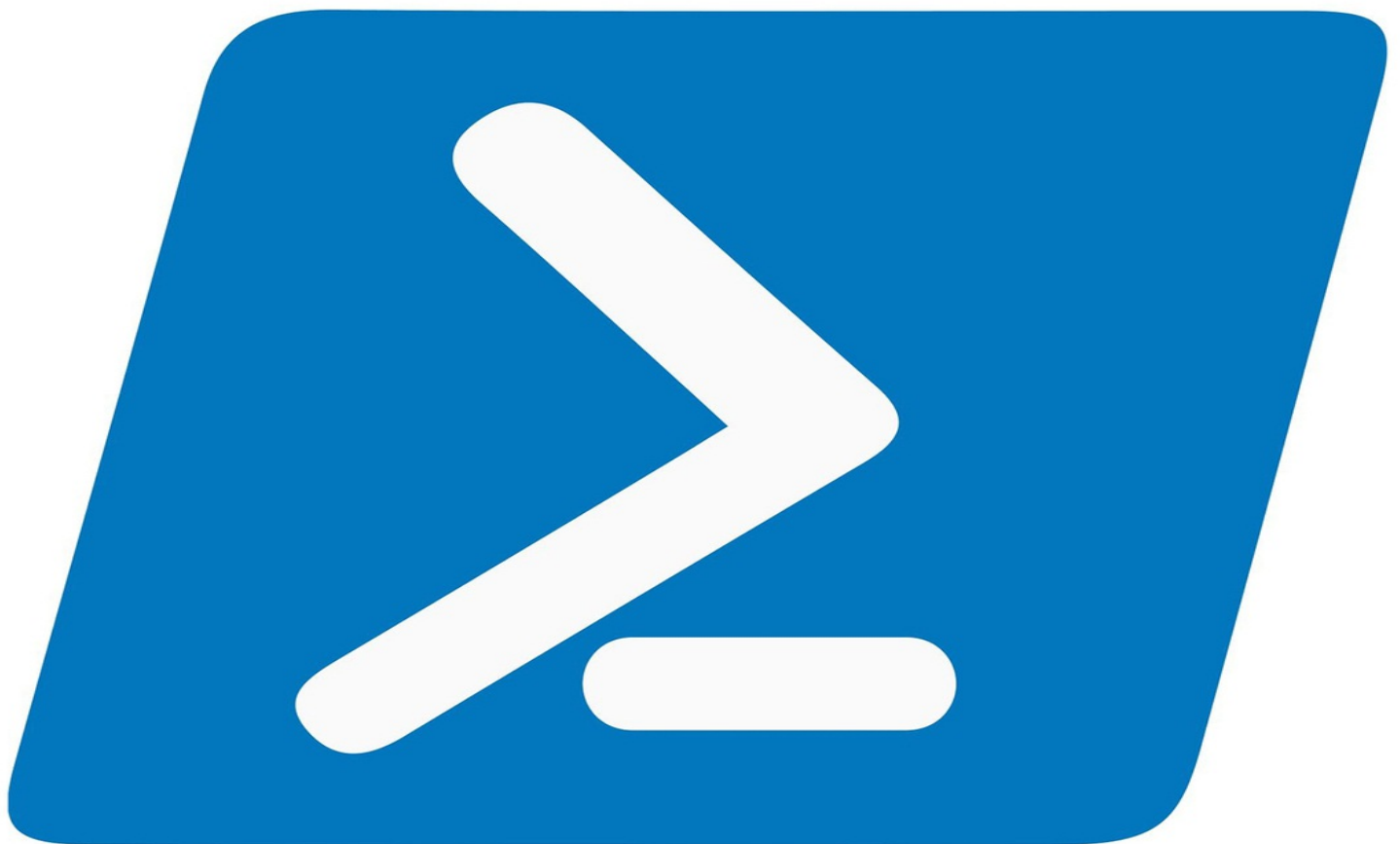


# PowerShell

## Step-by-Step

SCRIPTS TO AUTOMATE WINDOWS PROCESSES,  
STREAMLINE PRODUCTION, OVERHAUL  
INTERFACE WITH AZURE AND OFFICE 365



**Brandon Shaw**



**© Copyright 2019 by Brandon Shaw**  
**All rights reserved.**

**T**his document is geared towards providing exact and reliable information with regard to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render accounting, officially permitted, or otherwise, qualified services. If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

Respective authors own all copyrights not held by the publisher.

The information herein is offered for informational purposes solely and is universal as so. The presentation of the information is without a contract or any type of guarantee assurance. The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are owned by the owners themselves, not affiliated with this document.

# Table of Contents

[Steps and key information of PowerShell](#)

[Process and feature in PowerShell](#)

[A fundamental information of PowerShell](#)

[Functions, switches, and looping structures of PowerShell](#)

[How to utilize .NET](#)

# Steps and key information of PowerShell

Microsoft PowerShell is an effective administrative tool that can help you automate duties for your PC and network in various cases. PowerShell carries the factors of the command prompt and is built on the .NET Framework. It has long been a priority tool for IT directors to control large-scale networks.

Learning to use PowerShell will allow you to simplify many tedious, every-day tasks. You will additionally be able to make system-wide adjustments for the duration of your network, so you do not need to make individual adjustments on every server. Of late, it has emerged as an essential part of running a hybrid cloud environment.

PowerShell is a critical time and has lots of different uses that make you more productive and jogging your work efficiently. Among the basic things you can do with it, you can time table every day updates on the system, review contemporary processes, bicycle services, and many different things. It is authentic that many of these functions can be achieved through the GUI, although the PowerShell factor is to do them faster.

If you have the challenge to chase that takes several minutes of keying and setup, you can write the same attribute in a single command that your title to PowerShell. So next time, you just drag that script down, below the identity you saved it and it will run in the background. Considering PowerShell's scripting logic, how this object and variable engine works, and neatly deploying it to your community, you feel that you've been prohibiting its use for a long time.

This information will tell you the basics of PowerShell, which is convenient for entry-level IT experts to learn, especially if you are already familiar with the window's command prompt. We will use fundamental tools and commands, explaining how to manipulate archives and folders, understand objects, use variables, and manage remote servers.

## **Introduction to tools, commands, and modules**

As with most cases in life, taking the time to study and fully understand the basics will go a long way towards reducing headaches and with a deeper dive into the world of PowerShell commands you will find more Will help in holding better principles. The three ideas brought to this section are necessary

to understand the key ideas that lay the foundation for PowerShell.

### **PowerShell tools**

PowerShell is installed by default in Windows 10, Windows 7, Windows Server 2008 R2 and later versions of Windows. Newer versions of PowerShell introduce new aspects and "cmdlets" (the term for Microsoft's PowerShell directives - "command-lets" are mentioned) and installed using a compatible version of the Windows Management Framework (WMF). Currently, WMF (?) 5.1 is a modern model promoted for use in production. In some cases, many new aspects are structured on a working machine other than the WMF version. For example, Windows Eight and Windows Server 2012 support the Test-Net Connection cmdlet, which allows you to check connectivity to a unique TCP / IP port, but this cmdlet is now available in Windows 7 even when the state is running Is not. Art WMF version.

On most Windows structures users will have the availability of Powderly environments, PowerShell Console and PowerShell ISE (Integrated Scripting Environment). The PowerShell console feels like a typical command line, although there is a whole load of PowerShell behind it. Variable names, loops, tab completion, and piping all work from the PowerShell console. To help you build and test your PowerShell code, PowerShell ISE offers more in-depth use (such as script creation) for PowerShell ISE tab completion, code highlighting, and Microsoft's Intelligence code full functionality. PowerShell ISE additionally allows working with multiple PowerShell scripts using tabbed navigation.

### **PowerShell cmdlets**

The groundwork of PowerShell commands is cmdlets. Microsoft designed some graph techniques when designing PowerShell cmdlets. The first is the ability to easily locate cmdlet names, or at the very least make them convenient to find. PowerShell commands or cmdlets are also designed to be simple to use with standardized syntax, making them easy to use interactively from the command line or to create powerful scripts.

PowerShell cmdlets use verb-noun structures in get-service, stop-service or import-CSV. The verb cmdlet identifies the action to be performed on the component noun. Typically, cmdlets used to record use the get verb, as with get-process or get-content. Commands used to modify some usually begin with an action set, while some things, including a new unit, start regularly

with add or new. In many cases, these verb-noun combos can be inferred or inferred because of the extensive naming convention.

Standardized cmdlet naming is not always the only factor in PowerShell that is designed to improve command line usability. Parameters commonly used during PowerShell additionally use trendy names. An example of this is the -ComputerName parameter, which allows a cmdlet to be completed against one or more remote computers. Similarly, credentials are used to provide a credential object, which contains the user's login credentials, to run the command as a specific user.

### **PowerShell module**

When using PowerShell by the console, aliases can be used for each cmdlet and parameters to preserve keystroke and reduce the average size of the command (a benefit that does not require the simultaneous piping command to be omitted ). Cmdlet aliases no longer always use a well-known naming convention, although they routinely denote common command-line utilities.

Alias DIR, CD, DEL and CLS in PowerShell correspond to the Get-ChildItem, Set-Location, Remove-Item, and Clear-Host cmdlets, respectively. Parameter aliases can work in ways: they can use predefined aliases described through the cmdlet, or they can be aliased using enough characters for a unique match between the cmdlet's supported parameters.

### **Manage files and folders**

Regardless of your niche in the IT industry, prospects are some of the steps in your daily piece that involve managing documents and folders in some way. Whether it is moving a folder to another area on a server, storing log files, or searching for large files; Almost every machine administrator spends their days managing files and folders. In cases where repetitive duties are being repeated on more than one file, or an equal set of duties are run repeatedly, automation via PowerShell can be a real time-saver.

**Searching Files and Folders** One of the first command-line tool directors learned in the old days of computers used to be Dir commands. For those new to the game, the dir will list the archives and folders contained within the fixed directory. PowerShell features a comparable command in the form of the Get-ChildItem cmdlet. Get-ChildItem allows you to quickly create a list of files in a list so that you can work on these files either through a piped

command or by way of assigning the output to a variable.

At its most basic, Get-ChildItem can be used by supplying only one path, through a pipeline, using the -path parameter, or just after the cmdlet name. To tune the returned response via Get-ChildItem, it is important to look at some of the parameters made accessible via the cmdlet.

The -filter parameter is a way by which you can search for files. By default, the Get-ChildItem cmdlet returns only the direct youth of the target directory. This functionality can be extended by using the switch, which searches for directories contained inside contemporary folders.

PowerShell provides the ability to limit the end result to both a file or a folder using the Get-ChildItem -File or -Directory switch in four. Earlier versions of PowerShell have to pipe the result to determine the where-and-where filter on the PSI retainer property. An example of both techniques being used to return folders contained in C: Users are proven here:

```
Get-ChildItem C: User -Directory | Get-ChildItem C: User | Where-Item {$_.PSIsContainer -eq $true}
```

The -Force switch needs to be used to detect hidden or system archives. In PowerShell Four and above ChildItem can only be used to hide these archives, which are hidden only, or using machine documents using the -Hidden, -ReadOnly and -System switches. Similar performance pre-variations can be done by filtering on the mode property wherever and where to use:

```
Get-ChildItem C: User | Where-Item {$_.Mode-like '* R *'}
```

### **Checking if a file exists**

Often when working with files, we all need to understand whether a file exists or a folder direction is valid. PowerShell introduces a cmdlet to perform this validation acting as a test-path, which returns either true or false values.

Test-path is often beneficial as a precautionary step that seeks to replicate or delete a specific file.

### **Copying, moving and deleting files**

As you would expect, PowerShell is fully successful in performing standard



file operations on multiple objects in a single pass. The copy-item cmdlet can be used to copy one or more documents or folders from one location, identified through the -path parameter, to the unique location via the -Destination option.

-WhatIf change allows you to see what will appear if you run a script or command, leaving only the manageable negative consequences of deleting the required script data. It is additionally really worth noting that -WhatIf is not always restricted to file operations, it is used extensively during PowerShell.

For scripts that you intend to run or malfunction manually, run a subordinate manually, consider using -confirm. This allows you to conscientiously require consumer interaction before the operation. It is definitely better to be prepared to let the completeness go (file backup complete, replication disabled, etc.) before the huge file operation is started.

### **Objects, piping, filtering and more**

The key to PowerShell, in addition to standardized naming and various features that make it intuitive, is that many cmdlets are object-based. By working with objects, PowerShell allows you to without difficulty modify multiple gadgets along the same line of code, modify a specific subset of objects inside thousands, or use these objects to gather statistics Or to process other related items. PowerShell Object

Objects, which are no longer familiar with the terminology, refer to objects that contain certain characteristics or properties; Such as strings of characters, a list of information and numeric values. A suitable example of an object is a Windows process, reclaimed use of the gate-process cmdlet, which has many properties indicating the executable name, priority, CPU usage, and dispatch usage.

The gate-member cmdlet can be used to search for objects and their member houses and methods. Not only will the gate-member show you the houses of an object and the type of information they type, but however it will also provide you with the object type, which in turn can be used to detect various cmdlets, which Can take delivery of an object type.

### **line pipe**

PowerShell allows you to take advantage of cmdlets and objects, which are

obtained through a technology identified as piping. Using the Pipe Persona (!), You can quickly and spontaneously select items and then act on them. A perfect example of piping is hitting the exact strategy of using gate-process java stop-process. Similarly, you can resume the use of a line such as a gate-service spooler. Restart service.

Often cmdlets with the same noun will be used during piping, although the technique is not restricted to cmdlets with the same noun. By using the object type returned using the get-member, you can search for other cmdlets that can be used to derive the pipe command. Using the -Parameter Type the get-command cmdlet with the object type will return a list of cmdlets that can take delivery of the specified object type.

### **Filtering**

PowerShell has a total list of cmdlets, which are used to do the heavy lifting with objects, specifically with an object noun. Many of these cmdlets are among the most commonly used cmdlets, while others are used for more specialized tasks.

Wherever the object cmdlet lets you limit or filter objects that exceed the pipeline. For example, the command `get-service | Where-Item {$_.Dependency-service-$null}` will return a list of offerings that have dependencies. Wherever the syntax used with the object is really noticeable and applies to some other object cmdlets. Squiggly brackets are used to delete a code block in PowerShell, and in this case, indicate the position to be applied to the object in the pipeline. The automatic variable `$_` is used to indicate the contemporary opportunity for the object being evaluated. PowerShell comparison operators use hyphenated formatting, so `-eq` (equal) is used in our example to find an exact suite with the word "stalled".

For interactive use within PowerShell consoles, the use of aliases can retail time and effort. The where-object cmdlet uses a question mark (?). PowerShell 3 also allows you to simplify your what-object syntax by removing the want for automatic variables from script blocks and pipelines. In PowerShell three, this command is equivalent to the supply given above: `Get-Service | The Dependent Services - $null`

### **Acting on objects**

The ForEach-Object is used to conduct motion on every occasion of an

object. Syntactically, `ForEach-Object` is similar to `Where-Object`, in which each script block and automatic variable are used with both cmdlets. Where `ForEach-Object` Excel is being able to perform duties as opposed to every object instance that is too complex for simple piping. For example, you may need to list the file security for a file share, in which case you want to use the pipe `Get-ChildItem` cmdlet for the `ForEach-Object`, and then the full name parameter (`$ _`). You can use `Get-ACL` in conflict. `FullName`) to list file security for a list of files.

`Where-for-object` cmdlet as an object can be simplified the use of an alias is indicated with a percentage sign (%). In addition, PowerShell supports three syntaxes to provide even more intuitive use.

While filtering an object or performing motion on an object's instance are common tasks, it is a good idea to avoid every `where-object` and `for-object` when possible. Many cmdlets provide an alternative option or different parameters that can help restrict different types of results without appearing on each instance of an object, which usually results in a good size performance improvement.

Similarly, `ForEach-Object` performs an individual action on each instance of the piped object. When possible, objects should be piped to cmdlets without delay, which, in addition to enumerating each object inside the object, can take the necessary action on the entire object.

### **Comparison operator and conditional logic**

System directors make choices every day concerning the choices that renovations are made to perform on the server-based solely on several criteria. Automating repetitive administrative tasks with PowerShell often forces the use of common sense to repeat this decision-making process. Several methods can be used to compare preferred results, achieve the use of filters and conditional logic.

### **PowerShell comparison operators**

You will not find any distance in increasing PowerShell scripts other than conditional logic performance, which begins with the comparison of values. The ability to test whether a consumer is present, if a file is created, or if a laptop is able to connect to another, all require an evaluation of the value. What to do with the syntax of a huge in PowerShell: using common

evaluation operators like `<or>` PowerShell instead of `-lt` or `-gt` to compare.

Many evaluation operators are often used with numerical values, although they have an area when working with dates or version numbers and other variable types. The following table contains the comparison operators of most operators used to compare numbers.

When compared against text strings — one can be used when necessary. The `-match` operator can be used when a part of a string is searched for, or wildcard searches can be used to perform the `-like`. PowerShell can additionally be used to search for an exact value inside an array through the use of `-in`, `-not in`, `-contains`, or `-not contains`.

In instances where more than one circumstance must be met, parent statements can be used to control the corporations of the circumstances. The following example can be used to select a large number of web browser-related processes:

### **PowerShell Where**

Many common PowerShell cmdlets return long lists of values that are of little use as a whole. Using a `where-object` allows you to quickly limit the results to the conditions you underline inside a script block. The following example lists documents that have a collection bit set in the state of the art user's profile:

This example demonstrates the use of the `$_` default variable, which is used to indicate the current record from the pipeline. PowerShell 4 allows you to access objects using Alias such as `Where-Object`, and also accepts shortcut notation for the condition. This example is functionally equivalent to the above:

If through statement is one of the more common strategies of controlling the flow of your script and demonstrating conditional common sense. To meet standards first, and then action must be taken, IT professionals can automate complex administrative tasks. Like most programming languages, in

PowerShell, the if statement can be used in conjunction with the else statement, which enables you to handle more than one scenario.

A simple if the declaration is needed along with the position in parentheses. Processed when the status is evaluated as code contained within subsequent script blocks; If it is false it is omitted. The following example shows if the declaration to test for network connectivity is an easy one:

With the **Hurt If** the statement you can add additional stipends to one of the statements. It differs from a pair if the statements in it are used only in the first situation. Incorrect statements are made on the basis of a declaration declaring the motion to work if none of the previous statements are completed. An example of a more complex situation if, and other blocks is below:

### **Switch statement**

Like the if statement, the change approves you to perform a set of commands when certain criteria are met. The major difference between if and swap is that swap statements consider a single set of parameters against a pair of probabilities, alternatively evaluating a probably unrelated set of criteria compared to every declaration.

The evaluation of switch statements begins with the change key-word observed by the expression. A script block enclosed in curly brackets follows the evaluated expression and includes matches to be received. Each of these fits is followed through a script block, which defines the movements to be taken when the condition is met. Default key-words can be used to perform movements in opposition to unmatched norms. This example uses the Get-Date cmdlet to check the day of the week and returns whether it is the day of the week or the day of the week:

The preceding example can be simplified by using the -Wildcard option with the example below. Using wildcards with a swap statement gives you an additional environmentally friendly way to perform a conditional action.

