few more examples of Get-EventLog in action in this post(@nextofwindows).
Other common parametersinclude (@SS64):

- -Verbose
- -Debug
- -ErrorAction
- -ErrorVariable
- -WarningAction
- -WarningVariable
- -OutBuffer
- -OutVariable

7. Get-Process

Much like getting a list of available services, it’s often useful to be able to get a quick list of all the currently running processes. The Get-Process command puts this information at your fingertips.

Bonus: Use Stop-Process to stop processes that are frozen or is no longer responding. If you’re not sure what process is holding you up, use

Get-Process to quickly identify the problematic process. Once you have the name or process ID, use Stop-Process to terminate it.

Here's an example. Run this command to terminate all currently running instances of Notepad (@MS_ITPro):

```
Stop-Process -processname notepad
```

You can use wildcard characters, too, such as the following example which terminates all instances of Notepad as well as any other processes beginning with note:

```
Stop-Process -processname note*
```

Check out this post for more information on killing processes with PowerShell (@howtogeek):

8. Clear-History

What if you want to clear the entries from your command history? Easy – use the Clear-History cmdlet. You can also use it to delete only specific commands. For example, the following command would delete commands that include “help” or end in “command” (@MS_ITPro):

```
PS C:\> Clear-History -Command *help*, *command
```

If you want to add entries to a session, use:

```
Add-History
```

Check out this post for some useful information on clearing the history and pre-loading the history with a list of certain commands (@MS_ITPro).

9. Where-Object

Where-Object is one of the most important cmdlets to know, as it enables you to take a dataset and pass it further down your pipeline for filtering (@jonathanmedd):

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

Status Name DisplayName
------ ---- -----------
Running AdobeARMservice Adobe Acrobat Update Service
Running AppHostSvc Application Host Helper Service
Running Appinfo Application Information
Running AudioEndpointBu... Windows Audio Endpoint Bu
ilder
Running Audiosrv Windows Audio
Running BFE Base Filtering Engine
Running BITS Background Intelligent Transfer Ser...
Running BrokerInfrastru... Background Tasks Infrastr
ucture Ser...
Running Browser Computer Browser
Running CDPSvc Connected Devices Platform Service
```

10. Set-AuthenticodeSignature

If you want to keep your work secure in production and prevent modification, use Set-AuthenticodeSignature to add an Authenticode signature to a script or file.

```
> Set-AuthenticodeSignature somescript.ps1 @(Get-ChildItem cert:\CurrentUser\My -codesigning)[0] -IncludeChain "All" -TimestampServer "http://timestamp.verisign.com/scripts/timstamp.dll"
```

PowerShell Commands for Getting Things Done

When it comes to productivity, PowerShell can help you get things done with the following commands.

11. ForEach-Object

The ForEach-Object cmdlet performs an operation against every item in a specified group of input objects. While many cmdlets work with every object in a collection anyway, you'll need ForEach-Object for those situations in which you want to make other modifications or apply specific formatting to all objects in a collection.

Here's an example (@MS_ITPro). If you want to display a list of process names and want those names to render in cyan, you might try the following:

```
Get-Process | Write-Host $_.name -foregroundcolor cy
an
```

But the above will produce the following error:

```
At line:1 char:25
+ get-process | write-host <<<< $_.name -foregroundc
olor cyan Write-Host : The input object
cannot be bound to any parameters for the command ei
ther because the command does not
take pipeline input or the input and its properties
do not match any of the parameters
that take pipeline input.
```

Because the Write-Host cmdlet doesn’t understand what you want to do with the data that’s sent over the pipeline.

So, using the ForEach-Object cmdlet solves this problem:

```
Get-Process | ForEach-Object {Write-Host $_.name -fo
regroundcolor cyan}
```

Check out this tutorial for more information on ForEach-Object and working with loops (@tomsitpro).

12. Clear-Content

If you want to delete the contents of an item but retain the item itself, you'll use the Clear-Content cmdlet:

```
Clear-Content C:\Temp\TestFile.txt
```

You can also use this command to clear the contents of all files with a specified file extension. The following code would clear the contents of all files with the .txt extension, for instance:

```
Clear-Content -path * -filter *.TXT –force
```

You can also use wildcard characters. Plus, you can clear the contents of any type of file, from .txt files to .doc, .xls, and more.

Check out this post for more details.

13. Checkpoint-Computer

If you’re making major changes or running a risky experiment, you can set a restore point on your machine with the Checkpoint-Computer cmdlet.

Note that you can only create a restore point using this cmdlet once every 24 hours. If you run the command again, it will keep the previous restore point:

```
PS C:\> Checkpoint-Computer -Description "My 2nd che
ckpoint" -RestorePointType "Modify_Settings"
PS C:\> Get-ComputerRestorePoint | format-list
__GENUS : 2
__CLASS : SystemRestore
__SUPERCLASS :
__DYNASTY : SystemRestore
__RELPATH : SystemRestore.SequenceNumber=59
__PROPERTY_COUNT : 5
__DERIVATION : {}
__SERVER : CLIENT2
__NAMESPACE : root\default
__PATH : \\CLIENT2\root\default:SystemRestore.Sequen
ceNumber=59
CreationTime : 20120202180537.316029-000
Description : My 2nd checkpoint
EventType : 100
RestorePointType : 12
SequenceNumber : 59
```

Check out this article from MCP Mag for more (@MCPmag).

14. Compare-Object

It’s often useful to be able to compare two objects directly. You can do this using Compare-Object, which generates a report on the differences between two sets such as (@Marcam923):

```
PS G:\lee\tools> cd c:\temp
PS C:\temp> $set1 = "A","B","C"
PS C:\temp> $set2 = "C","D","E"
PS C:\temp> Compare-Object $set1 $set2

InputObject SideIndicator
----------- -------------
D =>
E =>
A <=
B <=
```

15. ConvertFrom-StringData

Use ConvertFrom-StringData to convert a string containing one or more value pairs to a hash table. Here’s an example of what the command looks like:

```
$settings = $TextData | ConvertFrom-StringData
```

This command is useful in a variety of situations, such as when you want to save the settings for a PowerShell script to enable others to edit the settings without working in the script code directly.

16. ConvertTo-SecureString

Convert an encrypted standard string to a secure string or plain text to a secure string using ConvertTo-SecureString. This cmdlet is used in conjunction with ConvertFrom-SecureString and Read-Host (@AdmArsenal):

```
ConvertTo-SecureString [-String] SomeString
ConvertTo-SecureString [-String] SomeString [-Secure
Key SecureString] ConvertTo-SecureString [-String] S
omeString [-Key Byte[]] ConvertTo-SecureString [-Str
ing] SomeString [-AsPlainText] [-Force]
```

17. ConvertTo-XML

Use the ConvertTo-XML cmdlet to create an XML-based representation of an object. This is also called serialization, and it’s a useful process for saving data for later re-use. Note that it’s important that your expression writes objects to the pipeline. Anything using Write-Host won't write to the pipeline and therefore can't be serialized. Here's an example of ConvertTo-XML in action (@PetriFeed):

```
Get-Service wuauserv -ComputerName chi-dc04,chi-p50,
chi-core01 |
Export-Clixml -Path c:\work\wu.xml
```

The specific cmdlet used in the above example, Export-Clixml, is suitable for most purposes. It converts the output of a PowerShell expression to XML and saves it to a file.

18. New-AppLockerPolicy

New-AppLockerPolicy creates a new AppLocker policy from a list of file information and other rule creation options. In total, there are five cmdlets that enable you to interact with AppLocker, including (@RootUsers_):

- **Get-AppLockerFileInformation:** Gets the required information for creating AppLocker rules from a list of files or the event log.

- **Get-AppLockerPolicy:** Used to retrieve a local, effective, or a domain AppLocker policy.
- **New-AppLockerPolicy:** As mentioned, this cmdlet is used for creating new AppLocker policies.
- **Set-AppLockerPolicy:** Sets the AppLocker policy for a specified group policy object.
- **Test-AppLockerPolicy:** Used to determine if a user or group of users will be able to perform certain actions based on the policy.

19. New-ItemProperty

New-ItemProperty creates a new property for an item and sets its value. You can use it to create and change registry values and data (properties of a registry key), for instance.

Check out this tutorialfrom mnaoumov.NET for some useful workarounds using this cmdlet (@mnaoumov).

20. New-Object

To create an instance of a Microsoft .NET Framework or Component Object Model (COM) object, use the New-Object cmdlet.

Here's an example that creates a new object using New-Object, stores it in a variable, then pipes it to Add-Member, which will then add properties or methods specified in the object created (@gngrninja):