### **Using variables, arrays, and hashtags**

Windows PowerShell is about making all IT professionals more efficient in their day to day tasks. PowerShell is sufficiently intuitive that entry-level or mid-level administrators can start studying the language, use cmdlets from the console, and even begin writing scripts that are reasonably easy.

Once your PowerShell script starts to get more complex, you need to start working with the elements of PowerShell that will take you back to the programming classes you took in college. Variables in PowerShell are essential for scripting, as they allow you to ignore information between components of your script.

### **Assigning and Referring PowerShell Variables**

You are probably familiar with the concept of variables at some stage, whether it is from prior experience in programming or mathematics. Values can be assigned to a variable during your PowerShell script to refer to that variable. The use of variables allows you to ensure that the cost remains constant for the duration of the script, making it less complicated to substitute the value later (either manually or programmatically), and usually, your scripts become more readable.

In PowerShell the variable name starts in \$, as in \$UserName, and the values are assigned using =, such as \$UserName = "John Smith" or \$UserAge = 40. PowerShell helps in a large range of variable types; Such as textual content strings, integers, decimals, arrays and even better types such as model numbers or IP addresses.

By default, PowerShell will try to figure out to use a variable type, although this can be implemented by indicating the type before the variable title as in [int32] \$UserAge = 40. In instances where the fixed type does not shape the type applied, the value will be changed if possible, or an error will occur.

Sometimes a requirement arises to dynamically assign or adjust a variable value, without doubt, an optional value. This should involve performing mathematical operations on a variable or adding text to a string. Numerical values can be assigned or modified using popular math handlers such as +, -, \*, and / in the following examples:

$\$SecondsPerDay = 24 * 60 * 60$   $\$weekPerYear = 365/7$

Numeric values can also be increased or decreased by the use of ++ - in

addition to adding or subtracting 1 from the value. These examples have the same effect, increasing the variable charge through 1:

```
$ myNumber = $ myNumber + 1 $ myNumber ++
```

PowerShell uses the addition operator (+) to add or combine textual content. Subsequent variations of PowerShell additionally fully assist in inserting a variable as a double-quoted string (as opposed to a single quote), even though this method was used carefully to maximize backward compatibility Go

```
$ Website = "www.business.com" $ myString = "my favorite internet site is"  
+ $ website $ myString = "my favorite website is $ website"
```

Variables can be used to hold data back from cmdlets. This is especially available if your script calls for the same set of frequently used statistics. Instead of running the same cmdlet more than once you can assign it to a variable and later use it as many times as you want in the script.

### **PowerShell arrays and hashtags**

Arrays are often used to create a list of values, such as a username or a list of cities. PowerShell provides a wide variety of arrays, each of which can be assigned and accessed in amazing ways. For the works of this book, we are going to be the most frequently used centers of attention. Basic PowerShell arrays can be defined by `$ nameArray = @ ("John", "Joe", "Mary")` as using a list of items in parentheses and preferring with `@symbol`. Items in an array can be accessed using their numeric index, setting with 0, such as inside square brackets: `$ nameArray [0]`.

A large superior form of the array, which is accepted as a hashtable, is assigned with the preceding squeeze bracket with the help of the `@` sign. While arrays are typically (but not always) used to contain the same data, hash tables are more suitable for associated (rather than identical) data. Individual items within the hashtable are named as `$ user = @ {FirstName = "John" as a numeric index assigned option; Last Name = "Smith"; MiddleInitial = "J"; Age = 40}`. Items inside the hashtag are easily used as variables and as key names in `$ user.LastName`.

### **Default variable**

One final component of being aware of variables in PowerShell is that the default variable contains a host, which automatically holds the specified

value. There are countless variables starting with the prefix "\$ env:" that can be obtained to get courses such as Windows directories, temporary folders, modern-day person titles or domains, and many other types of device information. A complete list of \$ env variables can be obtained using the following command:

Many different additional variables are on hands such as \$ Home and \$ PsHome, which provide you with a list of the user's home and route to the PowerShell home directory, respectively. The \$ host variable gives information about the current PowerShell environment to an object, whether it is PowerShell ISE or console. Finally, \$ PSVersionTable contains information about the version of PowerShell that is installed, including \$ PSVersionTable.PSVersion.Major, which suggests the original PowerShell model work on the host running the script.

For Boolean values of True and False, the PowerShell value is robotically assigned to the default variables \$ true and \$ false.

### **Remote server and session management**

As with many gadget administration tools, PowerShell presents great benefits for your enterprise in addition to being leveraged as opposed to multiple systems only. In most scenarios, it executes PowerShell cmdlets or scripts in a way that moves against the remote system.

Let's look at a few different pox PowerShell brings to the desk in the way of remote management. We will also evaluate some of the requirements desired to use this functionality and some hints on how to get the most from PowerShell far-flung management.

### **Remote Server Management with PowerShell**

The simplest approach to using PowerShell to conduct administrative tasks against remote servers or Windows computers, in general, is the use of the -computer name parameter. We explained how to detect cmdlets, which are given the -ComputerName parameter in our publication when using PowerShell Assistant features.

Many of the common PowerShell cmdlets you'll use every day (Get-Process, Get-Service, Get-EventLog, etc.). This way you are allowed to run



movements as opposed to a computer away. It is also well worth noting that - computer names will be given multiple hostnames, which allow you to target a list of computers in addition to executing the cmdlet more than once.

Management of Farway computers The use of this approach bypasses some requirements for frequent PowerShell Farway sessions.

### **Enable PowerShell out of session**

The massive feature set introduced in the PowerShell Model 2.0 was the remoting feature. Remoting PowerShell using WinRM (Windows Remote Management) allows you to create far-flung sessions for the computer. The performance compares to familiar Farway administration tools such as Telnet or SSH, although it uses industry-popular ports and protocols such as HTTP and SOAP.

Although most Windows systems on your neck must have at least PowerShell 2.0 hooks (this is the default on Windows 7 and Windows Server 2008), PowerShell remoting is not enabled by Windows Server 2012 by default. The WinRM service must be remodeled to enable PowerShell. Set to start and start automatically, and a firewall rule must be able to allow the verbal exchange to the server.

Fortunately, PowerShell remoting can be enabled in a single step, so that the Enable-PSRemoting cmdlet can be used with the -Force swap from the enabled PowerShell prompt. Because Windows Server 2012 and later PowerShell remoting is enabled by default, there is no need to run Enable-PSRemoting unless it is disabled for some reason.

### **Why use away sessions?**

There are some huge objectives that require remote classrooms to be used in some situations as compared to remote computers, the first demonstration. When you run a cmdlet such as gate-service towards ten specific computers, the nearby PCs have to execute the command and method the facts on each computer. When the remote duration is leveraged, each remote PC runs the command by spreading the workload over ten computers. Once you start talking about PowerShell scripts that run on stacks or on lots of computers, overall performance becomes one of your priority priorities.

PowerShell is a task automation and configuration management framework from Microsoft, which includes a command-line shell and scripting language.

Initially, only one Windows component, known as Windows PowerShell, was made open-source and cross-platform on 18 August 2016 with the introduction of PowerShell Core. The former is built on the .NET Framework while the latter on the .NET Core.

In PowerShell, administrative tasks are usually performed by cmdlets (explicit command-lets), which are special .NET classes that implement a particular operation. These work by accessing data in different data stores, such as a file system or registry, which are made available to PowerShell through providers. Third-party developers can add cmdlets and providers to PowerShell. Cmdlets can be used by scripts and scripts can be packaged into modules.

PowerShell provides full access to COM and WMI, enabling administrators to manage both local and remote Windows systems, as well as remote Linux systems and network devices to enable WS-management and CIM. PowerShell also provides a hosting API with which the PowerShell runtime can be embedded inside other applications. These applications can then use the PowerShell functionality to implement certain operations, including those exposed through a graphical interface. This capability has been used by Microsoft Exchange Server 2007 to reveal its management functionality as PowerShell cmdlets and providers and implement the graphical management tools as PowerShell hosts that implement the required cmdlets. Other Microsoft applications, including Microsoft SQL Server 2008, also expose their management interface through PowerShell cmdlets.

PowerShell includes its extensive, console-based help (similar to the man pages of Unix shells) accessible through the Get-Help cmdlet. Local help content can be obtained from the Internet through the update-help cmdlet. Alternatively, help from the web can be obtained on a case-by-case basis, for on-demand help via the -online switch.

Every version of Microsoft Windows for personal computers includes a command-line interpreter (CLI) for managing the operating system. Its predecessor, MS-DOS, relied exclusively on a CLI. These are COMMAND.COM in MS-DOS and Windows 9x and cmd.exe in the Windows NT family of operating systems. Both support some basic internal commands. For other purposes, a separate console application must be written. They also include a basic scripting language (batch files), which can

be used to automate various tasks. However, they cannot be used to automate all aspects of graphical user interface (GUI) functionality, as the command-line equivalents of operations are limited, and the scripting language is primary. In Windows Server 2003, the situation was improved, but scripting support was still unsatisfactory.

Microsoft tried to overcome some of the drawbacks by introducing Windows Script Host and its command-line based host: `cscript.exe` in 1998 with Windows 98. It integrates with Active Script Engine and allows scripts to be written in compatible languages, such as JScript and VBScript, leveraging APIs exposed by applications via COM. However, it has its drawbacks: its documentation is not very accessible, and it has gained a reputation as a system vulnerability vector after the vulnerability of many high-profile computer viruses to its security provisions. Different versions of Windows provided various special-purpose command-line interpreters (such as `.Net` and `WMIC`) with their own command sets but were not interoperable.

In an interview published on September 13, 2017, Jeffrey Snover explained the motivation for the project:

I'm creating a bunch of manage changes, and then I basically took the UNIX tool and made them available on Windows, and then it didn't work. right? Because there is a main architectural difference between Windows and Linux. On Linux, everything is an ASCII text file, so anything that can be a management tool. AWK, grep, sed? happy Days!

I brought those tools available to Windows, and then they did not help manage Windows because, in Windows, everything is an API that returns structured data. Therefore, it did not help. [...] I came up with this idea of PowerShell, and I said, "Hey, we can do it better."

By 2002, Microsoft began developing a new approach to command line management, including a CLI called Monad (also known as Microsoft Shell or MSH). The ideas behind it were published in August 2002 in a white paper called Monad Manifesto. Monad was to have a new extensible CLI with a new design that would be able to automate a full range of core administrative tasks. Microsoft first featured Monad at the Professional Development Conference in Los Angeles in October 2003. A few months later a private beta program started, which eventually became a public

Microsoft published the first Monad public beta release on 17 June 2005, Beta 2 on 11 September 2005 and Beta 3 on 10 January 2006. Not much later, on April 25, 2006, Microsoft formally announced that Monad had been renamed to Windows PowerShell. , Positioning it as an important part of their management technology offerings. PowerShell's release Candidate 1 was released around the same time. An important aspect of both the name and RC was that it was now a component of Windows, not an add-on product.

PowerShell version 1 release Candidate 2 was released on November 26, 2006, with a final release for the Web (RTW) on November 14, 2006, and announced at TechEd Barcelona. PowerShell for earlier versions of Windows was released on January 30, 2007.

PowerShell v2.0 development started before PowerShell v1.0 shipped. During development, Microsoft sent three Community Technology Previews (CTP). Microsoft made these releases available to the public. The last CTP release of Windows PowerShell v2.0 was made available in December 2008.

The builders of PowerShell based on the core grammar of the device on the POSIX 1003.2 Korn shell.

### **Windows PowerShell can execute four types of named commands:**

- cmdlets (designed to attach .NET Framework packages to PowerShell)
- PowerShell script (files filed with .ps1)
- PowerShell function
- Standalone executable program

If a command is a standalone executable program, PowerShell launched it in a separate process; If it is a cmdlet, it executes in the PowerShell process. PowerShell provides an interactive command-line interface from which instructions can be entered and their output displayed. The consumer interface, based primarily on the Win32 console, completes customizable tabs. PowerShell allows the creation of aliases for cmdlets, which explain the text in invoices for PowerShell unique commands. PowerShell supports each of the named and positional parameters for commands. To execute a cmdlet, the task of binding the argument value to a parameter is obtained only through PowerShell, but for external executions, arguments are rendered with

the help of an external executable independently of the PowerShell interpretation. [citation needed]

PowerShell Extended Type System (ETS) is based entirely on .NET type systems, but with extended semantics (for example, Properties and third-party extensibility). For example, it enables creating one type of view of objects with the help of exposing only a subset of custom fields, properties, and methods, as well as specifying custom formatting and sorting behavior. These ideas are mapped to the original purpose of using XML-based configuration files.

### **cmdlets**

Cmdlets are special commands in PowerShell environments that implement precise functions. These are the basic instructions in the PowerShell stack. Cmdlets follow a Verb-Noun naming pattern, such as Get-ChildItem, to make them self-descriptive. Cmdlets produce their results as objects and can additionally receive objects as inputs, making them suitable for use as recipients in a pipeline. If a cmdlet outputs multiple objects, each object in the collection is passed through the entire pipeline, allowing the next object to be processed.

Cmdlets are special .NET classes that instantiate the PowerShell runtime and invoice at run-time. Cmdlets derive from both Cmdlet or PSCmdlet, the latter being used when the cmdlet wants to interact with the PowerShell runtime. These base instructions specify positive methods - Begin Processing (), Process Record () and End Processing () - which override the cmdlet's implementation to complete the functionality. Whenever a cmdlet is run, PowerShell implements these methods in sequence, referenced with Process Record () such that it receives pipeline input. If a series of objects are piped, the method is applied to each object in the collection. The class implementing Cmdlet must have a .NET attribute - Cmdlet Attribute - that specifies the verb and noun that makes up the cmdlet title. Common verbs are supplied as an enum.

If a cmdlet receives pipeline input or command-line parameter input, the class with the matador implementation needs to have a similar property.

PowerShell invokes matadors with parameter costs or pipeline inputs, which are saved with the help of mutant implementations in the categorical variables. These values are referenced using techniques that implement the

functionality. Properties that map to command-line parameters are marked by Parameter Attribute and set before calling Begin Processing (). Those that map to pipeline input are also flanked via the Parameter Attribute, but with the Value from Pipeline attribute parameter set.

Implementation of these cmdlet directives can refer to any .NET API and can also occur in any .NET language. In addition, PowerShell makes APIs available, such as Write Object, which are used to gain access to PowerShell-specific functionality, such as writing resulting objects in a pipeline. Cmdlets can use .NET data to gain access to the API directly or use the PowerShell infrastructure of PowerShell Providers, which makes the data repository usable by specific paths. Data shops use the use of force sheets and hierarchies inside them, which are addressed as directories. Windows PowerShell ships vendors with filesystems, registries, certificate stores, as well as namespaces for command aliases, variables, and functions. Windows PowerShell has several types of cmdlets for managing a range of Windows systems, including the file system, or using Windows Management Instrumentation to manipulate Windows components. Other functions can register cmdlets with PowerShell, therefore allowing it to manage them, and, if they attach a data store (such as a database), they can also add specific companies. [citation needed]

PowerShell V2 gave a larger portable version of the cmdlets known as modules. Status of PowerShell V2 launch notes:

The module allows script developers and directors to partition and organize their Windows PowerShell code into self-contained, reusable units. The code from a module executes in its own context and has no effect on the state outdoor of the module. The module additionally enables you to define a finite runway environment through the use of scripts.

### **line pipe**

PowerShell applies the idea of a pipeline, which allows us to pipe the output of a cmdlet to any other cmdlet as input. For example, the output of a gate-process cmdlet to filter out any method where the object can be piped to filter a phased memory of less than 1 MB, and then type the Sort-Object cmdlet (eg, type To) objects by calculation), and then to select only the first 10 (i.e., 10 processes based primarily on the manage count) to the Select-Object cmdlet. [citation needed]

As with the Unix pipeline, the PowerShell pipeline can assemble complex commands, using | To join operator steps. However, the PowerShell pipeline differs from the Unix pipelines, in the degree to which they run alternatively within the PowerShell runtime as a set of coordinated approaches with the help of running systems, and structured rather than byte streams. NET objects are more than one step. next. PowerShell wants to serialize data structures using objects and tiers inside the runtime or extract them to explicitly parse text content output. An object can also contain positive functions that operate on implicit data, which become accessible to the recipient command for use. For the remaining cmdlet in a pipeline, PowerShell routinely pipes its output object to the out-default cmdlet, which transforms the objects into a circulation of the structure object and then presents them to the screen.

Because all PowerShell objects are .NET objects, they share a ToString () method, which retrieves the text representation of information in an object. In addition, PowerShell allows definitions to be specified, so literal content delineation of objects can be customized through choosing which information elements to display and in which ways. However, to maintain backward compatibility, if an external executable is used in a pipeline, it receives a textual content rendering the object, optionally once integrated with a system like PowerShell. it happens.

## **Scripting**

Windows PowerShell includes a dynamically typed scripting language, which can essentially complicate the use of cmdlets. The scripting language supports variables, functions, branching (if-then-then), loops (while, do, for, and foreach), structured error/exception encounters and / with lambda expressions, [41] k. As integration with. met. Variables in PowerShell scripts are prefixed with \$. Variables can be assigned any value with the output of cmdlets. Strings can be enclosed in single charges or double quotation marks: when using double quotation marks, variables will be expanded even if they are inside quotation marks. The inclusion of the syllabus in a file in the first braces (as in \$ {C: \ foo.txt}) with the help of the greenback hint makes a reference to the contents of the file. If it is used as an L-value, anything assigned to it will be written to the file. When used as an R-value, the contents of the file will be read. If an item is assigned, it is sorted before it is stored. [citation needed]

Objects can be accessed using contributors. Notation, as in the C # syntax. PowerShell returns specific variables, such as \$ args, which is an array of all command-line arguments more than a feature from the command line, and \$ \_ , which refers to the modern object in the pipeline. [42] PowerShell also pastes arrays and associative arrays. The PowerShell scripting language also immediately evaluates arithmetic expressions entered on the command line, and it parses consecutive abbreviations such as GB, MB, and KB.

To create PowerShell functions, using the function keyword, gives the following universal form:

```
Function Name ($ Param1, $ Param2)
{
instructions
}
```

The defined attribute applies in one of the following forms:

- Name value1 value2
- Name - Param 1 Value 1 - Param 2 Value 2

PowerShell supports named parameters, positional parameters, switch parameters, and dynamic parameters.

PowerShell lets any .NET methods go through the supply of their namespaces enclosed in parentheses ([ ]), and then lets the pair of the colon ( : ) be used to indicate the static method. For example, [System.Console] :: WriteLine ("PowerShell"). Objects are constructed using the new-object cmdlet. The calling methods of .NET objects are accomplished by the way they are used every day. Notation.

PowerShell accepts strings, each uncapped and survived. A string enclosed between single quotation marks is a unique string, while a string enclosed between double quotation marks. PowerShell considers straight and curly prices the same.