```
$ourObject = New-Object -TypeName psobject

$ourObject | Add-Member -MemberType NoteProperty -Name ComputerName -Value $computerInfo.Name
$ourObject | Add-Member -MemberType NoteProperty -Name OS -Value $osInfo.Caption
$ourObject | Add-Member -MemberType NoteProperty -Name 'OS Version' -Value $("$($osInfo.Version) Build $($osInfo.BuildNumber)")
$ourObject | Add-Member -MemberType NoteProperty -Name Domain -Value $computerInfo.Domain
$ourObject | Add-Member -MemberType NoteProperty -Name Workgroup -Value $computerInfo.Workgroup
$ourObject | Add-Member -MemberType NoteProperty -Name DomainJoined -Value $computerInfo.Workgroup
$ourObject | Add-Member -MemberType NoteProperty -Name Disks -Value $diskInfo
$ourObject | Add-Member -MemberType NoteProperty -Name AdminPasswordStatus -Value $adminPasswordStatus
$ourObject | Add-Member -MemberType NoteProperty -Name ThermalState -Value $thermalState
```

21. New-WebServiceProxy

New-WebServiceProxy creates a web service proxy object that enables you to use and manage the web service from within PowerShell. This cmdlet is a beautiful thing for developers – it makes it unnecessary to write a lot of complex code to try to accomplish something in PowerShell when you can simply call another service that already makes it possible.

Here's an example:

```
$url = http://<webapp>.azurewebsites.net/CreateSite.asmx

$proxy = New-WebServiceProxy $url

$spAccount = "<username>"

$spPassword = Read-Host -Prompt "Enter password" –AsSecureString

$projectGuid = ""

$createOneNote = $false
```

Now, you can run the following to view a list of all available methods:

```
$proxy | gm -memberType Method
```

22. New-WSManInstance

Similarly to New-WebServiceProxy, New-WSManInstance creates a new instance of a management resource.

```
New-WSManInstance winrm/config/Listener
-SelectorSet @{Address="*";Transport="HTTPS"}
-ValueSet @{Hostname="Test01";CertificateThumbprint=
"01F7EB07A4531750D920CE6A588BF5"}
```

Check out this tutorial for a complete step-by-step example of how to get the information you need to execute this script successfully (@jonathanmedd).

23. New-WSManSessionOption

New-WSManSessionOption creates a new management session hash table that's used as input parameters to other WS-Management cmdlets including:

- Get-WSManInstance

- Set-WSManInstance
- Invoke-WSManAction
- Connect-WSMan

Here's the syntax (@SAPIENTech):

```
New-WSManSessionOption [-NoEncryption] [-OperationTimeout] [-ProxyAccessType] [-ProxyAuthentication] [-ProxyCredential] [-SkipCACheck] [-SkipCNCheck] [-SkipRevocationCheck] [-SPNPort] [-UseUTF16] [<CommonParameters>]
```

24. Select-Object

The Select-Object cmdlet selects the specified properties of a single object or group of objects. Additionally, it can select unique objects from an array or a specified number of objects from the beginning or end of an array.

```
PS > Get-Process | Sort-Object name -Descending | Select-Object -Index 0,1,2,3,4
```

This tutorial provides more information about the various ways you can use Select-Object (@infosectactico).

There are other cmdlets with similar functions including:

- Select-String: Finds text in strings or files.
- Select-XML: Finds text in an XML string or document.

25. Set-Alias

Set-Alias is a great command for enhancing productivity. It allows you to set an alias for a cmdlet or other command element in the current session (similar to a keyboard shortcut) so you can work faster.

The following example sets Notepad to np in the current session using Set-Alias (@powershellatoms):

```
New-Alias np c:\windows\system32\notepad.exe
```

Note that you can also customize your PowerShell profile with the aliases you use most often (@howtogeek).

26. Set-StrictMode

Set-StrictMode establishes and enforces coding rules in scripts, script blocks, and expressions. It's a useful command for enforcing code quality and preventing you from slacking off and writing sloppy code when it's 3:00 a.m. and you haven't had any sleep in two days (@adbertram).

To use it, there are two parameters to consider: -Off and -Version, and -Version has three possible values:

- Version 1.0: Prevents you from using variables that haven't been initialized (such as Option Explicit in VBScript)
- Version 2.0: Prevents you from using variables that have not been initialized and also prevents the calling of non-existent properties on objects, prevents you from calling a function like a method, and prohibits the creation of variables without a name.
- Version Latest: This option selects the latest StrictMode version available and uses it. This is a good option because it means that the latest StrictMode version is used regardless of the version of PowerShell you're using.

27. Wait-Job

Wait-Job suppresses the command prompt until background jobs running in the current session are complete. Wait-Job doesn’t show the output from jobs, however, but it can be used in conjunction with Receive-Job. Multithreading is possible in PowerShell thanks to -Jobs.

```
### Start-MultiThread.ps1 ###
$Computers = @("Computer1","Computer2","Computer3")

#Start all jobs
ForEach($Computer in $Computers){
Start-Job -FilePath c:ScriptGet-OperatingSystem.ps1
-ArgumentList $Computer
}

#Wait for all jobs
Get-Job | Wait-Job

#Get all job results
Get-Job | Receive-Job | Out-GridView
```

```
### Start-MultiThread.ps1 ###
$Computers = @("Computer1","Computer2","Computer3")

#Start all jobs
ForEach($Computer in $Computers){
Start-Job -FilePath c:ScriptGet-OperatingSystem.ps1
-ArgumentList $Computer
}

#Wait for all jobs
Get-Job | Wait-Job

#Get all job results
Get-Job | Receive-Job | Out-GridView
```

28. Write-Progress

Who doesn’t love a status bar? Monitor your progress using Write-Progress, which displays a progress bar within a Windows PowerShell command window.

Here’s an example that gives you a full progress bar and runtime strings (@credera):

```
$TotalSteps = 4
$Step = 1
$StepText = "Setting Initial Variables"
$StatusText = '"Step $($Step.ToString().PadLeft($TotalSteps.Count.ToString().Length)) of $TotalSteps | $StepText"'
$StatusBlock = [ScriptBlock]::Create($StatusText)
$Task = "Creating Progress Bar Script Block for Groups"
Write-Progress -Id $Id -Activity $Activity -Status (&amp; $StatusBlock) -CurrentOperation $Task -PercentComplete ($Step / $TotalSteps * 100)
```

Cmdlets for Performance Monitoring, Testing, and Debugging

There are also a variety of cmdlets useful for developers for troubleshooting, testing, debugging, and monitoring purposes. Here are a few you need to know.

29. Debug-Process

Developers love debugging! Well, we like it even more when there are no bugs to eliminate, but sadly that's not always the case. With PowerShell, you can debug a process using the Debug-Process cmdlet.

You can also debug jobs using Debug-Job (@MS_ITPro). And, you can set breakpoints or use the Wait-Debugger cmdlet:

```
PS C:\> $job = Start-Job -ScriptBlock { Set-PSBreakpoint C:\DebugDemos\MyJobDemo1.ps1 -Line 8; C:\DebugDemos\MyJobDemo1.ps1 }

PS C:\> $job

PS C:\> Debug-Job $job
```

30. Disable-PSBreakpoint

If you have at one time set breakpoints but want to eliminate them, do so easily using Disable-PSBreakpoint, which disables breakpoints in the current console. Here's the syntax (@ActiveXperts):

```
Disable-PSBreakpoint [-Breakpoint] [-PassThru] [-Con
firm] [-WhatIf] []
Disable-PSBreakpoint [-Id] [-PassThru] [-Confirm] [-
WhatIf] []
```

Alternatively, if you want to enable breakpoints in the current console, use Enable-PSBreakpoint.

31. Get-Counter

Get-Counter gets real-time performance counter data from the performance monitoring instrumentation in Windows OS. It's used to get performance data from local or remote computers at specific sample intervals that you specify.

In this example, you'll get a counter set with a sample interval for a specified maximum sample (@MS_ITPro):

```
PS C:\> Get-Counter -Counter "\Processor(_Total)\% P
rocessor Time" -SampleInterval 2 -MaxSamples 3
```

In the example below, this command gets specific counter data from multiple computers:

```
The first command saves the **Disk Reads/sec** count
er path in the $DiskReads variable.
PS C:\> $DiskReads = "\LogicalDisk(C:)\Disk Reads/se
c"

The second command uses a pipeline operator (|) to s
end the counter path in the $DiskReads variable to t
he **Get-Counter** cmdlet. The command uses the **Ma
xSamples** parameter to limit the output to 10 sampl
es.
PS C:\> $DiskReads | Get-Counter -Computer Server01,
Server02 -MaxSamples 10
```

32. Export-Counter

Export-Counter
exports PerformanceCounterSampleSet objects as counter log files. Two properties are available:

- CounterSamples: Gets and sets the data for the counters.
- Timestamp: Gets and sets the date and time when the sample data was collected.

And several methods, all of which are inherited from Object:

- Equals(Object)
- Finalize()
- GetHashCode()
- GetType()
- MemberwiseClone()
- ToString()

For example, the following commanduses Get-Counter to collect Processor Time data and exports it to a .blg file using Export-Counter (@TechGenix):

```
Get-Counter "\Processor(*)\% Processor Time" | Expor
t-Counter -Path C:\Temp\PerfData.blg
```

33. Test-Path

Test-Path lets you verify whether items exist in a specified path. For instance, if you're planning to use another command on a specified file, you may need to verify that the file exists to avoid throwing an error.

```
Test-Path C:\Scripts\Archive
```

If the folder exists, it will return True; if it doesn't, it will return False.

It can also work with the paths used by other PowerShell providers. For instance, if you need to know if your computer has an environment variable called *username*, you could use the following:

```
Test-Path Env:\username
```

Test-Path works with variables, certificates, aliases, and functions. Check out this post from TechNet for more details (@MS_ITPro).

34. Get-WinEvent

Look at Windows event logs using Get-WinEvent. For a list of available logs, use:

```
Get-WinEvent -ListLog *
```

Then, to review the details of a specific log, replace * with the name (pipe the output to format-list to view all the details):

```
Get-WinEvent -ListLog $logname | fl *
```

You can also view all the events in a log by using:

```
Get-WinEvent -LogName System
```

Check out this tutorial for more details (@rakheshster).

35. Invoke-TroubleshootingPack

Troubleshooting packs are collections of PowerShell scripts and assemblies that help you troubleshoot, diagnose, and repair common system problems (@ITNinjaSite). Find troubleshooting packs at:

```
C:\Windows\Diagnostics\System
```

You can run this script to get a list of all the troubleshooting packs available on the current system (@TechGenix):

```
Get-ChildItem C:\Windows\Diagnostic\System
```

Then, from an elevated PowerShell window, run a troubleshooting pack using this command:

```
Invoke-TroubleshootingPack (Get-TroubleshootingPack
 C:\Windows\diagnostics\system\networking)
```

36. Measure-Command

If you want to time operations in PowerShell, Measure-Command is a must-know cmdlet. It measures how long a script or scriptblock to run. Here's an example (@ToddKlindt):

```
Measure-Command { Mount-SPContentDatabase –Name wss_content_portal –WebApplication http://portal.contoso.com }
```

The output is a TimeSpan object, so it contains properties such as Hour, Minute, Second, etc., and it's easy to tailor the output to your preferences.

37. Measure-Object

You might also want to know how large a given object is. Use Measure-Object to calculate the numeric properties of any object, including characters, words, and lines in a string object, such as files of text.

Just specify the name and the type of measurement to perform, along with parameters such as (@WindowsITPro):

- -Sum: adds values
- -Average: calculates the average value
- -Minimum: finds the minimum value
- -Maximum: finds the maximum value

The following command sums the VirtualMemorySize property values for all process objects:

```
Get-Process | measure VirtualMemorySize -Sum
```

38. New-Event

New-Event is used to create a new event. A related cmdlet is New-EventLog, which creates a new event log as well as a new event source on a local or remote computer. If you have an automation engine supported by PowerShell, it’s a good practice to set up an event log (by creating a custom Event Log Type) that logs all messages sent by PowerShell. This is one example where you can implement Custom Logging in Event Viewer.

Start by creating a new Event Log LogName (@BundaloVladimir):

```
New-EventLog -LogName Troubleshooting_Log -Source Fa
lloutApp
```

Then, to send messages to your new event log, run the following using the Write-Log cmdlet:

```
Write-EventLog -log Troubleshooting_Log -source Fall
outApp -EntryType Information -eventID 10 -Message
"FalloutApp has been successfully installed"
```

39. Receive-Job

If you need to get the results of Windows PowerShell background jobs in the current session, use Receive-Job. This is usually used after using Start-Job to begin a job when you need to view the specific results.