The following list of one type of characters is supported with PowerShell:

PowerShell special characters



```
Windows PowerShell
PS C:\Users> Get-Help "Rename-Item"
NAME
    Rename-Item

SYNTAX
    Rename-Item [-Path] <string> [-NewName] <string> [-Force] [-PassThru] [-Credential
    <pscredential>] [-WhatIf] [-Confirm] [-UseTransaction] [<CommonParameters>]

    Rename-Item [-NewName] <string> -LiteralPath <string> [-Force] [-PassThru]
    [-Credential <pscredential>] [-WhatIf] [-Confirm] [-UseTransaction]
    [<CommonParameters>]

ALIASES
    rni
    ren

REMARKS
    Get-Help cannot find the Help files for this cmdlet on this computer. It is displaying
    only partial help.
    -- To download and install Help files for the module that includes this cmdlet,
    use Update-Help.
    -- To view the Help topic for this cmdlet online, type: "Get-Help Rename-Item
    -Online" or
    go to https://go.microsoft.com/fwlink/?LinkID=113382.

PS C:\Users>
```

## Interpretation

- ``0` tap
- ``A` warning
- ``B` Backspace
- ``E` escape
- ``F` Form Feed
- ``N` newline
- ``R` carriage return
- ``t` horizontal tab
- ``U {x}` Unicode escape sequence
- ``v` vertical tab
- `-%` to stop

To deal with the error, PowerShell provides a .NET-based exception-handling mechanism. In the case of errors, objects containing statistics about

the error (exception object) are thrown, which captures the use of the attempt ... gather (although a trap collect is also supported). PowerShell can be configured to resume execution silently, leaving only the exception; It can be terminated on a single command, single session, or both permanent.

PowerShell can be used to maintain written scripts in a ps1 file or psm1 file (the latter is used as a module) in both classes throughout the classes. Later, either the entire script or a male or female function can be used in the script. Scripts and attributes operate similarly with cmdlets, in which they can be used as commands in pipelines, and parameters can be ensured for them. Pipeline objects can be passed natively between functions, scripts, and cmdlets. To forestall accidental jogging of scripts, script execution is disabled by default and must be explicitly enabled. Enabling the script can be performed at both the system, individual, or session-level. PowerShell scripts can be signed to confirm their integrity, and code access is the problem for security.

The PowerShell scripting language supports binary prefix notation, similar to scientific notation supported with the help of several programming languages in the C-family.

## **hosting**

Additionally, an administration can use embedded PowerShell in the application, which uses PowerShell runtime to implement administration functionality. For this, PowerShell provides a managed web hosting API. Through the API, the software can instantiate a run space (an instance of the PowerShell runtime), which runs in the application's process and is exposed as a run space object. The country of the run space is enclosed in a SessionStat object. When Runspace is created, Windows PowerShell initializes the runtime instance, which includes initializing vendors and enumerating cmdlets and updating the SessionState object accordingly. The run space must then open for both synchronous processing or asynchronous processing. It can then be used to execute commands.

To execute a command, a pipeline (represented via a pipeline object) needs to be created and associated with the run space. The pipeline object is then populated with the cmdlets that make up the pipeline. For sequential operations (as a PowerShell script), a Pipeline object is created for each declaration and nested inside another Pipeline object. When a pipeline is

created, Windows PowerShell invokes the pipeline processor, which resolves the cmdlets into their respective assemblies (command processors) and adds a reference to them in the pipeline, and assigns them to the input pipe, output pipe, and error output pipe objects. Represents with. Relationship with the pipeline. The types are verified and fixed using reflection. Once the pipeline is established, the host calls the invoke () technique to run the command, or its asynchronous equivalent – invoke and sink (). If the pipeline has a right-host cmdlet above the pipeline, it writes the result to the console screen. If not, the effects are passed to the host, which can both follow further processing or show the output.

Microsoft Exchange Server 2007 uses the Web Hosting API to present its administration GUI. Each operation exposed in the GUI is mapped to a sequence of PowerShell instructions (or pipelines). The host creates pipelines and executes them. In fact, the interactive PowerShell console itself is a PowerShell host, which interprets scripts entered at the command line and creates and invokes important pipeline objects.

### **Desired state configuration**

Allows DSC to declare how the software environment should be configured.

When strolling a configuration, DSC will ensure that the machine acquires the nation described in the configuration. DSC configurations are lazy. The Local Configuration Manager (LCM) periodically pollutes the gadget with the help of assets with a described manipulation float (mandatory fraction of DSC) to ensure that the state of the configuration is maintained.

### **Edition**

Initially using the code name "Monad", PowerShell was first publicly proven at the Professional Developers Conference in September 2003. All primary releases are still supported, and each major release has shown backward compatibility with previous versions.

#### **PowerShell 1.0**

PowerShell 1.0 was released in November 2006 for Windows XP SP2, Windows Server 2003 SP1 and Windows Vista. [54] It is a non-mandatory element of Windows Server 2008.

PowerShell 2.0 is integrated with Windows 7 and Windows Server 2008 R2 and has been launched for Windows XP with Service Pack 3, Windows

Server 2003 with Service Pack 2, and Windows Vista with Service Pack 1.

PowerShell v2 includes modifications to the scripting language and hosting APIs, in addition to including more than 240 new cmdlets.

New points of PowerShell 2.0 include:

**PowerShell Remoting:** Using WS-Management, PowerShell 2.0 allows scripts and cmdlets to be mounted on a remote machine or a large set of remote machines.

**Background Jobs:** Also known as PSJob, it lets a command sequence (script) or pipeline be implemented in an assembly. Jobs can be run on a nearby computing device or on more than one remote machine. An interactive cmdlet in PSJob blocks the execution of a job until the user is granted access.

**Transaction:** Enable cmdlet and developers can perform transactional operations. PowerShell 2.0 includes starting, committing and redoing a transaction as well as a collaborating cmdlet and transaction cmdlets to direct transactions to the company's operations. The PowerShell registry company supports transactions.

**Advanced functions:** These cmdlets are written using the PowerShell scripting language. Initially called "script cmdlets", the feature was later renamed "Advanced Functions".

**Steppable Pipeline:** This allows the consumer when the BeginProcessing (), ProcessRecord () and EndProcessing () functions of a cmdlet are added.

**Module:** This script allows developers and directors to arrange and partition PowerShell scripts into self-contained, reusable units. The code from a module executes in its individual self-contained context and does not affect the country outside the module. Modules can outline a constrained run space environment using the use of a script. They have public and private members as well as a continuous state.

**Data language:** A domain-specific subset of the PowerShell scripting language that allows data to be separated from scripts and imported localized string assets into the script at runtime (script internationalization).

**Script Debugging:** This allows breakpoints to be set in a PowerShell script or function. Breakpoints can be set on lines, lines and columns, commands, and the entry of variables can be obtained. It consists of a set of cmdlets to manipulate breakpoints through scripts.

**Eventing:** This feature allows administration, and machine events to be heard,

forwarded, and displayed. The event allows PowerShell hosts to notify their managed entities about nationwide adjustments. It additionally subscribes PowerShell scripts to ObjectEvents, PSEvents, and WmiEvents and types them in a synchronous and asynchronous manner.

Windows PowerShell Unified Scripting Environment (ISE): PowerShell 2.0 includes a GUI-based PowerShell host, which includes, with a built-in debugger, syntax highlighting, tab completion, and 8 PowerShell Unicode-enabled (Runspaces) tabs. Ability to run only selected components in a script.

Nerk File Transfer: Basic support for priority, throttle, and asynchronous switches of documents between machines using the Background Intelligent Transfer Service (BITS).

New cmdlets: including out-grid-view, which shows tabular facts in the WPF GridView object, on the structures that enable it, and if ISE is set and enabled.

New operators: -Split, -Join, and Splatting (@) operators.

Exception Handling with Try-Catch - Finally: Unlike various .NET languages, it approves more than one exception type for a single trap block.

Nestable Hear-Strings: The PowerShell Here-Strings has been expanded and can now nest.

Block Comments: PowerShell 2.0 helps block the response to the use of <# and #> as delimiters.

New APIs: New APIs differ from PowerShell parser and host to perform additional manipulation at runtime, to create and manage a series of Runspaces (RunspacePools) as well as the ability to create restricted Runspaces that are just a configured subgroup of PowerShell allows for. Has been activated. The new APIs also assist in participation in a Windows PowerShell managed the transaction.

### PowerShell 3.0

PowerShell 3 is integrated with Windows Eight and Windows Server 2012.

Microsoft has simplified PowerShell 3 for Windows 7 with Service Pack 1 for Windows Server 2008 with Service Pack 1 and for Windows Server 2008 R2 with Service Pack 1.

```
PowerShellExamples.ps1 X
1 $PSVersionTable
2 Get-Help
3 clear
4
5 New-Item -Path 'G:\padmini\NewPowerShellFolder' -ItemType Directory

New-Item : An item with the specified name G:\padmini\PowerShellFolder already exists.
At G:\Users\Administrator\Desktop\PowerShellExamples.ps1:4 char:1
+ New-Item -Path 'G:\padmini\PowerShellFolder' -ItemType Directory
+ ~~~~~
+ CategoryInfo          : ResourceExists: (G:\padmini\PowerShellFolder:String) [New-Item], IOException
+ FullyQualifiedErrorId : DirectoryExist,Microsoft.PowerShell.Commands.NewItemCommand

PSPath           : Microsoft.PowerShell.Core\FileSystem::G:\padmini\NewPowerShellFolder
PSParentPath     : Microsoft.PowerShell.Core\FileSystem::G:\padmini
PSChildName      : NewPowerShellFolder
PSDrive          : G
PSProvider       : Microsoft.PowerShell.Core\FileSystem
PSIsContainer    : True
Name             : NewPowerShellFolder
FullName        : G:\padmini\NewPowerShellFolder
Parent           : padmini
Exists           : True
Root            : G:\
Extension        :
CreationTime     : 9/24/2018 7:31:49 PM
CreationTimeUtc  : 9/24/2018 2:01:49 PM
LastAccessTime   : 9/24/2018 7:31:49 PM
LastAccessTimeUtc : 9/24/2018 2:01:49 PM
LastWriteTime    : 9/24/2018 7:31:49 PM
LastWriteTimeUtc : 9/24/2018 2:01:49 PM
Attributes       : Directory
Mode            : d-----
BaseName        : NewPowerShellFolder
Target          : {}
LinkType        :
```

PowerShell 3 is part of a larger package, Windows Management Framework 3 (WMF3), which also has a WinRM service to aid remoting. Microsoft released several community technology previews for WMF3. An early Community Science Preview 2 (CTP 2) version of Windows Management Framework 3 was launched on 2 December 2011. The Windows Management Framework was launched on 3 December 2012 for time-honored availability and is included by Windows Eight and Windows Server 2012. Default mode.

**Scheduled jobs:** Jobs can be scheduled to run at a predetermined time and date.

**Session Connectivity:** Sessions can be disconnected and reconnected. Remote classes have become more tolerant of temporary neck failures.

**Improved code writing:** Code completion (IntelliSense) and snippets are added. PowerShell ISE lets customers use dialog packing containers to fill the parameters for PowerShell cmdlets.

**Delegate support:** Administrative tasks may be delegated to clients who no longer have permission for that type of work, in addition to regular additional permissions.

Help update: The help document can be updated through the update-help command.

Automatic module detection: The module is loaded every time a command is invoked from that module. The code works for unloaded modules as soon as they are completed.

New commands: Added dozens of new modules, such as performance-WmiObject win32\_logicaldisk, volume, firewalls, network connections, and printer management, so far performed via WMI, to create performance disturbances. [Requires further explanation]

#### PowerShell 4.0

PowerShell 4 is built with Windows 8.1 and Windows Server 2012 R2.

Microsoft has made PowerShell 4 available for Windows 7 SP1, Windows Server 2008 R2 SP1 and Windows Server 2012.

#### **New aspects in PowerShell 4 include:**

Desired state configuration: declarative language extensions and tools that enable the deployment and administration of configuration data for structures using DMTF administration requirements and WS-management protocols

New default execution policy: On Windows Server, the default execution policy is now RemoteSigned.

Save-Help: Help can now be saved for modules that are installed on remote computers. Increased Debugging: The debugger now helps in debugging workflows, eliminating script execution, and debugging periods to reconnect a PowerShell session.

-Pipeline variable switch: a new ubiquitous parameter to expose the current pipeline object as a variable for programming purposes

Physical diagnostics and to manage Hyper-V's virtualized community switch

The Where and ForEach technique provides an alternative method of filtering and iterating over syntax objects.

#### PowerShell 5.0

Windows Management Framework (WMF) 5.0 RTM including PowerShell 5.0, which was re-released on the Net on February 24, 2016, after its initial release on February 24, 2016, with an extreme bug. Key points include providing support for OneGet PowerShell cmdlets to support Chocolatey's repository-based package administration, and to layer 2 community switches for change management.

```
Windows PowerShell
PS C:\Users\ramya-pc> Get-Service

Status Name DisplayName
-----
Running AdobeARMService Adobe Acrobat Update Service
Stopped AdobeFlashPlaye... Adobe Flash Player Update Service
Stopped airtel. RunOuc airtel. OUC
Stopped AJRouter AllJoyn Router Service
Stopped ALG Application Layer Gateway Service
Stopped AnimationService AnimationService
Stopped AppIDSvc Application Identity
Running Appinfo Application Information
Running Apple Mobile De... Apple Mobile Device Service
Stopped AppReadiness App Readiness
Stopped AppXSvc AppX Deployment Service (AppXsvc)
Running aswbIDSAgent aswbIDSAgent
Running AudioEndpointBu... Windows Audio Endpoint Builder
Running Audiosrv Windows Audio
Running avast! Antivirus Avast Antivirus
Stopped AxInstSV ActiveX Installer (AxInstSV)
Running BcmBTRSupport Bluetooth Driver Management Service
Stopped BDESVC BitLocker Drive Encryption Service
Running BFE Base Filtering Engine
Running BITS Background Intelligent Transfer Ser...
Running Bonjour Service Bonjour Service
Running BrokerInfrastru... Background Tasks Infrastructure Ser...
Running Browser Computer Browser
Stopped BthHFSrv Bluetooth Handsfree Service
Stopped bthserv Bluetooth Support Service
Running CDPSvc Connected Devices Platform Service
Running CDPUsersvc_d08ca CDPUsersvc_d08ca
Stopped CertPropSvc Certificate Propagation
Running ClickToRunSvc Microsoft Office ClickToRun Service
Stopped ClipSvc Client License Service (Clipsvc)
Stopped COMSvcApp COM+ System Application
```

## New aspects of PowerShell 5.0 include:

- PowerShell Classification Definitions (Properties, Methods)
- PowerShell .NET Enumeration
- Debugging for remote processes in PowerShell Runspaces
- Debugging for PowerShell background jobs
- Desired State Configuration (DSC) Local Configuration Manager (LCM) version 2.0
- DSC Partial Configuration
- DSC Local Configuration Manager Meta-Configuration
- Authoring DSC resources using PowerShell classes

## PowerShell 5.1

It was once released with the Windows 10 Anniversary Update on August 2, 2016, and in Windows Server 2016. BackManagement now supports proxies, PSReadLine now supports ViMode, and new cmdlets have been added: get-timezone and set-timezone. The LocalAccounts module allows for adding/removing local person accounts. A preview for PowerShell 5.1 for Windows 7, Windows Server 2008, Windows Server 2008 R2, Windows Server 2012 and Windows Server 2012 R2 was launched on July 16, 2016, and was launched on January 19, 2017.

PowerShell 5.1 is the first version to come in variants of "desktop" and "core". The "desktop" version is a continuation of the standard Windows



PowerShell that runs on the full .NET Framework stack. The "core" version runs on .NET Core and is bundled with Windows Server 2016 Nano Server. In exchange for the small footprint, the latter lacks some features such as cmdlets to control the clipboard or a domain of the computer, part of the WMI Model 1 cmdlets, event log cmdlets, and profiles. This was the rest of PowerShell's model built entirely for Windows.

### **PowerShell Core 6.0**

PowerShell Core 6.0 was first introduced on August 18, 2016, when Microsoft unveiled PowerShell Core and selected it to make the product cross-platform, independent of Windows, free and open source. It completed time-honored availability on 10 January 2018 for Windows, macOS, and Linux. It has a personalized support lifecycle and follows the Microsoft lifecycle policy brought with Windows 10: only the modern model of PowerShell Core is supported. Microsoft hopes to launch a mini version for PowerShell Core 6.0 every six months.

The most full-size change in this version of PowerShell is development for other platforms. For Windows administrators, this model of PowerShell is devoid of any basic new features. In an interview with the neighborhood on eleven January 2018, the PowerShell team was asked to list the top 10 most thrilling things a Windows IT expert would look for, migrating from Windows PowerShell 5.1 to PowerShell Core 6.0; In response, Microsoft's Angel Calvo can identify only: cross-platform and open-source.

### **PowerShell Core 6.1**

Compatibility with 1900+ current cmdlets in Windows 10 and Windows Server 2019

Built on top of .NET Core 2.1

Support for the latest versions of Windows, macOS, and Linux

- Significant overall performance improvement
- Markdown cmdlets
- Experimental feature flags

### **PowerShell Core 6.2**

The PowerShell Core 6.2 launch has been specifically targeted at overall performance improvements, worm improvements, and smaller

cmdlet/language enhancements that enhance survival for users.

## **PowerShell hell**

PowerShell 7 should be developed as an optional product for PowerShell Core 6.x merchandise, such as Windows PowerShell 5.1, which is the remaining supported Windows PowerShell version. For PowerShell 7 to be an attainable option for Windows PowerShell 5.1 it must have parity with Windows PowerShell in phrases of compatibility with modules that ship with Windows.

## **Comparing cmdlets with comparative commands**

The following desk includes a selection of cmdlets that ship with PowerShell, taking into account comparable instructions in various common command-line interpreters. Many similar instructions come out-of-the-box defined as aliens inside PowerShell, making it easier for human's familiar with different shells to initiate various tasks.

### **1: PS1 files**

A PowerShell script is without a doubt nothing more than an easy textual file. The file is a collection of PowerShell commands, with each command acting on a separate line. The text content file is treated as a PowerShell script, its filename intends to use the PS1 extension.

### **2: execution permissions**

To prevent the execution of malicious scripts, PowerShell enforces an execution policy. By default, the execution policy is restricted, which will no longer run the ability to run PowerShell scripts. You can determine modern-day performance coverage using the following cmdlet:

### **Get-Execution Policy**

The execution policies you can use are:

#### **Restricted - Scripts cannot run.**

Remote Sign - regionally created scripts will run, although they will not be downloaded from the Internet (unless they are digitally signed through the publisher).

All Signed - Scripts will only run if they are signed with the help of a trusted publisher.

Unrestricted - Scripts will run regardless of where they come from and whether they are signed or not.

### **3: running script**

For years now, if you want to run an executable file from the command line, the practice is to navigate the path of the file and then type the name of the executable file. However, this age-old technique does not work for PowerShell scripts.

If you prefer to execute a PowerShell script, you will need to write the complete syllabus with the file name as usual. For example, to run a script called SCRIPT.PS1, you can type:

#### **C: \ Script \ Script.ps1**

The big exception is that you can execute a script with no doubt typing its name if the folder containing the script is in the path of your system. You can also use a shortcut if you are already in a folder containing the script. In such a case, instead of typing the full route of the script, you can enter. \\_ And script name. For example, you can type:

. \ Script.ps1

### **4: pipelining**

Pipelining is the period of time the output of a command is fed to every other command. This allows the second command to process the entry is received. To pipeline instructions (or cmdlets), separate them honestly with the pipe symbol (|).

To help you understand how pipelining works, you choose to create a list of tactics that is running on the server and perform that kind of listing using the process ID number. You can get a list of techniques with the help of the use of the get-process cmdlet, but the list will no longer be sorted. However, if you pipeline the output of the cmdlet to the Sort-Object ID command, the list will be sorted. The command used looks like this:

Go-process | Sort-object id

### **5: variable**

Although you can use pipelining to feed the output of one command to another command, sometimes pipelining by itself may not get the job done. When you pipeline the output of a command to every other command, that output is used immediately. Sometimes, you may also need to keep the output for some time so that you can use (or reuse) it later. This is where the variables come into play.

It is easy to think of a variable as a repository for storing a value, but in PowerShell, a variable can store the full output of a command. For example, suppose you choose to have a list of technologies jogging on the server as a variable. To do this, you may want to use this line of code:

**\$ a = get-process**

Here, the variable name is \$ a. If you wish to use the variable, name it with the help of the name. For example, typing \$ prints the contents of the variable on the screen.

You can assign a variable to the final output of several commands that are piped together. Surround the instructions with parentheses only. For example, to type jogging strategies through the process ID and then assign the output to the variable, you may want to use this command:

**\$ a = (Get-Process | Sort-Object ID)**

## **6: @ symbol**

Using the @ symbol, you can flip the contents of a list into an array. For example, take the following line of code, which creates a variable named \$ Procs that contains multiple traces of textual content (an array):

```
$ procs = @ {name = "Explorer", "svc host"}
```

When the variable is used, you can additionally use the @ image, to ensure that it is treated as a value, as an array. For example, the line of code below will run a gate-process cmdlet as opposed to a variable defined a moment ago. In doing so, Windows will show all the processes used with the help of Windows Explorer and Seacoast. Note that the @ symbol in front of the variable name is being used as an alternative to the greenback signal, which we usually use:

**Get-Process @procs**

## **7: division**

The breakup operator primarily splits a text string based on the character you named. For example, consider that you want to damage a sentence in a sentence that includes every individual word in the sentence. You can do this by using a command like this:

```
"This is a test" -Split ""
```

The end result will look like this:

## 8: join

Just as a cut-up can cut a text string into a few pieces, part of the operator can combine some blocks of text into one. For example, this line will create a text string consisting of my first name and last name:

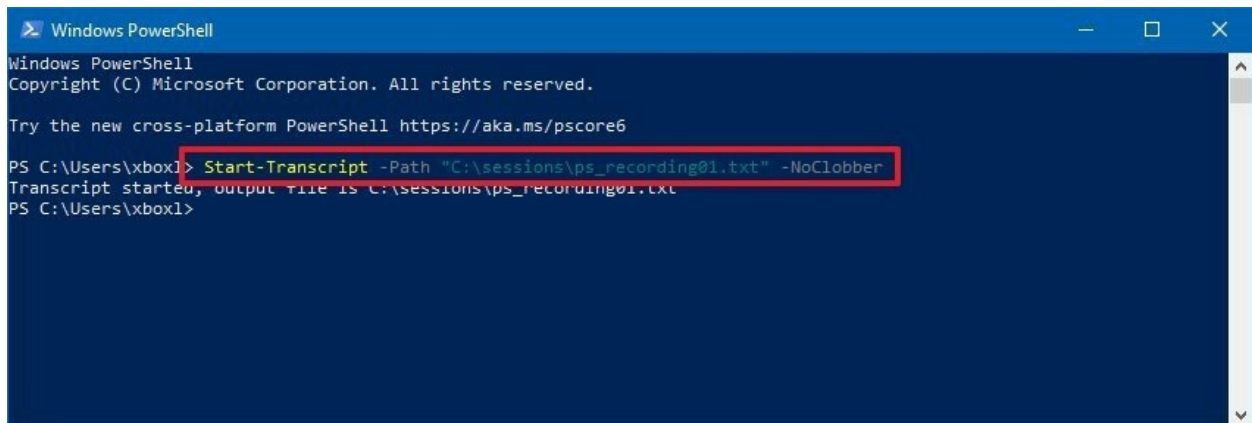
**"Brian", "Posey" -Join ""**

The field between quotation marks at the top of the command tells Windows to insert a house between text strings.

## 9: breakpoint

Running a newly created PowerShell script may result in unexpected penalties if the script incorporates a bug. One way to protect yourself is to place breakpoints at strategic locations inside your script. In this way, you can ensure that the script is working as before rather than fully working.

The line number is the best way to insert a breakpoint. For example, to place a smashing point on the 10th line of a script, you can use a command like this:

A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell". The window content shows the following text: "Windows PowerShell", "Copyright (C) Microsoft Corporation. All rights reserved.", "Try the new cross-platform PowerShell https://aka.ms/pscore6", "PS C:\Users\xbox1> Start-Transcript -Path 'C:\sessions\ps\_recording01.txt' -NoClobber", "Transcript started, output file is C:\sessions\ps\_recording01.txt", and "PS C:\Users\xbox1>". The command line is highlighted with a red rectangular box.

New-PS Breakpoint -Script C: \ Scripts \ Script.ps1 -Line 10

You can additionally bind a breakpoint to a variable. So if you wanted your script to waste any time \$ content, you could use a command like this:

New-PS Breakpoint -Script C: \ scripts \ Script.ps1 -variables a

Note that I did not include the greenback signal after the variable name.

There are several types of actions that you can use with PS Breakpoint such as New, Get, Enable, Disable, and Remove.

## 10: step

When debugging a script, it may be necessary to run the script line through

the line. To do this, you can use the Step-Into cmdlet. This will prompt the script to pause after each line whether a breakpoint exists or not. When you're done, you can use the step-out cmdlet to prevent Windows from stepping through the script. However, it is worth noting that the breakpoint has not been processed even after the step-out cmdlet has been used.

Incidentally, if your script uses functions, you can also join the step-over cmdlet. Step-over works the same way as step-into, if without this function is called, then there may be no Windows step. The entire facility will run non-stop.

Windows PowerShell is an object-oriented automation engine and scripting language. This gadget is designed for administrators frequently. It helps IT professionals in manipulating and automating the administration of Windows OS and other applications.

It gave some compelling new standards that enable you to stretch

The understanding and scripts you won are created within the Windows command prompt and the Windows Script Host environment.

It combines the power of scripting, command-line speed, and GUI-based admin tools. This lets you correctly solve problems by helping the machine administrator overcome future manual labor hours. We will go through all the important issues that you have to research on PowerShell.

In this coaching course, you will learn

- What is PowerShell?
- Why Use PowerShell?
- PowerShell history
- PowerShell Features
- How to launch PowerShell
- PowerShell Cmdlet
- Cmdlet vs command:
- PowerShell data type:
- Special variable
- PowerShell scripts
- First PowerShell script
- What is PowerShell ISE?

- PowerShell concepts
- PowerShell vs Command Prompt
- PowerShell applications
- This is a complete guide to PowerShell ... let's get started!

## Why Use PowerShell?

Here, there are some essential reasons for using PowerShell:

PowerShell confirms a well-integrated command-line trip to the operation system

PowerShell provides complete access to all types of .NET Framework Trusted with the help of system administrators.

PowerShell is an easy way to manipulate server and laptop components It is geared towards device directors through developing a more simple syntax.

PowerShell is more impregnable than strolling VBScript or other scripting languages

PowerShell history

The first version 1.0 of PowerShell was launched in 2006. Today, PowerShell is on the 5.1 model. As of 12 months and version long, PowerShell's capabilities and Internet hosting environment have increased significantly.

## **Let's look at the history of the version Smart History:**

PowerShell version 1 supported the administration of Windows Server 2003 Pass

PowerShell 2.0 was integrated with Windows 7 and Windows Server 2008 R2. This model enhances the capacity of PowerShell and supports transactions, historical past jobs, events, debugging, and more.

PowerShell 3 was once released as an internal part of the Windows administration framework. It was once installed in Windows Eight and Windows Server 2012. You can add job and schedule connectivity, session connectivity, automatic module loading, etc.

PowerShell Char was once shipped with Windows 8.1 and Windows Server 2012 R2. This version added custom state configuration, more appropriate debugging, support for community diagnostics.

PowerShell 5.0 was launched as an internal segment of the Windows Management Framework. Features in this model are fond of remote

debugging, class definitions, .NET enumerations, and more.

### PowerShell Features

**PowerShell remoting:** PowerShell allows scripts and cmdlets to be mounted on a remote machine.

**Background Jobs:** It helps you to implement a script or pipeline asynchronously. You can run your job on a neighborhood Laptop or a pair of remotely operated machines.

**Transaction:** Enable cmdlet and allow developers to perform

**Evening:** This command helps you manage, listen to, forward and perform gadget events.

**Nerk File Transfer:** PowerShell introduces native guides for prioritized, asynchronous, throttled, transferring archives between machines using Background Intelligent Transfer Service (BITS) technology.

### How to launch PowerShell

PowerShell comes pre-installed in all contemporary versions of Windows.

We need to launch PowerShell, for this we need to follow the given steps:

**Step 1) Search for PowerShell in Windows. Select and click**

**Step 2) PowerShell window opens**

### PowerShell Cmdlet

A cmdlet, additionally referred to as the command lat, is a lightweight command used in a window base PowerShell environment. PowerShell invokes these cmdlets in a command prompt. You can create cmdlets and allow the use of PowerShell APIS.

### Cmdlet vs command:

Cmdlets are unique from commands in the following manners in different command-shell environments -

Cmdlets are .NET Framework objects. It cannot be done separately

Cmdlets can collect from as few as a dozen strains of code

Parsing, output formatting, and error presentation are not handled via cmdlets

The cmdlets process works on objects. Therefore text moves and objects cannot be crossed as output for pipelining

Cmdlets are record-based, so it treats only one object at a time

Most of the performance of PowerShell comes from the Cmdlet which is persistent in the verb-noun structure and not in the plural. Also, Cmdlet's



return objects are not text. A cmdlet is a collection of commands, which is more than a line, stored in a textual content file with a PSL extension.

A cmdlet consists of a verb and a noun, separated by a hyphen. Some actions you use to study PowerShell:

- Get - to get something
- Start - something to run
- Out - something to produce
- Stop - give up what is going on
- Set - to define something
- New - to make something
- PowerShell command

# Process and feature in PowerShell

Windows 10 ships with Windows PowerShell 5.0. Windows 8.1 is installed with Windows PowerShell 4.0. The new version hosts several new features designed to simplify its language, less complex to use, and to keep away from common errors. If you are using the up-to-date model of PowerShell on your Windows running system, migrating to this version of Windows PowerShell will have many benefits. This not only lets gadget administrators manipulate everything on the Windows Server OS, but also provides control over SQL, Exchange, and Lync-based servers.

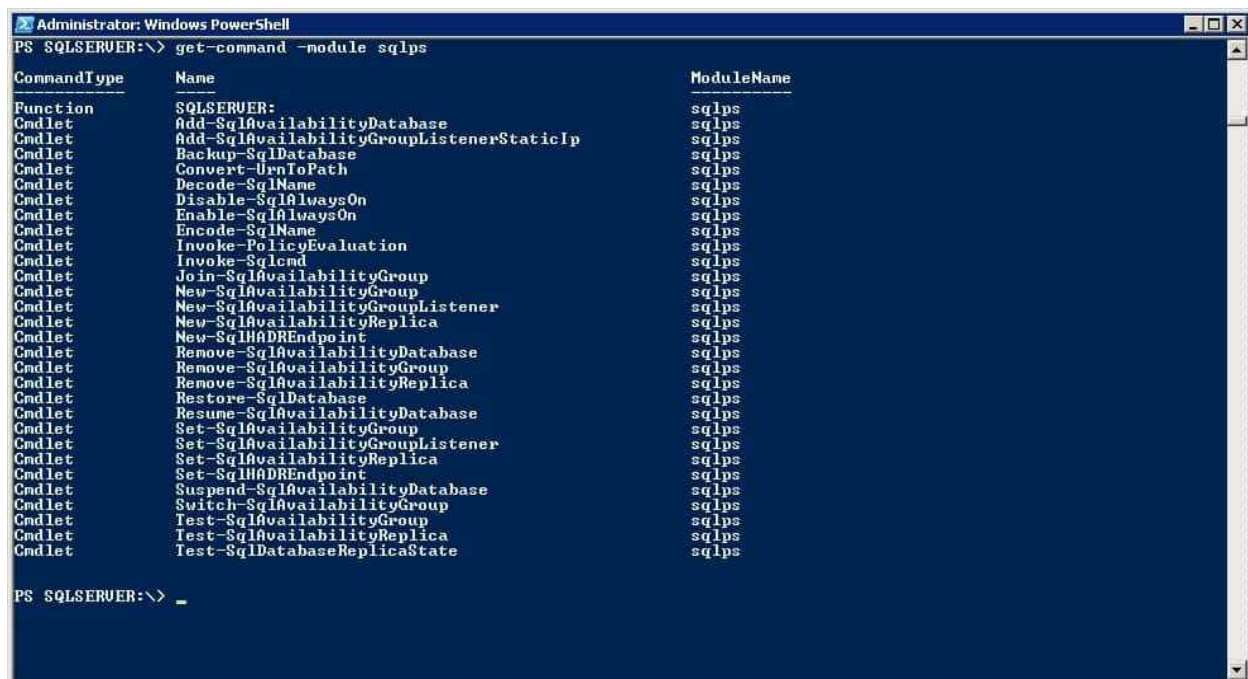
## Which version of PowerShell I is running

To find out which model of PowerShell you are using, do the following.

Windows 7 with Windows Server 2012, Windows Server 2008 R2, Windows Server 2008 SP2, Windows 8 and SP1 customers will be in a position to use Windows PowerShell 3.0.

Windows Server 2012 R2, Windows Server 2012, Windows Server 2008 R2, Windows 8.1 and Windows 7 SP1 will be in a position to use Windows PowerShell 4.0 with customers.

Windows 10 ships with Windows PowerShell 5.0.



```
Administrator: Windows PowerShell
PS SQLSERVER:\> get-command -module sqlps
```

CommandType	Name	ModuleName
Function	SQLSERVER:	sqlps
Cmdlet	Add-SqlAvailabilityDatabase	sqlps
Cmdlet	Add-SqlAvailabilityGroupListenerStaticIp	sqlps
Cmdlet	Backup-SqlDatabase	sqlps
Cmdlet	Convert-UrnToPath	sqlps
Cmdlet	Decode-SqlName	sqlps
Cmdlet	Disable-SqlAlwaysOn	sqlps
Cmdlet	Enable-SqlAlwaysOn	sqlps
Cmdlet	Encode-SqlName	sqlps
Cmdlet	Invoke-PolicyEvaluation	sqlps
Cmdlet	Invoke-Sqlcmd	sqlps
Cmdlet	Join-SqlAvailabilityGroup	sqlps
Cmdlet	New-SqlAvailabilityGroup	sqlps
Cmdlet	New-SqlAvailabilityGroupListener	sqlps
Cmdlet	New-SqlAvailabilityReplica	sqlps
Cmdlet	New-SqlHADREndpoint	sqlps
Cmdlet	Remove-SqlAvailabilityDatabase	sqlps
Cmdlet	Remove-SqlAvailabilityGroup	sqlps
Cmdlet	Remove-SqlAvailabilityReplica	sqlps
Cmdlet	Restore-SqlDatabase	sqlps
Cmdlet	Resume-SqlAvailabilityDatabase	sqlps
Cmdlet	Set-SqlAvailabilityGroup	sqlps
Cmdlet	Set-SqlAvailabilityGroupListener	sqlps
Cmdlet	Set-SqlAvailabilityReplica	sqlps
Cmdlet	Set-SqlHADREndpoint	sqlps
Cmdlet	Suspend-SqlAvailabilityDatabase	sqlps
Cmdlet	Switch-SqlAvailabilityGroup	sqlps
Cmdlet	Test-SqlAvailabilityGroup	sqlps
Cmdlet	Test-SqlAvailabilityReplica	sqlps
Cmdlet	Test-SqlDatabaseReplicaState	sqlps