```
Receive-Job -Name HighMemProcess
```

Check out this helpful tutorial on using Receive-Job and what to do if there are seemingly no results (@proxb).

40. Register-EngineEvent

This cmdlet is used to subscribe to the events generated by the Windows PowerShell engine and the New-Event cmdlet. For example, the following command subscribes to an event when the current PowerShell session exits and saves information (such as date and time) to a log file (@jonathanmedd):

```
Register-EngineEvent PowerShell.Exiting
-Action {"PowerShell exited at " + (Get-Date) | Out-
File c:\log.txt -Append}
```

41. Register-ObjectEvent

Register-ObjectEvent is similar to Register-EngineEvent, but rather than subscribe to events generated by the PowerShell engine and New-Event, it subscribes to the events generated by a Microsoft .NET Framework Object.

Here's an example (@NetworkWorld):

```
Register-ObjectEvent -InputObject $MyObject -EventNa
me OnTransferProgress -SourceIdentifier Scp.OnTransf
erProgress `

-Action {$Global:MCDPtotalBytes = $args[3]; $Global:
MCDPtransferredBytes = $args[2]}

Register-ObjectEvent -InputObject $MyObject -EventNa
me OnTransferEnd `

-SourceIdentifier Scp.OnTransferEnd -Action {$Global
:MCDPGetDone = $True}
```

There are other Register- cmdlets that you may find useful including:

- Register-PSSessionConfiguration: Creates and registers a new session configuration.
- Register-WmiEvent: This cmdlet subscribes to a WMI event.

42. Remove-Event

When you want to remove an event, use the Remove-Event cmdlet. If you need to remove an entire event log, you'd use Remove-EventLog, which deletes an event log or unregisters an event source.

Alternatively, Unregister-Event cancels an event subscription but does not delete an event from the event queue (@SS64).

43. Set-PSDebug

This cmdlet turns script debugging features on and off. It also sets the trace level and toggles StrictMode.

By using Set-PSDebug at the top of your script file just after the param() statement (if any), you can prevent errors with scripts that PowerShell doesn’t provide adequate information about for troubleshooting purposes. Here’s an example (@r_keith_hill):

```
Set-PSDebug -Strict

$Suceeded = test-path C:\ProjectX\Src\BuiltComponent
s\Release\app.exe

if ($Succeded) {
"yeah"
}
else {
"doh"
}

PS C:\Temp> .\foo.ps1
The variable $Succeded cannot be retrieved because i
t has not been set yet.
At C:\Temp\foo.ps1:6 char:14
+ if ($Succeded) <<<< {
```

44. Start-Sleep

If you need to suspend the activity in a script or session, use Start-Sleep, which halts the activity for a specified time period.

```
Start-Sleep -Seconds xxx
Start-Sleep -Milliseconds xxx
```

If you need to pause one or more running services, you'd use Suspend-Service.

45. Tee-Object

If you're analyzing performance or code quality, it's useful to be able to view the output of a command. Tee-Object stores the command output in a file or variable and also displays it in the console if it's the last variable in the pipeline. If it's not the last variable in the pipeline, Tee-Object sends it down the pipeline.

Here's the syntax:

```
Tee-Object [-FilePath] <string> [-InputObject <psobject>] [<CommonParameters>]

Tee-Object -Variable <string> [-InputObject <psobject>] [<CommonParameters>]
```

46. Test-AppLockerPolicy

Test-AppLockerPolicy evaluates whether input files are permitted to run for a specific user based on the specified AppLocker policy.