```
PS SQLSERVER:\> _
```

## Windows PowerShell Features

Windows PowerShell 3 provided the following new functionality:

- Windows PowerShell workflows
- CIM cmdlets
- CDML (CDXML) on Objects
- Windows PowerShell web access
- Module Automatic Loading
- Updatable support
- Strong and hacked session
- Scheduled jobs
- Windows PowerShell 4 Launched:
- Desired State Configuration (DSC)
- Improvements to Windows PowerShell Web Access
- Workflow enhancements
- New aspects of Windows PowerShell web services
- Save-help
- Windows PowerShell 5.0, which is protected in Windows 10, introduces the following functionality:
- Classes can be described in functionality
- DSC enhancements
- Details available in all hosts
- Major enhancements to debugging, including the ability to debug Windows PowerShell jobs
- Nerk transformation module
- OneGet for managing software packages

PowerShellGet for managing Windows PowerShell modules through OneGet  
Performing reps when a COM object is used

The TechNet library has precisely defined these points in detail. Let us have a brief look at some of them.

Windows PowerShell Workflow: Functionality brings the power of the Windows PowerFlow Foundation to Windows PowerShell. You can write workflows in XAML or Windows PowerShell language and run them as you would a cmdlet.

Improvements for existing core Cmdlets and Providers: Windows PowerShell 3 includes new points for current cmdlets, including simplified syntax, and new parameters for cmdlets such as - Computer cmdlets, CSV cmdlets, Get-ChildItem, Get-Command, Get-content, get-content -Hstar, measure-object, security cmdlets, select-object, selection-string, split-path, start-process, T-object, test-connection and add-member.

Remote Module Import and Search: Windows PowerShell extends three modules search and built-in remoting capabilities on remote computers.

Module cmdlets are the functionality to import modules to far-flung computer systems for neighborhood computers using Windows PowerShell remaking.

New CIM session support: allows CIM and WMI to be used to manage non-Windows computer systems through importing commands for remotely running computers to nearby computers.

Auto-complete feature: saves typing time, and reduces your typos.

PowerShell 3 Intellisense: When you hover the mouse pointer on the wavy line, you correct the error in purple and suggest an improvement.

Update-help cmdlet: This fixes many minor errors or disturbing typos in the underlying documentation.

Enhanced console host experience: Built-in changes to the Windows PowerShell console host application are enabled by default in PowerShell 3. In addition, the new "Run with PowerShell" option in File Explorer lets you run scripts in unrestricted sessions via right-click.

RunAs and Shared Host Support: RunAs features, designed for Windows PowerShell workflows, allow clients of a session configuration to create classes that run with the permission of a shared person account. The shared host feature, on the other hand, allows a pair of users on a pair of computers to connect to a workflow session and carefully tracks the development of a workflow.

Special character handling improvements: A fast lap round Windows PowerShell point to improve the software's ability to interpret and efficiently manage three specific characters, which handle one of a type characters in the

path The literal path parameter is valid on almost all cmdlets with a path parameter, such as the new update-help and save-help cmdlets.

it's the strategies that are jogging on the neighborhood computer.

Syntax

PowerShell

Copy

- Get-Process
- [[-Name] <string[]>]
- [-Module]
- [-FileVersionInfo]
- [<commonparameters>]
- PowerShell

Copy

- Get-Process
- [[-Name] <string[]>]
- [-IncludeUserName]
- [<commonparameters>]
- PowerShell

Copy

- Get-Process
- -Id <int32[]>
- [-IncludeUserName]
- [<commonparameters>]
- PowerShell

Copy

- Get-Process
- -Id <int32[]>
- [-Module]
- [-FileVersionInfo]
- [<commonparameters>]

- PowerShell

## Copy

- Get-Process
- -InputObject <process[]>
- [-Module]
- [-FileVersionInfo]
- [<commonparameters>]
- PowerShell

## Copy

- Get-Process
- -InputObject <process[]>
- [-IncludeUserName]
- [<commonparameters>]

## Description

The Get-Process cmdlet gets the procedures on a neighborhood computer.

Without parameters, this cmdlet receives all of the procedures on the nearby computer. You can also specify a precise technique by means of technique name or technique ID (PID) or pass by a procedure object through the pipeline to this cmdlet.

By default, this cmdlet returns a procedure object that has particular facts about the manner and helps methods that let you start and stop the process. You can also use the parameters of the Get-Process cmdlet to get file version statistics for the application that runs in the technique and to get the modules that the method loaded.

## Examples

Example 1: Get a list of all active procedures on the neighborhood computer

PowerShell

## Copy

### Get-Process

This command gets a listing of all lively approaches jogging on the local

computer. For a definition of each column, see the Notes section.

Example 2: Get all on hand facts about one or greater processes

PowerShell

**Copy**

**Get-Process win word, explorer | Format-List \***

This command gets all reachable facts about the Winword and Explorer methods on the computer. It uses the Name parameter to specify the processes, however, it omits the optional parameter name. The pipeline operator | passes the records to the Format-List cmdlet, which shows all on hand properties \* of the Winword and Explorer method objects.

You can additionally perceive the processes by using their procedure IDs. For instance, Get-Process -Id 664, 2060.

Example 3: Get all tactics with a working set higher than a particular size

PowerShell

**Copy**

**Get-Process | Where-Object {\$\_.WorkingSet -gt 20000000}**

This command gets all processes that have a working set increased than 20 MB. It makes use of the Get-Process cmdlet to get all jogging processes. The pipeline operator | passes the technique objects to the Where-Object cmdlet, which selects only the object with a fee larger than 20,000,000 bytes for the working set property.

WorkingSet is one of many properties of procedure objects. To see all of the properties, type Get-Process | Get-Member. By default, the values of all residences are in bytes, even although the default display lists them in kilobytes and megabytes.

Example 4: List approaches on the laptop in businesses based totally on priority

PowerShell

Copy

**\$A = Get-Process**

**\$A | Get-Process | Format-Table -View priority**

These instructions listing the processes on the pc in businesses based on their priority class. The first command receives all the strategies on the pc and then shops them in the \$A variable.

The 2d command pipes the Process object stored in the \$A variable to the Get-Process cmdlet, then to the Format-Table cmdlet, which formats the processes by using the usage of the Priority view.

The Priority view and other views are described in the PS1XML format files in the PowerShell domestic listing (\$pshome).

Example 5: Add a property to the widespread Get-Process output display

PowerShell

Copy

- `Get-Process pwsh | Format-Table ``
- `@ {Label = "NPM(K)"; Expression = {[int]($_.NPM / 1024)}},`
- `@ {Label = "PM(K)"; Expression = {[int]($_.PM / 1024)}},`
- `@ {Label = "WS(K)"; Expression = {[int]($_.WS / 1024)}},`
- `@ {Label = "VM(M)"; Expression = {[int]($_.VM / 1MB)}},`
- `@ {Label = "CPU(s)"; Expression = {if ($_.CPU)`  
`{$_ .CPU.ToString("N")}}},`



```

Windows PowerShell

-----
This PowerShell module allows you to connect to Exchange Online service.
To connect, use: Connect-EXOPSSession -UserPrincipalName <your UPN>
This PowerShell module allows you to connect Exchange Online Protection and Security & Compliance Ce
nter services also.
To connect, use: Connect-IPPSSession -UserPrincipalName <your UPN>

To get additional information, use: Get-Help Connect-EXOPSSession, or Get-Help Connect-IPPSSession
-----

PS C:\Users\Jeff> Connect-EXOPSSession
WARNING: The names of some imported commands from the module 'tmp_meakisqt.iz1' include unapproved
verbs that might make them less discoverable. To find the commands with unapproved verbs, run the
Import-Module command again with the Verbose parameter. For a list of approved verbs, type
Get-Verb.
PS C:\Users\Jeff> Connect-MsolService
PS C:\Users\Jeff> Connect-AzureAD

Account          Environment TenantId                               TenantDomain AccountType
-----
jeff@jdskspe.net AzureCloud  dcbc9ae9-6053-4a51-be69-83a268eb8cc6 jdskspe.net  User

PS C:\Users\Jeff> Import-Module SkypeOnlineConnector
PS C:\Users\Jeff> $skype = New-CsOnlineSession
Please enter the user principal name (ex. User@Domain.Com): jeff@jdskspe.net
PS C:\Users\Jeff> Import-PSSession $skype

ModuleType Version      Name                               ExportedCommands
-----
Script      1.0          tmp_pykondmg.ij0                  {Clear-CsOnlineTelephoneNumberReservat...

PS C:\Users\Jeff>

```

Id, Machine Name, Process Name -Auto Size

NPM(K) PM(K) WS(K) VM(M) CPU(s) Id MachineName ProcessName

-----

- 6 23500 31340 142 1.70 1980 . pwsh
- 6 23500 31348 142 2.75 4016 . pwsh
- 27 54572 54520 576 5.52 4428 . pwsh

This example gives a Format-Table command that provides the MachineName property to the general Get-Process output display.

```

</commonparameters></process[]></commonparameters></process[]>
</commonparameters></int32[]></commonparameters></int32[]>
</commonparameters></string[]></commonparameters></string[]>

```

## Example 6: Get model data for a process

### PowerShell

Copy

```
Get-Process push -FileVersionInfo
```

ProductVersion	FileVersion	FileName
----------------	-------------	----------

6.1.2	6.1.2	C:\Program Files\PowerShell\6\pwsh.exe
-------	-------	--

This command makes use of the FileVersionInfo parameter to get the model records for the pwsh.exe file that is the principal module for the PowerShell process.

To run this command with procedures that you do no longer own on Windows Vista and later versions of Windows, you should open PowerShell with the Run as administrator option.

## Example 7: Get modules loaded with the precise process

### PowerShell

Copy

```
Get-Process SQL* -Module
```

This command makes use of the Module parameter to get the modules that have been loaded via the process. This command receives the modules for the approaches that have names that begin with SQL.

To run this command on Windows Vista and later versions of Windows with strategies that you do not own, you need to start PowerShell with the Run as administrator option.

## Example 8: Find the proprietor of a process

### PowerShell

Copy

```
Get-Process push -IncludeUserName
```

Handles	WS(K)	CPU(s)	Id	UserName	ProcessName
---------	-------	--------	----	----------	-------------

782	132080	2.08	2188	DOMAIN01\user01	push
-----	--------	------	------	-----------------	------

This command indicates how to find the owner of a process. On Windows, the IncludeUserName parameter requires increased consumer rights (Run as Administrator). The output exhibits that the owner is Domain01\user01.

Example 9: Use an automated variable to discover the method of hosting the current session

PowerShell

Copy

Get-Process push

NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	--	--	-----
eighty three	96.21	105.95	4.33	1192	10	pwsh
79	83.81	117.61	2.16	10580	10	pwsh

Get-Process -Id \$PID

NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	--	--	-----
83	96.21	77.53	4.39	1192	10	push

These commands show how to use the \$PID computerized variable to discover the system that is hosting the cutting-edge PowerShell session. You can use this approach to distinguish the host process from other PowerShell tactics that you would possibly want to give up or close.

The first command gets all of the PowerShell procedures in the cutting-edge session.

The 2nd command gets the PowerShell procedure that is hosting the modern session.

Example 10: Get all approaches that have an important window title and show them in a table

PowerShell

Copy

Get-Process | Where-Object {\$\_.mainWindowTitle} | Format-Table Id, Name, main window title -AutoSize

This command receives all the strategies that have a fundamental window

title, and it displays them in a desk with the procedure ID and the procedure name.

The main window title property is just one of many useful properties of the Process object that Get-Process returns. To view all of the properties, pipe the consequences of a Get-Process command to the Get-Member cmdlet `Get-Process | Get-Member`.

#### Parameters

##### `-FileVersionInfo`

Indicates that this cmdlet gets the file version records for the application that runs in the process.

On Windows Vista and later versions of Windows, you need to open PowerShell with the Run as administrator option to use this parameter on methods that you do not own.

To get file version records for a system on a far off computer, use the `Invoke-Command` cmdlet.

Using this parameter is equal to getting the `MainModule.FileVersionInfo` property of each technique object. When you use this parameter, `Get-Process` returns a `FileVersionInfo` object `System.Diagnostics.FileVersionInfo`, not a procedure object. So, you can't pipe the output of the command to a cmdlet that expects a system object, such as `Stop-Process`.

Type: `SwitchParameter`

- Aliases: `FV`, `FVI`
- Position: `Named`
- Default value: `None`
- Accept pipeline input: `False`
- Accept wildcard characters: `False`

##### `-Id`

Specifies one or more methods via manner ID (PID). To specify multiple IDs, use commas to separate the IDs. To find the PID of a process, kind `Get-Process`.

Type: `Int32[]`

- Aliases: PID
- Position: Named
- Default value: None
- Accept pipeline input: True (ByPropertyName)
- Accept wildcard characters: False

#### -IncludeUserName

It indicates that the UserName value of the Process object is again with the outcomes of the command.

- Type: SwitchParameter
- Position: Named
- Default value: None
- Accept pipeline input: False
- Accept wildcard characters: False
- -InputObject

Specifies one or greater system objects. Enter a variable that incorporates the objects, or kinda command or expression that gets the objects.

- Type: Process[]
- Position: Named
- Default value: None
- Accept pipeline input: True (ByValue)
- Accept wildcard characters: False

#### -Module

Indicates that this cmdlet receives the modules that have been loaded by using the processes.

On Windows Vista and later versions of Windows, you ought to open PowerShell with the Run as administrator alternative to using this parameter on techniques that you do not own.

To get the modules that have been loaded by a system on a remote computer, use the Invoke-Command cmdlet.

This parameter is equal to getting the Modules property of every system object. When you use this parameter, this cmdlet returns a Process Module Object System Diagnostics Process Module, now not a system object. So,

you can't pipe the output of the command to a cmdlet that expects a procedure object, such as Stop-Process.

When you use each Module and File Version Info parameters in the identical command, this cmdlet returns a File Version Info object with data about the file version of all modules.

# Fundamental information of PowerShell

## 1: PS1 files

A PowerShell script is certainly nothing more than an easy text file. The file consists of a series of PowerShell commands, with each command appearing on a separate line. For a textual content file to be dealt with as a PowerShell script, its filename needs to use the PS1 extension.

## 2: execution permissions

To prevent the execution of malicious scripts, PowerShell enforces an execution policy. By default, the execution policy is restricted, which the PowerShell script will not run. You can define a modern execution policy using the following cmdlet:

### Get-Execution Policy

The execution policies you can use are:

#### **Restricted - Scripts cannot run.**

Remote Sign - regionally created scripts will run, but will no longer be downloaded from the Internet (unless they are digitally signed through a trusted publisher).

All Signed - Scripts will only run if they are signed through a trusted publisher.

Unrestricted - Scripts will run regardless of where they have come from and signed or not.

## 3: running script

For years now, if you wanted to run an executable file from the command line, the practice was to navigate to the path of the file and then type the title of the executable file. However, this age-old method does not work for PowerShell scripts.

If you prefer to execute a PowerShell script, you will usually have to type the complete direction with the file name. For example, to run a script called SCRIPT.PS1, you can type:

### **C:\Script\Script.ps1**

The major exception is that you can actually tell a script by typing its identify whether the folder containing the script is in the path of your system. If you are already in a folder with scripts, then there is a shortcut you can use. In

such a case, instead of typing the full path of the script, you can enter. \\_ And script name. For example, you can type:

```
. \ Script.ps1
```

#### **4: pipelining**

Pipelining is the term for feeding the output of one command to another command. It approves the 2D command to act on the received input. To pipeline commands (or cmdlets), of course, separate them with the pipe image (!).

To help you understand how pipelining works, imagine that you want to list the methods that are running on the server and type that list using the process ID number. You can get a list of methods using the get-process cmdlet, although the list will not be serialized. However, if you pipeline the output of the cmdlet to the Sort-Object ID command, the listing will be sorted. The string of instructions used looks like this:

**Go-process | Sort-object id**

#### **5: variable**

Although you can use pipelining to feed the output of one command to another command, my own pipelining might not get the job done every now and then. When you pipeline the output of a command to every other command, that output is used immediately. Sometimes, you additionally want to save the output for a time so that you can use (or reuse) it later. This is where the variables come into play.

It is simple to assume a variable as a repository to store the value, but in PowerShell, a variable can purchase the full output of a command. For example, consider the desire to store listings of ways to run on the server as a variable. To do this, you should use this line of code:

```
$ a = get-process
```

Here, the variable name is \$ a. If you choose to use a variable, state it with certainty. For example, typing \$ prints the contents of the variable on the screen.

You can assign a variable to the remaining output of several instructions that are pipelined together. Include instructions with parentheses only. For example, to serialize jogging processes through the process ID and then assign the output to a variable, you can use this command:



```
$ a = (Get-Process | Sort-Object ID)
```

6: @ symbol

Using the @ symbol, you can convert the contents of the list into an array.

For example, take the following line of code, which creates a variable named \$ Procs that contains strains of text (an array):

```
$ procs = @ {name = "Explorer", "svc host"}
```

You can also use the @ image when the variable is used, to ensure that it is dealt with as an array from a single value as an option. For example, the line of code will run a gate-process cmdlet as opposed to a variable defined a moment ago. When doing so, Windows will display all technologies via Windows Explorer and Seacoast. Notice how the @ symbol is being used in front of the variable symbol, rather than the greenback signal we usually use:

```
Get-Process @process
```

## 7: division

The cut-up operator splits a textual content string based on a fully named personality. For example, trust that you like to waste a sentence in an array consisting of every character word in a sentence. You should do this through the use of a command like this:

```
"This is a test" -Split ""
```

The end result would look like this:

## 8: join

Just as a breakup can cut a text string into several pieces, a join operator can combine multiple blocks of text into one. For example, this line will create a text content string consisting of my first name and last name:

```
"Brian", "Posey" -Join ""
```

The area between quotation marks at the top of the command tells Windows to insert a field between textual content strings.

## 9: breakpoint

If a script creates a bug, running a newly created PowerShell script may result in unexpected penalties. One way to shield yourself is to insert breakpoints at strategic locations within your script. In this way, you can ensure that the script is working as before as compared to working perfectly.

The best way to insert a breakpoint is through the line number. For example, to insert a breakpoint on the tenth line of a script, you should use a command like this:

```
New-PS Breakpoint -Script C:\Scripts\Script.ps1 -Line 10
```

You can also bind a breakpoint to a variable. So if you desired your script to destroy \$ return content at any time, you should use a command like this:

```
New-PS Breakpoint -Script C:\scripts\Script.ps1 -variables a
```

Note that I did not include a dollar sign after the variable name.

There are a variety of actions that you can use with PS Breakpoint including New, Get, Enable, Disable, and Remove.

# Functions, switches, and looping structures of PowerShell

If you have been working with different programming languages, you will quickly feel at home with PowerShell's a range of loop types. Their performance and syntax are based totally on loops in C and derived languages such as JavaScript. Thus, the loop circumstance is commonly enclosed in parentheses and the loop body in braces. However, the conditional operators that you require for the loop condition need getting used to. Apart from that, PowerShell's key phrases are case-insensitive, so you can use it for, For, or FOR.

## **Loops for exclusive purposes** ^

The distinctive loop types cowl all thinkable utility scenarios. For instance, PowerShell supports pretest and posttest loops that enable you to consider the boolean situation either at the start or at the top of the loop body. If you want a loop counter that you want to initialize and increment in the loop condition, you can work with a for a loop. However, if you simply have to test a positive condition, you can make use of a while loop.

For with built-in loop counter ^

PowerShell for loop doesn't deliver any awesome surprises and behaves as in different programming languages.

```
for ($i=1; $i -le 10; $i++) {$i,"`n"}
```

1

```
for ($i=1; $i -le 10; $i++) {$i,"`n"}
```

This easy instance initializes the loop counter \$i with the fee 1. It then increments the counter by using one with each iteration until it reaches 10, which is the circumstance that quits the loop.

## **While loop** ^

The while loop works simply as you would possibly understand it from other languages. It is called a pretest loop because the directions in the loop body are now not executed, even once, if the loop situation doesn't match. In contrast to a for loop, the circumstance can only contain a boolean expression. If you desire to use a loop counter, you have to initialize it before the whilst announcement and increment or decrement it in the loop body. The

following snippet demonstrates how you ought to use whilst for a simple menu system. With each iteration, the user is asked to press a key to pick a menu item. Entering “Q” terminates the loop.

The whilst loop exams the circumstance earlier than the first iteration. As you can see in the example, you can also call cmdlets and assign values in the loop condition.

In addition, PowerShell helps the posttest loops do-while and do-until. In both cases, the instructions in the loop physique are performed at least as soon as due to the fact the situation is at the cease of the loop.

From a syntactical point of view, do-while and do-until are identical. The distinction is logical in nature. do-while continues to run as long as the situation is authentic and terminates when the condition is no longer fulfilled; do-until works the other way around: it quits when the condition takes the cost TRUE. The universal structure appears like this:

while/until(condition)

Basically, do-until and do-while can replace each different if you negate the condition.

Loop manage with the wreck and proceed ^

PowerShell offers additional methods to manage loops. The instructions spoil and continue are recognized from other languages and are rarely required in well-structured scripts. Before you work with these language elements, you ought to attempt to avoid them with the aid of the use of any other algorithm. Code that makes use of destroying and proceed is hard to read and can have unwanted side effects.

The spoil command can be used now not solely in all loop kinds however also in switch blocks. The command usually has the identical effect—that is, the block will be terminated and manipulate is transferred to the father or mother block. The wreck command is similar to exit for and exit does in VBScript.

Using wreck in loops quits the iteration, and the following code outside the loop will be executed. For nested loops, this would be the greater order for-, while-, for each-, or do block. The sole state of affairs place the utilization of

wreck in all likelihood makes the experience is within an if announcement due to the fact otherwise the loop wouldn't be iterated a 2nd time.

Break with goto ^

An area of expertise of PowerShell's damage command is that you can mix it with a label. This works comparable to the notorious GoTo in Basic. The sole exception is that the goals have to be in the identical line with a for-, while-, for each-, or do- command.

while(condition){

- loop physique instructions
- if(condition) {break: DoLoop}
- loop physique instructions
- }
- :DoLoop do{

loop body instructions

}

until(condition)

1

2

3

4

5

6

7

8

9

while(condition) {

- loop body instructions
- if(condition) {break: DoLoop}
- loop body instructions

}

: Do Loop do {

loop physique instructions

```
}
```

### **until(condition)**

In this construct, the application would bounce to the do loop when the if condition within the whilst block applies.

### **Continue command ^**

As with the wreck command, the proceed command terminates the execution of the loop body. In contrast to breaking, proceed doesn't direct the software to go with the flow to the code after the loop; instead, it resumes the execution of the loop in the next iteration. Here, again, use proceed sparingly!

Even although PowerShell comes with the entire loop weaponry of compiled languages, you will hardly ever make the most it's whole workable due to the fact scripts typically have an incredibly simple structure.

Clearly, the most everyday use of loops, especially in interactive mode, is the iteration through the factors of an array because many cmdlets return this statistics type.

Implicit new release thru array elements

Because of this, Microsoft simplified such operations and permits admins to pick between special methods. Starting with PowerShell 3.0, the easiest and most dependent one is implicit for each that, in some cases, approves you to avoid this keyword. The example below demonstrates how you can retrieve a sure property of all array elements without the usage of a loop or Select-Object.

**(Get-AD User -Filter \*). Surname**

**1**

**(Get-AD User -Filter \*). Surname**

This name of Get-AD User returns all Active Directory user objects as an array. The above command displays the remaining identity of each element.

In this PowerShell tutorial, we show you how to use the for loop, For Each-Object loop, and the While, Do-While and Do-Until loops.

PowerShell loops, at their most basic, truly repeat the identical set of commands a set wide variety of times. Ideal for performing regular actions for a set length of time or a certain variety of records, loops can simplify your

scripts in a massive way. PowerShell in specific facets a number of cmdlets -- incredibly those that begin with the verb Get -- which return objects containing massive numbers of similar data.

There are numerous kinds of loops reachable in PowerShell, and in many cases extra than one loop approach can be used effectively. At times figuring out the most environment-friendly loop kind is required, either from an overall performance or code readability perspective.

These parameters can be distinct either by using the -Input Object and -Process parameter names or by using piping the object to the for Each-Object cmdlet and set the script block as the first parameter. To illustrate this fundamental syntax, the following instance suggests techniques of the use of ForEach-Object to loop via the contents of a user's Documents folder:

```
$my Documents = Get-Child Item $env:USERPROFILEDocuments -File  
$myDocuments | ForEach-Object {$_.FullName}  
ForEach-Object -InputObject $myDocuments -Process {$_.FullName}
```

Insure eventualities it may additionally be beneficial to perform one or more movements simply earlier than or just after the loop is performed. The -Begin and -End parameters can be used to outline script blocks to execute simply earlier than or after the contents of the -Process script block. This can be used to set or regulate a variable earlier than or after the execution of the loop.

ForEach-Object has aliases, ForEach, and %, and additionally helps shorthand syntax establishing in PowerShell 3.0. The following three examples are the same in function.