```
Test-AppLockerPolicy [-PolicyObject] -Path [-User ]
[-Filter >] []

Test-AppLockerPolicy [-XMLPolicy] -Path [-User ] [-F
ilter ] [ arameters>]
```

This tutorial offers more details about the available parameters and examples of Test-AppLockerPolicy in action (@powershellhelp).

47. Test-ComputerSecureChannel

This cmdlet tests and repairs the connection between a local computer and its domain. Without this command, the usual solution was previously to remove a computer from its domain and then reconnect it in order to reestablish the relationship. Test-ComputerSecureChannelmakes it possible to reestablish the connection in less time (@WindowsITPro).

When signed on as a local administrator, simply run the following:

```
Test-ComputerSecureChannel -credential WINDOWSITPRO
\Administrator -Repair
```

You can use Test-Connection to send Internet Control Message Protocol (ICMP) echo request packets (pings) to one or more computers.

48. Test-Path

Use Test-Path to determine whether all elements of a path exist. Essentially, it helps you handle errors before they occur. In its simplest form, it returns True or False (@MCPmag):

```
PS C:\> test-path c:\
True
PS C:\> test-path z:\foo
False
```

49. Trace-Command

Trace-Command configures and starts the trace of a specified command or expression. To use it, you’ll also need to use Get-TraceSource in order to look for particular names using wildcard characters:

```
PS&gt; Get-TraceSource -Name *param*
```

You can filter the output to match the description to the pattern you’re after. Once you’ve identified the possible trace name, you’ll use Trace-Command to get the answers you need. Here’s an example:

```
[CmdletBinding(DefaultParameterSetName = 'Host')]

param (
# ScriptBlock that will be traced.
[Parameter(
ValueFromPipeline = $true,
Mandatory = $true,
HelpMessage = 'Expression to be traced'
)]
[ScriptBlock]$Expression,

# Name of the Trace Source(s) to be traced.
[Parameter(
Mandatory = $true,
HelpMessage = 'Name of trace, see Get-TraceSource fo
r valid values'
)]
[ValidateScript({
Get-TraceSource -Name $_ -ErrorAction Stop
})]
[string[]]$Name,

# Option to leave only trace information
# without actual expression results.
[switch]$Quiet,
# Path to file. If specified - trace will be sent to
file instead of host.
[Parameter(ParameterSetName = 'File')]
[ValidateScript({
Test-Path $_ -IsValid
})]
[string]$FilePath
)

begin {
if ($FilePath) {
# assume we want to overwrite trace file
$PSBoundParameters.Force = $true
} else {
$PSBoundParameters.PSHost = $true
}
if ($Quiet) {
$Out = Get-Command Out-Null
$PSBoundParameters.Remove('Quiet') | Out-Null
} else {
$Out = Get-Command Out-Default
}
}

process {
Trace-Command @PSBoundParameters | &amp; $Out
```

```
}
}


PS&gt; New-Alias -Name tre -Value Trace-Expression
PS&gt; Export-ModuleMember -Function * -Alias *
```

Check out this post for more details on playing detective with Trace-Command (@PowerShellMag).

50. Write-Debug

Write-Debug writes a debug message to the console. When you write this in a function or script, it doesn't do anything by default; the messages essentially lay in wait until you either modify your $DebugPreference or activate the -debug switch when calling a function or script. When $DebugPreference is set to 'inquire' or the -debug switch is activated, the message creates a breakpoint, giving you an easy way to pop into debug mode.

Take this example (@RJasonMorgan):

```
function Get-FilewithDebug
{
[cmdletbinding()]
Param
(
[parameter(Mandatory)]
[string]$path
)
Write-Verbose "Starting script"
Write-Debug "`$path is: $path"
$return = Get-ChildItem -Path $path -Filter *.exe -R
ecurse -Force
Write-Debug "`$return has $($return.count) items"
$return
}
```

The example above produces the following when run with -debug:

```
[C:\git] > Get-FilewithDebug -path C:\Users\jmorg_00
0\ -Debug
DEBUG: $path is: C:\Users\jmorg_000\


Confirm
Continue with this operation?
[Y] Yes [A] Yes to All [H] Halt Command [S] Suspend
[?] Help (default is "Y"):
```

What PowerShellcommands do you use most often in your development work? Share your go-to cmdlets with us in the comments below. To learn about other essential tools to add to your dev toolkit, download our Ultimate Dev Toolbox for insights into tools that are portable, budget-conscious, and fit for the task at hand – not to mention simple, free, and efficient.