```
Get-WMIObject Win32_LogicalDisk | ForEach-Object {$_.FreeSpace}  
Get-WMIObject Win32_LogicalDisk | ForEach {$_.FreeSpace}  
Get-WMIObject Win32_LogicalDisk | percent FreeSpace
```

For loops are normally used to iterate via a set of commands a designated number of times, both to step through an array or object, or just to repeat the same block of code as needed. A loop is built by means of putting the value of a variable when entering the loop, the circumstance on which the loop be terminated, and a motion to be performed against that variable every time via the loop.

The following instance shows a primary For loop used to create a multiplication table:

```
For ($i=0; $i -le 10; $i++) {  
    "10 * $i = " + (10 * $i)  
}
```

For loops can be used to step through array values with the aid of putting the preliminary value to the preliminary index of the array and incrementally growing the count till the array size is met. The array index is unique with the aid of setting the incremented variable in square brackets right away following the variable name, as shown in the following example:

```
$colors = @("Red","Orange","Yellow","Green","Blue","Indigo","Violet")  
For ($i=0; $i -lt $colors.Length; $i++) {  
    $colors[$i]  
}
```

### While, Do-While, and Do-Until Loops

A third type of loop that PowerShell support includes putting a situation that either let in the loop to process as long as the condition is proper or till it is met. While and Do-While loops are both used to operate a motion whilst the situation evaluates to \$true, and differ only in their syntax. Do-Until loops have a similar syntax to Do-While, however, cease processing as soon as the situation statement is met.

Do-While and Do-Until loops each begin with the Do key-word prefacing a script block, accompanied with the aid of the condition keyword (While or Until) and the condition. As an example the following loops characteristic identically, only the situation is reversed:

```
$i=1  
Do {  
    $i  
    $i++  
}  
While ($i -le 10)  
  
$i=1
```



```
Do {  
    $i  
    $i++  
}  
Until ($i -gt 10)
```

While loops perform identically to Do-While loops, solely the syntax is altered slightly. While loops use only the While keyword, observed through the condition, and subsequently the script block. This loop is the same in feature to the previous examples, and uses the same situation as the Do-While loop:

```
$i=1  
While ($i -le 10)  
{  
    $i  
    $i++  
}
```

Any of these three loop sorts -- Do-While, Do-Until, and While loops -- can also be used to loop indefinitely; While and Do-While loops with the situation set to \$true and Do-Until with the condition set to \$false.

In some situations, you may additionally need to exit a loop early based totally on something other than the loop condition. In this case, the Break keyword can be invoked in order to exit out of the loop. This last example suggests the identical functionality, but makes use of an infinite loop and the Break keyword to exit out at the excellent time:

```
$i=1  
While ($true)  
{  
    $i  
    $i++  
    if ($i -gt 10) {  
        Break  
    }  
}
```

Instead of copying and pasting the same code over and over again, create a

PowerShell function.

When first beginning out writing PowerShell scripts you're not always worried about matters such as modularity, reusability and "best practices." You're simply getting your feet wet. However, as time goes on, you will quickly understand that you commence repeating yourself in code.

You'll observe that want to do the equal component over and over continues getting greater. You should simply replica and paste that same code, however, it is not sustainable. Instead, why not create small "building blocks" in code so that they can be reused? It's time to begin developing PowerShell functions.

A characteristic in PowerShell is a grouping of code that has a non-compulsory input and output. It's a way of accumulating up a bunch of code to function one or many one-of-a-kind times with the aid of simply pointing to it as an alternative of duplicating that code repeatedly.

Basic functions

There are kinds of functions in PowerShell. We have a "basic" function and a superior function. "Basic" features are the easiest structure of a function that can be created. They don't have any of the fancy built-in abilities that superior functions do. It is created or declared by means of the usage of the function announcement accompanied by a set of curly braces.

function Do-Something.

The above is technically a characteristic that can then be known as with the aid of invoking Do-Something however as you will find it doesn't do very much. This is because we haven't covered any code inside to run. Let's add some simple code interior to ensure it does something. To demonstrate, I'll use the Write-Host command that writes text to the PowerShell console.

```
function Do-Something {  
    Write-Host 'I did something!'  
}
```

```
PS > Do-Something  
I did something!
```

Now you can see above that when invoked; it runs the code inside the function. What if we might like to skip something into the code inside of the character when it is running? We can create one or extra parameters inside of a parameter block.

```
function Do-Something {  
    param( $String )  
    Write-Host "I did something -- $String!"}
```

```
PS > Do-Something -String 'thing'  
I did something -- thing!
```

Notice that I'm certain the String parameter through including a sprint followed by way of the parameter title accompanied through the cost proper after I have known as Do-Something. This is the basics of passing parameters to functions.

### **Advanced functions**

Basic functions work, however, most of the time you may find yourself developing the superior function. Advanced features are features that encompass all of the performance as fundamental features however additionally come with some built-in points that primary functions do not. For example, PowerShell has a concept of streams called Error, Warning, Verbose, etc. These streams are fundamental incorrectly exhibiting output to users. Basic functions do now not inherently apprehend these streams.

Let's say we desire to show an error message if something happens inside of the function. However, we also want the ability to cover this error message for some motive at solely positive times. With a simple function, it would be kludgy to do this. However, with an advanced function, that performance is built properly. Advanced functions, through default, already have parameters even if you don't consist of them like Verbose, ErrorAction, WarningVariable and so on. These can be leveraged in some extraordinary ways. In our error example, let's say I've modified our function to be "advanced" with the aid of the use of the [CmdletBinding()] keyword.

```
function Do-Something {  
    [CmdletBinding()]  
    param( $String )
```

```
Write-Error -Message 'Danger, Will Robinson!'
}
```

```
PS> Do-Something
Do-Something: Danger, Will Robinson!
At line:1 char:1
+ Do-Something
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Write-Error],
WriteErrorException
+ Fully Qualified Error Id :
```

When this is run, you will right now see a crimson text which indicates it came from the error stream. Let's now silence this error.

A function is a listing of PowerShell statements that has a title that you assign. When you run a function, you kind the function name. The statements in the list run as if you had typed them at the command prompt.

Functions can be as simple as:

PowerShell

Copy

```
function Get-PowerShellProcess { Get-Process PowerShell }
```

A function can additionally be as complicated as a cmdlet or a utility program.

Like cmdlets, features can have parameters. The parameters can be named, positional, switch, or dynamic parameters. Function parameters can be studied from the command line or from the pipeline.

Functions can return values that can be displayed, assigned to variables, or surpassed to other features or cmdlets. You can also specify a return value for the usage of the return keyword. The return keyword does not have an effect on or suppress different output lower back from your function. However, the return key-word exits the function at that line. For more information, see [about\\_Return](#).

The function's statement listing can include distinctive sorts of statement lists

with the key phrases Begin, Process, and End. These assertion lists cope with entering the pipeline differently.

A filter is a distinctive kind of function that uses the Filter keyword.

Functions can also act like cmdlets. You can create a function that works simply like a cmdlet without the use of C# programming. For more information, see [about\\_Functions\\_Advanced](#).

### Syntax

The following is the syntax for a function:

### Copy

```
function [<scope:>]<name> [([type]$parameter1[, [type]$parameter2))]  
{
```

- param([type]\$parameter1 [, [type]\$parameter2))
- dynamicparam {<statement list="">}
- begin {<statement list="">}
- process {<statement list="">}
- cease {<statement list="">}
- }

A characteristic consists of the following items:

- A Function keyword
- A scope (optional)
- A title that you select
- Any variety of named parameters (optional)
- One or extra PowerShell commands enclosed in braces {}

For greater facts about the Dynamicparam keyword and dynamic parameters in functions, see [about\\_Functions\\_Advanced\\_Parameters](#).

### Simple Functions

Functions do not have to be problematic to be useful. The simplest features have the following format:

### Copy

function <function-name> {statements}

For example, the following characteristic starts off evolved PowerShell with the Run as Administrator option.

PowerShell

```
</function-name></statement></statement></statement></statement>  
</name></scope:>
```

To use the function, type: Start-PSAdmin

To add statements to the function, type every assertion on a separate line, or use a semicolon; to separate the statements.

For example, the following feature finds all .jpg archives in the present-day user's directories that were changed after the start date.

PowerShell

Copy

```
function Get-NewPix  
{  
$start = Get-Date -Month 1 -Day 1 -Year 2010  
$allpix = Get-ChildItem -Path $env:UserProfile\*.jpg -Recurse  
$allpix | Where-Object {$_.LastWriteTime -gt $start}  
}
```

You can create a toolbox of beneficial small functions. Add these functions to your PowerShell profile, as described in [about\\_Profiles](#) and later in this topic.

## Function Names

You can assign any identity to a function, however, functions that you share with others observe the naming rules that have been mounted for all PowerShell commands.

Functions names should consist of a verb-noun pair in which the verb identifies the motion that the function performs and the noun identifies the item on which the cmdlet performs its action.

Functions need to use the standard verbs that have been authorized for all PowerShell commands. These verbs help us to hold our command names

simple, consistent, and convenient for customers to understand.

For more information about the well-known PowerShell verbs, see [Approved Verbs in the Microsoft Docs](#).

## Functions with Parameters

You can use parameters with functions, which include named parameters, positional parameters, switch parameters, and dynamic parameters. For more statistics about dynamic parameters in functions, see [about\\_Functions\\_Advanced\\_Parameters](#).

## Named Parameters

You can define any range of named parameters. You can consist of a default cost for named parameters, as described later in this topic.

You can define parameters inside the braces the usage of the Param keyword, as proven in the following pattern syntax:

Copy

```
function <name> {  
    param ([type]$parameter1,[type]$parameter2)  
    <statement list="">  
}
```

You can additionally define parameters backyard the braces without the Param keyword, as proven in the following sample syntax:

PowerShell

Copy

```
function <name> [([type]$parameter1,[type]$parameter2)] {  
    <statement list="">  
}
```

Below is an example of this alternative syntax.

PowerShell

Copy

```
Function Add-Numbers ($one, $) {  
    $one + $  
}
```

While the first approach is preferred, there is no distinction between these methods.

When you run the function, the value you grant for a parameter is assigned to a variable that consists of the parameter name. The cost of that variable can be used in the function.

The following example is a characteristic called Get-small files. This characteristic has a \$size parameter. The feature shows all the documents that are smaller than the fee of the \$size parameter, and it excludes directories:

PowerShell

```
Copy
function Get-SmallFiles {
Param($Size)
Get-ChildItem $HOME | Where-Object {
    $_. Length -lt $Size -and !$_.PSIsContainer
}
}
```

In the function, you can use the \$size variable, which is the identify described for the parameter.

To use this function, type the following command:

PowerShell

```
Copy
Get-SmallFiles -Size 50
```

You can additionally enter a price for a named parameter except for the parameter name. For example, the following command gives the same end result as a command that names the Size parameter:

PowerShell

```
Copy
Get-SmallFiles 50
```

To outline a default fee for a parameter, kind an equal sign and the fee after the parameter name, as proven in the following variant of the Get-SmallFiles example:



## PowerShell

### Copy

```
function Get-SmallFiles ($Size = 100) {  
  Get-ChildItem $HOME | Where-Object {  
    $_. Length -lt $Size -and !$_.PSIsContainer  
  }  
}
```

If your kind Get-SmallFiles barring a value, the function assigns 100 to \$size. If you provide a value, the feature uses that value.

Optionally, you can supply a short assist string that describes the default cost of your parameter, by means of adding the PSDDefaultValue attribute to the description of your parameter and specifying the Help property of PSDDefaultValue. To provide a help string that describes the default cost (100) of the Size parameter in the Get-SmallFiles function, add the PSDDefaultValue attribute as proven in the following example.

## PowerShell

### Copy

```
function Get-SmallFiles {  
  param (  
    [PSDefaultValue (Help = '100')]  
    $size = 100  
  )  
}
```

For more information about the PSDDefaultValue attribute class, see PSDDefaultValue Attribute Members.

## Positional Parameters

A positional parameter is a parameter except for a parameter name. PowerShell uses the parameter price order to partner each parameter value with a parameter in the function.

When you use positional parameters, type one or extra values after the function name. Positional parameter values are assigned to the \$args array variable. The cost that follows the feature name is assigned to the first role in the \$args array, \$args[0].

The following Get-Extension feature provides the .txt file identify extension to a file title that you supply:

PowerShell

Copy

```
function Get-Extension {  
$name = $args[0] + ".txt"  
$name  
}
```

PowerShell

Copy

Get-Extension myTextFile

Output

Copy

myTextFile.txt

Switch Parameters

A swap is a parameter that does not require a value. Instead, you kind the characteristic name followed with the aid of the name of the swap parameter.

To define a swap parameter, specify the kind [switch] earlier than the parameter name, as shown in the following example:</statement></name></statement></name>

PowerShell

Copy

```
function Switch-Item {  
param ([switch]$on)  
if ($on) { "Switch on" }  
else { "Switch off" }  
}
```

When you type the On change parameter after the characteristic name, the feature displays "Switch on". Without the swap parameter, it shows "Switch off".

PowerShell

- Copy

- Switch-Item -on
  - Output
- 
- Copy
  - Switch on
  - PowerShell
- 
- Copy
  - Switch-Item
  - Output
- 
- Copy
  - Switch off
  - You can also assign a Boolean price to a swap when you run the function, as proven in the following example:
- 
- PowerShell
- 
- Copy
  - Switch-Item -on:\$true
  - Output
- 
- Copy
  - Switch on
  - PowerShell
- 
- Copy
  - Switch-Item -on:\$false
  - Output

Copy

Switch off

Using Splatting to Represent Command Parameters

You can use splatting to signify the parameters of a command. This feature is introduced in Windows PowerShell 3.0.

Use this technique in functions that call commands in the session. You do not need to declare or enumerate the command parameters or alternate the feature when command parameters change.

The following pattern feature calls the Get-Command cmdlet. The command uses @Args to signify the parameters of Get-Command.

PowerShell

Copy

```
function Get-My Command { Get-Command @Args }
```

You can use all of the parameters of Get-Command when you call the Get-MyCommand function. The parameters and parameter values are exceeded to the command using @Args.

PowerShell

Copy

```
Get-MyCommand -Name Get-ChildItem
```

Output

Copy

CommandType	Name	ModuleName
-------------	------	------------

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

Cmdlet	Get-ChildItem	Microsoft.PowerShell.Management
--------	---------------	---------------------------------

The @Args characteristic uses the \$Args automatic parameter, which represents undeclared cmdlet parameters and values from closing arguments.

For more information about splatting, see about\_Splatting.

Piping Objects to Functions

Any characteristic can take enter from the pipeline. You can manage how a function methods enter from the pipeline the usage of Begin, Process, and End keywords. The following pattern syntax suggests the three keywords:

Copy

```
function <name> {  
begin {<statement list="">}  
process {<statement list="">}  
end {<statement list="">}  
}
```

The Begin assertion listing runs one time only, at the establishing of the function.

## Important

If your function defines a Begin, Process or End block, all of your code has to dwell inner one of the blocks.

The Process assertion list runs one time for every object in the pipeline. While the Process block is running, each pipeline object is assigned to the `$_` automatic variable, one pipeline object at a time.

After the feature receives all the objects in the pipeline, the End declaration list runs one time. If no Begin, Process, or End keywords are used, all the statements are handled like an End declaration list.

The following feature uses the Process keyword. The feature shows examples from the pipeline:

PowerShell

Copy

```
function Get-Pipeline
```

```
{
```

```
system {"The cost is: $_"}
```

```
}
```

To demonstrate this function, enter a listing of numbers separated by commas, as proven in the following example:

PowerShell

Copy

```
1,2,4 | Get-Pipeline
```

Output

Copy

```
The fee is: 1
```

```
The fee is: 2
```

```
The value is: 4
```

When you use a character in a pipeline, the objects piped to the characters are assigned to the `$input` computerized variable. The characteristic runs statements with the Begin keyword earlier than any objects come from the pipeline. The characteristic runs statements with the End key-word after all the objects have been received from the pipeline.

The following instance shows the \$input automatic variable with Begin and End keywords.

PowerShell

Copy

```
function Get-PipelineBeginEnd
{
start {"Begin: The enter is $input"}
stop {"End: The input is $input" }
}
```

If this feature is run via the use of the pipeline, it displays the following results:

PowerShell

Copy

```
1,2,4 | Get-PipelineBeginEnd
Output
```

Copy

```
Begin: The input is
End: The enter is 1 2 4
```

When the Begin assertion runs, the function does not have enter from the pipeline. The End announcement runs after the function has the values.

If the character has a Process keyword, every object in \$input is eliminated from \$input and assigned to \$\_. The following example has a Process assertion list:

PowerShell

Copy

```
function Get-PipelineInput
{
procedure {"Processing: $_ " }
quit {"End: The enter is: $input" }
}
```

In this example, every object that is piped to the function is despatched to the Process assertion list. The Process statements run on every object, one object

at a time. The \$input computerized variable is empty when the character reaches the End keyword.

PowerShell

Copy

1,2,4 | Get-Pipeline Input

Output

Copy

Processing: 1

Processing: 2

Processing: 4

End: The input is:

For greater information, see Using Enumerators

Filters

A filter is a type of feature that runs on every object in the pipeline. A filter resembles a characteristic with all its statements in a Process block.

The syntax of a filter is as follows:

Copy

```
filter [<scope:>]<name> {<statement list="">}
```

The following filter takes log entries from the pipeline and then displays either the total entry or solely the message element of the entry:

PowerShell

Copy

```
filter Get-ErrorLog ([switch]$message)
```

```
{
```

```
if ($message) { Out-Host -InputObject $_.Message }
```

```
else { $_ }
```

```
}</statement></name></scope:></statement></statement></statement>
```

```
</name>
```

Function Scope

A feature exists in the scope in which it was once created.

If a characteristic is part of a script, the feature is handy to statements within

that script. By default, a character in a script is not handy at the command prompt.

You can specify the scope of a function. For example, the function is brought to the world scope in the following example:

PowerShell

Copy

```
function global:Get-DependentSvs {  
Get-Service | Where-Object {$_. Dependent Services}  
}
```

When a feature is in the world scope, you can use the feature in scripts, functions, and at the command line.

Functions generally create a scope. The objects created in a function, such as variables, exist solely in the characteristic scope.

For more statistics about scope in PowerShell, see [about\\_Scopes](#).

Finding and Managing Functions Using the Function: Drive

All the features and filters in PowerShell are routinely saved in the Function: drive. This drive is exposed by means of the PowerShell Function provider.

When referring to the Function: drive, type a colon after Function, just as you would do when referencing the C or D power of a computer.

The following command displays all the features in the modern session of PowerShell:

PowerShell

Copy

Get-ChildItem function:

The instructions in the feature are saved as a script block in the definition property of the function. For example, to show the instructions in the Help function that comes with PowerShell, type:

PowerShell

Copy

(Get-ChildItem function:help).Definition



You can additionally use the following syntax.

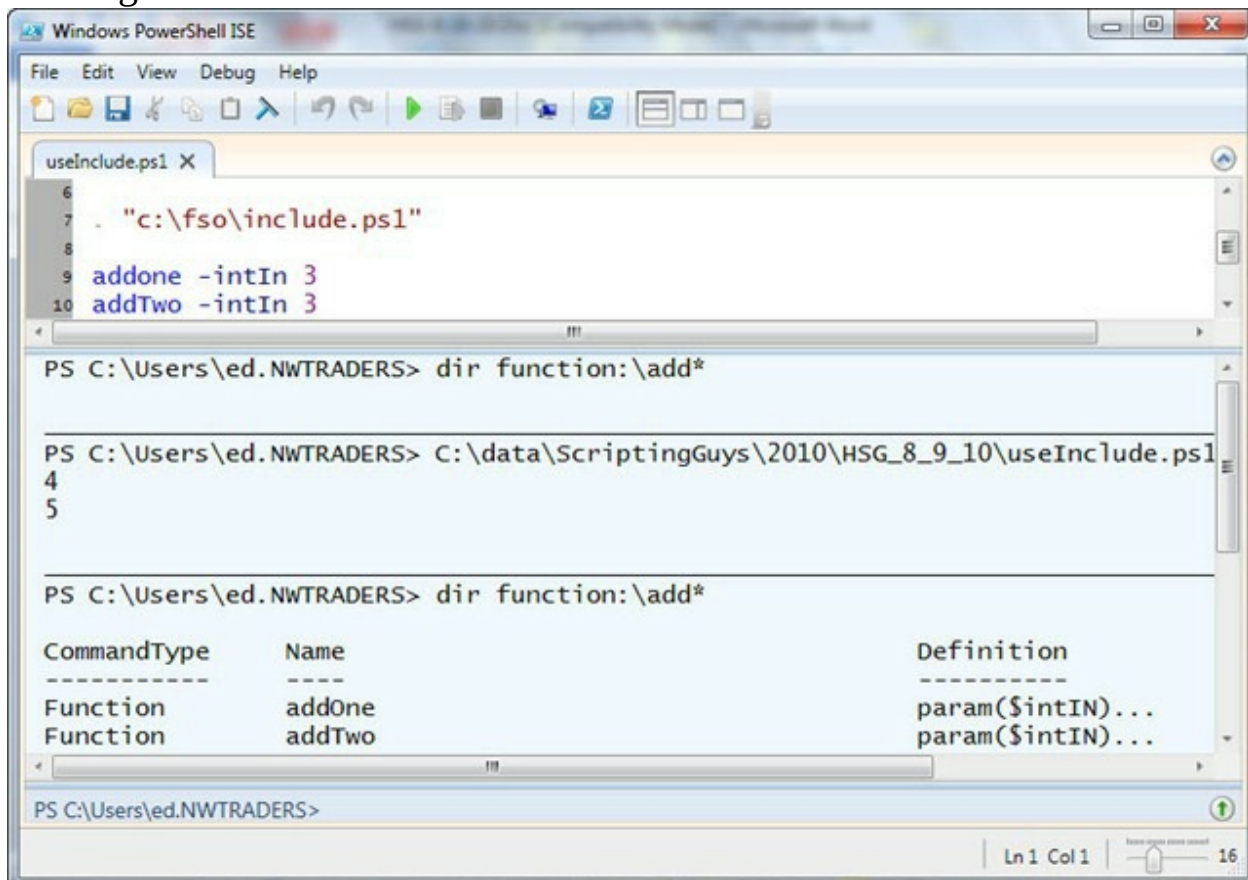
PowerShell

Copy

\$function:help

For greater records about the Function: drive, see the help topic for the Function provider. Type Get-Help Function.

## Reusing Functions in New Sessions



```
useInclude.ps1 X
6
7 . "c:\fso\include.ps1"
8
9 addOne -intIn 3
10 addTwo -intIn 3

PS C:\Users\ed.NWTRADERS> dir function:\add*

PS C:\Users\ed.NWTRADERS> C:\data\ScriptingGuys\2010\HSG_8_9_10\useInclude.ps1
4
5

PS C:\Users\ed.NWTRADERS> dir function:\add*

CommandType      Name              Definition
-----
Function          addOne            param($intIN)...
Function          addTwo            param($intIN)...
```

When you type a function at the PowerShell command prompt, the function turns into a section of the modern-day session. It is accessible until the session ends.

To use your characteristic in all PowerShell sessions, add the characteristic to your PowerShell profile. For extra information about profiles, see [about\\_Profiles](#).

You can also save your character in a PowerShell script file. Type your function in a text file, and then store the file with the ps1 file identify

extension.

For small features and/or when trying out it is additionally feasible to surely kind (or copy and paste) a complete feature at the PowerShell command line. This is not virtually realistic for longer scripts because it fills up lots of your command history. PowerShell does no longer store functions or filters permanently by using the default. So if you shut and reopen PowerShell, the function/filter will no longer be available.

To make a feature permanently available, you can add it to your PowerShell \$Profile it will then be loaded for each and every new PowerShell session and can be known as from a couple of specific scripts without having to be explicitly included.

Dot sourcing a script is very similar to using the PowerShell \$Profile however rather than the usually loaded \$Profile, you keep the function in a script of your deciding on and then load it (by dot sourcing) solely when it is needed:

```
. C:\Scripts\Tools.ps1  
Add-Numbers 5 10
```

Another method of making a characteristic available to a couple of scripts/sessions is to include it as part of a PowerShell Module. When saved in either the machine Module folder or the user Module folder, automated cmdlet discovery (in PowerShell v3 and above) will then make certain that these functions are available.

When you name the characteristic name, the code within the function will run, a feature can take delivery of imported values both as arguments or via the pipeline. If the characteristic returns any values, they can be assigned to variables or handed to different features or cmdlets.

### **Choose a top function name**

The built-in PowerShell alias names will take precedence over a function (or script) with the same name. To keep away from conflicts constantly take a look at if the identifier is already in use with help name. Choosing a true descriptive title for your function will help with this

A nice practice is to use pick out a name in Verb-Noun format, by advantage

of containing a '-' character, this will no longer fighting with any constructed in Aliases. You can generate a list of the authorized PowerShell verbs through strolling the cmdlet Get-Verb

The distinction between a filter function and an ordinary character is the way they cope with items passed via the pipeline:

With an everyday function, pipeline objects are bound to the \$input automated variable, and execution is blocked until all enter is received. The characteristic then starts processing the data.

With a filter function, data is techniques while it is being received, barring ready for all input. A filter receives every object from the pipeline through the \$\_ computerized variable, and the script block is processed for each object.

The param\_list is an optionally available listing of comma-separated parameter names; these might also additionally be preceded via their statistics sorts in brackets. This makes the feature more readable than the use of \$args and also offers you the option to provide default values.

### **Advanced function**

An Advanced PowerShell feature carries the [cmdlet binding ()] attribute (or the Parameter attribute). This provides quite a few competencies such as extra parameter checking, and the potential to without difficulty use Write-Verbose.

A function with cmdlet binding will throw an error if unhandled parameter values appear on the command line.

Advanced PowerShell functions normally include Begin. Process. End blocks for processing the enter data, documentation and auto-help, which include the parameters.

### **Variable Scope**

By default, all variables created in functions are local, they only exist within the function, though they are nonetheless visible if you name a 2d function from inside the first one.

# How to utilize .NET

## **What is .Net?**

.Net is an open-source development framework that actually matches well with more than one platform, languages, and libraries. It is useful for building applications that span a variety of running systems, including Windows and macOS, and systems different from Linux, as well as iOS and Android. It is customizable and was developed by Microsoft to manipulate Windows functions and is available on Github.

NET is not a language, but a framework. Associated programming languages include C #, F #, and Visual Basic (vb.net). The ecosystem includes Visual Studio, an integrated improvement environment, Xamarin, an agency bridging the gap for cross-platform native interfaces, and ASP.net, a developer platform designed to shorten development time. Developers are tools.

## **Learn why .Net**

.Net is a large ecosystem of languages, libraries, and platforms. Businesses that build an employer's choices can use the nick's advanced tools and keep integrity regardless of their users' preferred platforms. Applications built on Android can behave the same for iOS and the same for macOS or Linux. Also, the .Net Framework is used by some of the largest agency groups around.

.Net structures are scalable and flexible, providing faster development and faster iterations to organizations. The framework can take care of this if you want server-aspect programming for desktop purposes or cellular applications.

For business, this skill is the architect of server-side solutions that can withstand vast amounts of information and pivot quickly. This is the perfect execution of employer options for large organizations that cannot operate in full tilt consistently.

## **.Net Courses and Certifications**

EdX, consisting of the introduction of C # from Microsoft and object-oriented programming in C #, offers several options for mastering .NET programming. Expand understanding in large-scale records with data access

in C # and .Net Core.

You can enhance your knowledge of the .Net ecosystem with coaching in Xamarin or use Azure to supply the Net utility with Artificial Intelligence capabilities. You can additionally take a deep dive into MVC programming and use ASP.NET core purge objectives to measure real-world problems.

### **Future of .Net**

Developers are wondering what is new with the framework and if it is really worth it then time can be assured. The improvement of the Net through the open-source model is still going strong. .Net developers are demanding for their ability to build. Business organizations can search for solutions, Internet applications, seed databases for machine learning, and cross-platform solutions. The .Net Foundation continues to innovate, and we predict that it will remain a core framework.

The .NET Framework is a runtime execution environment that manages apps that target the .NET Framework. This repeatedly includes language runtime, which supports administration and other system services, and an adequate range of libraries, enabling programmers to leverage robust, reliable code for all the most important areas of application development. makes.

### **pay attention**

The .NET Framework is the only handheld on Windows structures. You can use .NET Core to run apps on Windows, macOS, and Linux.

### **What is the .NET Framework?**

The .NET Framework is a managed execution environment for Windows that offers a variety of offerings to its running apps. It consists of major components: the common language runtime (CLR), the execution engine that handles jogging apps, and the .NET Framework class library, which reuses the tested library, reusable code that builders Can name from your personal application. The .NET Framework includes the following to introduce strolling apps:

memory management. In many programming languages, programmers are responsible for allocating and releasing memory and dealing with object

lifetimes. In the .NET Framework app, CLR offers these services on behalf of the app.

A common type of system. In common programming languages, primary types are defined using the compiler, which complicates cross-language interoperability. In the .NET Framework, the primary types are described through the .NET Framework type device and are common to all languages that target the .NET Framework.

A tremendous class library. Instead of writing large amounts of code to handle common low-level programming tasks, programmers use the likable libraries and their individuals from the .NET Framework class library without difficulty.

Development Framework and Technologies. The .NET Framework includes libraries for specific areas of app development, such as ASP.NET for web apps, ADO.NET for record access, Windows Communication Foundation for service-oriented apps, and Windows Presentation Foundation for Windows computer apps.

Language differences Language compilers targeting the .NET Framework emit an intermediate code called Common Intermediate Language (CIL), which is compiled at runtime using the common language runtime. With this feature, routines written in one language are in different languages, and programmers focus on growing apps in their favorite languages.

Version compatibility. With rare exceptions, applications developed using a specific version of the .NET Framework run barring modifications on a later version.

Side-by-Side Execution. The .NET Framework helps to address version conflicts by allowing multiple forms of common language runtime on a single computer. This capability can coexist more than one type of apps and that app can run on the model of .NET Framework with which it was once built. The side-by-side execution .NET Framework model applies to organizations 1.0 / 1.1, 2.0 / 3.0 / 3.5 and 4 / 4.5.x / 4.6.x / 4.7.x / 4.8.

Multitargeting. By targeting the .NET standard, developers build classification libraries that work on more than one .NET Framework platform supported through that model of the standard. For example, libraries that

target the .NET Standard 2.0 can be used through applications targeting the .NET Framework 4.6.1, .NET Core 2.0, and UWP 10.0.16299.

### **.NET Framework for Users**

If you do not improve the .NET Framework apps, although you use them, you do not necessarily have to have accurate information about the .NET Framework or its operation. For the most part, the .NET Framework is perfectly clear to users.

### **.NET Framework for Users**

If you do not extend the .NET Framework apps, however, you use them, then you do not think it is necessary to have specific information about the .NET Framework or its operation. For the most part, the .NET Framework is perfectly clear to users.

If you are using the Windows running system, the .NET Framework may already be placed on your computer. In addition, if you set up an app that requires a .NET Framework, the app's setup software can set a specific version of the .NET Framework on your computer. In some cases, you may also see a dialog area that asks you to deploy the .NET Framework. If you just tried to run an app when this dialog container appears and if your computer has Internet access, you can go to a webpage, which allows you to deploy the missing model of the .NET Framework. For more information, see the installation guide.

In general, you should not uninstall the variety of .NET Framework installed on your computer. There are objectives for this:

If the app you use depends on a specific model of the .NET Framework, then if that model is removed, this app may break.

Some versions of the .NET Framework are in-place updates to earlier versions. For example, the .NET Framework 3.5 is an in-place replacement for version 2.0, and the .NET Framework 4.8 is an in-place update to change 4 through 4.7.2. For more information, see the .NET Framework version and dependencies.

On Windows versions prior to Windows 8, if you choose to eliminate the .NET Framework, usually use the control panel's programs and features to

uninstall it. Never manually finish a version of the .NET Framework. On Windows Eight and above, the .NET Framework is a working machine component and cannot be uninstalled independently.

Note that some versions of the .NET Framework can coexist on a PC at the same time. This capability that you do not need to uninstall earlier variants to set a later version.

### **.NET Framework for Developers**

If you are a developer, select any programming language that helps in .NET Framework to build your apps. Because the .NET Framework provides language independence and interoperability, you have interaction with other .NET Framework apps and elements, regardless of the language with which they were developed.

To strengthen the .NET Framework application or components, do the following:

If it is not preinstalled on your working system, then install the model of .NET Framework that will target your app. The most recent manufacturing version is .NET Framework 4.8. It is preinstalled on the Windows 10 May 2019 update and is available for download on earlier changes to the Windows working system. For .NET Framework machine requirements, see System Requirements. For records on how to insert into other forms of the .NET Framework, see the Installation Guide. Additional .NET Framework applications are left out of band, meaning they are launched on an everyday or outdoor rolling foundation of a scheduled release cycle. For data about these packages, see .NET Framework and out-of-band releases.

Select the language or languages supported through the .NET Framework that you want to use to advance your apps. A variety of languages are available from Microsoft, including Visual Basic, C #, F #, and C ++ / CLI. (A programming language that allows you to improve apps for the .NET Framework follows the Common Language Infrastructure (CLI) specification.)

Select and install the development environment to use to build your application and which supports your selected programming language or languages. The Microsoft Built-in Improvement Environment (IDE) for .NET Framework apps in Visual Studio. It is useful in a number of versions.



More and additional college students are turning to the Internet when doing research for their assignments, and more and more instructors require such research when placing topics. However, research on the net is very different from general library research, and variations can cause problems. The net is a great resource, although it has to be used carefully and critically.

The printed resources you have discovered in the library have been rated almost exclusively through professionals before they were published. This system of "peer review", for example, is a distinction between this book in the Time Journal and a journal such as the University of Toronto's Quarterly. In addition, when books and various materials come into the university's library system, they are painstaking and systematically cataloged and cross-referenced with the use of procedures seen through research libraries around the world. This method is the basis for the way substances are prepared in the library and makes it possible to have a variety of search features of web catalogs.

On the Internet, on different hands, "anything goes." Anyone can put anything on a web site, there is no evaluation or screening process, and there are no preferred ways to learn and grow crosses. Reference. This is each glory and weak point of the net - it is both freedom and chaos depending on your point of view, and it is likely that you have to pay close attention while looking on-line. There are a number of strong academic sources on the net, including a plethora of journals and websites established with the help of universities and scholarly or scientific organizations. The Toronto Library's electronic resource web page is one such educational source. Using clothes from those sources is not a problem; It's like going to the library, only on-line. These are all other things on the net that you have to be vigilant about.

### **Here are some basic tips:**

It does not matter only on network resources. Sometimes your undertaking will be to just do a lookup on the net, although usually, your instructor will expect you to use both the Internet and library resources. Cross Checking of records against the net from the library is an accurate way to ensure that the net content is reliable and authoritative.

Narrow down your lookup theme before logging on. The internet accepts that

you get entry with so many terrible figures that you can easily get overwhelmed. Before starting your search, consider what you are looking for and if possible prepare some very specific questions to direct and limit your search.

Know your difficulty directories and search engines. There are several highly brilliant peer-reviewed issue directories containing selected hyperlinks using challenge experts. INFOMINE and educational information are accurate examples. These are wonderful places to start searching for your tutorials on the Internet. Google, Bing, Yahoo, and various search engines are notable for how they work, how many tons of net they search, and the different types of effects you can expect to get from them. Spending some time gaining knowledge of what every search engine will do and how excellent it is to use can help you stay away from great frustration and later wasting time. Because everyone will search for things unique to you, it is always an accurate assumption to use more than one search engine. For search engines and directories, in particular, you would probably like to try beaucoup, which includes 2,500+ search engines and directories or the search engine Colossus International Directory of Search Engines, which includes search engines from 230+ countries in the world.

Keep a specific report of the websites you visit and the sites you use. Lookup on the net is essentially the ability to visit some websites that are beneficial and many that are not. It is necessary to keep the music so that you can tell the beneficiaries again later, and also put the necessary references in your paper. Just do not count on the history function of your browser, as it maintains the web addresses or URLs of all the websites you visit, excellent or bad, and if you are using a PC at the university then in the history file The reminder will be deleted at the end of your session. It is better to write or bookmark websites you have considered useful so that you have a permanent record.

Double-check all the URLs you have entered in your paper. It is easy to make mistakes with complex Internet addresses, and Typos will make your references useless. To be safe, type them into your browser's location container and have a look that they take you to the right site.

The following are the factor recommendations for evaluating the specific

sources you search for on the net. If you ask these questions while searching the web site, you can avoid many errors and problems.

### Audience level

For which audience is the web page created? You select the facts at the university or lookup level. Do not use websites that are too technical for your needs, used for basic college students or web sites.

### Currency

- Is the web website online?
- Is the website dated online?

Is the latest replacement date given? Generally, Internet assets must be up-to-date; Ultimately, obtaining the most cutting-edge data is the primary objective of using the net for research in the first place.

Are all links updated and working? Broken links may also suggest an out-of-date website; They are indeed a sign that it is now well maintained.

- Content Reliability / Accuracy
- Is the cloth on the web site reliable and accurate?
- Are the figures factual, not opinion?
- Can you confirm the data in print sources?

Is the supply of data realistically stated, whether authentic lookup cloth or secondary content is borrowed from elsewhere?

- How valid is the research by source?
- Does matter have depth and depth?

Where arguments have been made, are they mainly based on strong evidence and excellent reasoning?

- Is the author's point of view fair and objective?
- Is the author's language free from emotion and prejudice?

- Is the website free from errors in online spelling or grammar and other signs of negligence in its presentation of content?
- Are additional digital and print sources on the web site equipped to supplement or assist with clothing?

If you can affirmatively answer all of these questions while searching on a specific site, then you can certainly be absolutely certain; If it doesn't work out one way or another, it's probably a web page to avoid. The key to the whole process is to think critically about what you find on the net; If you want to use it, you are responsible for making sure that it is reliable and accurate.

A PowerShell script is a simple text file containing several commands that can be executed in Windows at once. Each command in the file is separated by a newline character. PS1 is an extension for PowerShell script files. PowerShell is a powerful scripting language for custom software development companies.

The PowerShell window is used to run all PowerShell script files. Need to type the full path in PowerShell script with filename only. For example - c: \ scripts \ PowerShell demo.ps1.

- How does it differ from other programming languages?
- PowerShell scripts can be compared with CMDs because they are similar in concept.

### PowerShell Cmd.exe

- It is an object-based scripting language. It is a text-based scripting language.
- CRMD command works in PowerShell cmdlets do not work in Cmd.exe

System administration tasks for managing the registry for WMI (Windows Management Instrumentation) are accessible through PowerShell. System administration functions (Windows Management Instrumentation) for managing the registry for WMI are not accessible through cmd.

It is a powerful scripting environment that can be used to create complex scripts for managing Windows systems with a command prompt. Composing complex scripts with CMD is more difficult

The equivalent command to change a directory: set-location to rename a file: name-item equivalent to change a directory generic command: CD to rename a file: rename

PowerShell is a strong scripting language and offers many advantages over traditional scripting languages such as VBScript and others.

PowerShell is an object-oriented scripting language

It provides more functions than cmd.exe and VBScript

It is more extensible via cmdlets, plugins

It supports .NET objects and forms: PowerShell has access to all .NET libraries of society software companies that can take advantage of anything with a language like C # or VB.NET. Any .NET DLL can load and take advantage of third-party libraries

It supports automation of tasks: adding one user for 500 machines is a time-consuming process if done manually. However, with the help of PowerShell scripts, anyone can automate this process.

Background Jobs: PowerShell runs on a single thread, so if someone tries to invoice a .NET library that creates a new thread like a background worker, it can run into issues. In PowerShell scripts, there is a concept of background

jobs.

**Interactive:** PowerShell is also interactive. This allows programmers to try things on the console first and integrate them into more complex scripts.

**Reusable:** Scripts written to perform a specific task can be saved for later use and also merged with other scripts to perform various tasks. Therefore repetitive tasks can be performed to run the script. This saves administrators time and money in configuring machines.

PowerShell supports Group Policy-based control of the environment

What are the disadvantages?

There are some disadvantages of using PowerShell scripts that the type of software companies should consider when using the same for software development. The following are some ideas.

PowerShell requires the .NET Framework

**Object-based:** As with most shells, text-based commands are used to get the job done when writing scripts. If you switch Windows PowerShell from other types of shell, programmers have to get used to thinking in a different way. This can take time to get used to.

**Security Risk:** Another possible drawback of using Windows PowerShell is that it can pose some potential security risks. Many IT professionals use it as a way to remotely connect to other computers and servers. When engaging in this process, PowerShell may leave some loopholes for security frameworks. This is the major drawback of using PowerShell scripts.

**Web server:** Another problem with Windows PowerShell is that the webserver needs to run on the server when using remote functionality. It takes up additional space on a server. In many cases, custom software development companies may not be ready to specify more resources for it.

- Which applications are appropriate for PowerShell scripts?

- The following applications are used to execute PowerShell scripts:
- Microsoft Windows PowerShell ISE
- Microsoft Windows PowerShell Web Access, via Control Panel
- Search
- Sapien Technologies PowerShell Studio 2015
- Amazon AWS Tools for Windows PowerShell
- Adam Driscoll's PowerShell tool for Visual Studio
- VMware vSphere PowerCLI

Time is the manager of a team of server engineers in New York City. He is a Microsoft Certified Trainer who specializes in SQL Server, SharePoint, and VMWare. He is a contributor to the Windows PowerShell Forum and Hey, Scripting Guy! Forum. Time is a part-time this booger and is currently organizing an inaugural meeting of a Windows PowerShell user group in New York City.

Tome also wrote about yesterday's this book remotng Office Power with Windows PowerShell 2.0 and WinRM. Thanks to me!)

System directors face many challenges. Whether it is searching for a great software program solution to a new or existing problem, or discovering a great state of caffeine for an all-nighter, every option has consequences that can each be desirable and worse. Have you ever gone with a piece of software that fully meets all your requirements to analyze after three months when the first problem is that it is almost impossible to get support on the phone? When you get them online, do you realize that you are the first buyer to try to use a great deal of information with their application? Have you ever tried drinking caffeine from a diet shop? Surely, you are there for an hour, though when you are moving uncontrollably and your fingers hit the

keyboard so hard that you start cracking the plastic, you realize that you have  
Also not a complete assumption.

All of us have made very negative decisions while swimming our way of life, though unknowingly in the IT world, with very little tolerance for error. Depending on the industry, every mistake should potentially cost your buyers or agency millions of dollars. So what is the high-quality way to reduce these mistakes? How can I make sure that I am considering every angle?

Definitely, write your thoughts all over the world, on a very publicly visited web website to watch and critique.

So, barring further ado, let's talk about Windows PowerShell as excellent and horrifying, to decide whether or not I'm going to enable it on all my servers as standard or not. If you are new to the remoting world, I study Ed Wilson's this book as a primer.

Ease of administration

There is something to be said about launching a shell in a far-flung PC to quickly retrieve a piece of information. I think the coldest element in Windows PowerShell is the versatility of the set-location and gate-children cmdlets, which are so fondly alias through the comfortable DOS names of CDs and DIRs. Traversing the registry has become so much less complicated from a command line that I use it continuously instead of re-edit. Now, open that power to all your computers from a single shell going to run from your workstation. Perhaps you want to confirm that an Office Outlook COM add-in is installed on the workstation. What's easier than this:

Although this used to be a rhetorical question, I would answer by saying,



"Absolutely nothing is simple!"

You can run commands that are designed to work only locally

As a part-time SharePoint administrator and full-time SharePoint user, this used to be the most compelling purpose for me in what appears to be remaking. Although this idea should apply to different systems, SharePoint is misused in this way. One of the limitations of the SharePoint object model is that your code must run locally on the SharePoint web interface.

Here is a pattern script, which connects to hundreds of object models, a SharePoint site, and adds a new entry to my leading calendar:

```
$ Script = {
```

- `$ site = new-item Microsoft.SharePoint.SPSite ("http: // server1")`
- `$ web = $ site.OpenWeb ("")`
- `$ list = $ web. Lists ["calendar"]`
- `$ items = $ list. Items.Add ()`
- `$ items ["title"] = "listen to the powerscripting podcast"`
- `$ items ["start time"] = "Thursday, February 04, 2010 9:30 AM"`

- `$ items ["Location"] = "Online Live!"`
- `$ item. UPDate ()`
- `}`
- `Invoke-Command -ComputerName server1 -scriptblock $ script  
-authentication credits`
- The following figure suggests an object brought to the SharePoint calendar.
- SharePoint calendar item image
- Future administration packs may also need it

It is extremely important to note that Exchange 2010 requires remoting in order to use the management pack. When you load Exchange Shell on your workstation, it provides you with a regular off-shell from the nearest Exchange server. Is this the future of the management pack? Although I have not read or heard anything that proposes that it goes to the previous Exchange, it is for this purpose that it may be the preferred method to use

Windows PowerShell on all future server products.

Run the script on more than one server with an invoke-command cmdlet.

It is top-notch admirable because it could theoretically change the way you strategize scripting tasks to ensure that you adopt the easiest approach.

Some cmdlets have their own interfaces for receiving records from far-flung computers. Take Get-WMIObject for example:

```
Get-WMIObject Win32_Service -Computername server1
```

It's an unreliable interface for jogging a command on remote computers but think of a more complex system that assesses if a carrier exists, stops that service, and then uninstalls it. Writing that a computer is far-fetched and challenging is no longer possible. With Remote, you can write a script to do this challenge on your local computer, and then run this script on several computers via the invoke-command cmdlet:

```
Invoke-Command -ComputerName ("server1", "server2", "server3") -  
FilePath .RemoveService.ps1.
```

If you are aware that remoting is easy in your environment, you can create a scripting strategy where you can adopt convenient routes across all routes, while having the flexibility to run on your computer as much as you do. Should be kept, as much as you want to do with very little effort. We use scripting languages to find a quick answer to a problem. The use of this method will fully aid that effort.

Import far-flung modules to your computer

After you create a session on the remote computer and load the appropriate modules or snap-ins such as those equipped with SQL Server or Exchange Server, you can run the Import-PSSession cmdlet to add those modules to your nearest shell:

- `$ Session = new-PSSession server1`

- `Invoke-command-session $ session-script {import-module import}`
- `Import-PSSession $ session`
- Remoting makes way for an attacker to exploit

This is a valid argument in the malware and virus-infected world. With a well-designed network relying on exquisitely configured firewalls and computers, this is well worth taking a threat. There is a reason as administrators, we usually enable remote laptop connections even if it is a risk. If your laptop is exposed to the Internet (in a DMZ), however, I think this argument needs to be accepted. I would by no means open a port that was not wanted inbound on my firewall.

WinRM requires a webserver to run on the server

The real problem with this is largely equivalent to the first point. People do not like running pointless services on their servers. If you are more relaxed or are already running IIS you can configure WinRM to run under IIS. You should assess the setup and configuration documentation for WinRM. Even if the default settings enforce encryption, it is likely that it is a good practice to spend time getting SSL to work. You also need to change the default port and perhaps reflect the on consideration to change the default URL prefix.

When Microsoft first developed the PowerShell scripting language a few years ago, many of the engineers I spoke to were resistant to knowing it because they preferred the comfort of the GUI. To be honest, I was really reluctant about myself. Now that I have been working with PowerShell for some time, however, I see that it presents many special benefits. Power cell:

Allows getting information otherwise easily. Depending on the software you are working with, you may also find that some information does not work entirely on the GUI, while various facts are available, although time-consuming to use. PowerShell fixes this. And if you're working with Microsoft Exchange, PowerShell gives you access to everything.

For example, you want a list of all the people in your organization who have "full mailbox access" rights to other people's mailboxes on your Exchange server. To use these statistics GUI, you have to go to each character user, click something to see their reputation in this field, and then reprogram and paste into the replacement sheet. Even if you only have 20 users, it must be painful! With PowerShell, a script will quickly provide you with a north-wide enterprise.

Quickly complete shaving tasks. Remember, "scripting" possible automation - so just about anything you can do in the GUI with a few mouse clicks can be accomplished more quickly in PowerShell. If you are undertaking more than or three times, then why not do it in the script now?

For example, perhaps you prefer to monitor the dimensions of each of your Exchange databases (which Explorer will report without "white space"). You cannot enter this statistic through the GUI ... however, you can create a one-line PowerShell command that will give you the size of every database faster.

Big jobs help to get quick. PowerShell is also available with a few large jobs that likely won't come very often, such as bulk imports, bulk migration, or growing multiple customers simultaneously. These cases are very time consuming to complete the use of the GUI and can appear like challenging tasks. PowerShell can certainly shine here, as it has command lets to manage these jobs quickly and easily.

After a 10 12 months' hiatus I again my attention to the world of computing, studying (and loving!) Perl which I often used for systems administration and migration tasks. I wrote tools like SetACL (in C++), dabbled in Visual Basic (at the time not based totally on .NET yet), dived into the arcane syntax of Windows batch files and picked up normal expressions along the way. Later I

wrote a couple of tools in C# and VB.NET (based on the .NET framework), e.g. SetACL Studio. For my cutting-edge product, uberAgent, I went lower back to C++.

I am citing all of this so that when you figure out that I am a fool after analyzing this book you at least provide me a credit score for being an idiot who has seen a bit of the programming world.

## **From Monad to PowerShell**

I heard about PowerShell first when it was still known as Monad. At the time I was overjoyed by the prospect of having a powerful scripting language for Windows. I was hoping for something like VBScript turned correct or Perl being protected in the default deploy of Windows. My enthusiasm used to be such that I wrote a paper explaining PowerShell and introduced it to colleagues and customers.

However, over time my enthusiasm waned. I determined extra and extra quirks in the language that made life unnecessarily difficult. Certain things that be easy and intuitive are not. Perl's slogan Easy things ought to be effortless and difficult things must be possible does not apply to PowerShell.

The rest of this submit is a listing of things that, in my humble opinion, are incorrect with PowerShell. Some of the factors may additionally be subjective. I might be wrong about an aspect or (in which case please correct me through commenting and I will restoration the error). You have been warned.

### **Syntax**

- Operators
- PowerShell's operators are horrible. E.g. checking out strings for equality:
  - `"ABC" -eq "ABC"`

- Instead, that be something like:
- `"ABC" == "ABC"`
- The identical applies to most other operators. E.g. comparing numbers:
- `8 -gt 6`
- Instead of:
- `8 &gt; 6`

## Escape Character

Having the back tick as an escape personality is simply undeniable weird. Chances are if you have used different programming languages the backslash feels an awful lot greater natural.

- `"Let's print line one` and line and a quote"`
- Instead of:
- `"Let's print line one\nand line \n and a quote \"`
- If Requires Curly Brackets

Many languages let you do away with the curly brackets if you have solely a single statement in an if. Not so PowerShell. Instead of this:

- `if (5 -gt 3)`
- `"hello"`
- You want to write this:

- `if (5 -gt 3)`
- `{`
- `"hello"`
- `}`

## Function Definitions

Functions can solely be used after they have been defined. This leads to constructions like this one where the script effectively starts off evolved at the end:

- `function main`
- `{`
- `# do stuff`
- `}`

- `# Call main`
- `main`

## Function Calls

PowerShell functions are called except parentheses, however, .NET functions are known as with parentheses.

- E.g. calling a self-defined PowerShell characteristic  
`LogMessage:`
- `LogMessage "Here's a message"`
- Versus calling a .NET function
- `[string]::IsNullOrEmpty($configFile)`



## Script Syntax Check

Scripts can't be syntax-checked before they are run. If you have a syntax error in a code branch that is now not carried out all through testing you might only find out about it after the script has been deployed to production machines.

E.g. PowerShell, fortunately, executes the following code besides the slightest warning:

- `Set-StrictMode -Version 2`
- `$variable = 10`
- `if ($variable -eq 30)`
- `{`
- `# It probable ought to complain about this`
- `}`

## Lack of Mandatory Variable Declarations

This (along with the subsequent point) is by means of a long way the worst I have observed while working with PowerShell.

If you prefer to write production-quality code, you need a way to pressure the language to pressure you to declare variables before they are used. Perl has its use strict, heck, even VBScript has alternative explicit. Only PowerShell has nothing.

Take a seam at the following code. You count on it to print “20”, however, due to the fact of a stupid typo it prints “10”:

## Set-Strict Mode -Version 2

- `$variable = 10`
- `# different code`
- `$variable = 20 # Notice the typo`
- `# print value`
- `$variable`

There is a Connect item on this.

Today's challenge is to automate the advent of a VPN button for Windows 10—based far off users here at the office. End-users then in principle can just double-click a PowerShell script that I've placed on a SharePoint server. I would then in my view share the link with them which would remotely installation the new VPN profile. Sounds convenient enough. In fact, it sounds a whole lot easier than the -page-long tutorial in a Word record which tries to instruct them how to do all this manually. Have you ever viewed how long an L2TP shared key phase can be? It's notably bad. Just assume of all the guide calls I'm going to get if I can't script this.

Is the PowerShell documentation effortless to use? Hell no, it's not. I've simply spent a full hour making an attempt to piece together the script required from this hobbled-together documentation on Add-VPN connection. Does my script work below a take a look at rig? I desire I knew, due to the fact that at the moment I can't really run the script in any structure or trend because Microsoft doesn't choose me to.

Now granted, I'm an Administrative consumer on my newly-upgraded Windows 10 laptop. The script fails with some terse error message which

suggests that I want to run the PowerShell command as Administrator. Well, that would foil matters right here in the real world because I'm making an attempt to have the end-users run this script remotely so that I—the administrator—don't have to be there in the first place.

So I doggedly trudge ahead and stop my session and open up PowerShell with the aid of right-mouse clicking it and choosing Run as Administrator. And yet, this nonetheless doesn't work. This time it fails with every other terse error message which suggests that Set-Execution Policy would possibly help. I then research this to find that "Unrestricted" is the possible attribute but when trying to run this, I get every other terse error message suggesting that I can't change the policy. Seriously?

I should now go returned to my beforehand research and re-learn how to digitally sign a script so that I can run it. But the manner to create and to troubleshoot a script typically requires a couple of iterations before the script works perfectly. And this is mainly authentic since nobody but on the Internet has supplied a suitable example for growing a VPN tunnel to a SonicWALL over L2TP/IPsec with a pre-shared secret and authenticating to the firewall as an alternative of the domain controller. Designing a script like this takes trial and error. Adding a signing segment between each script strives efficiently means: I'm not going to do this.

"Adding a signing segment between each script tries correctly means: I'm no longer going to do this."

In short, this is why Microsoft PowerShell sucks. If you have to signal scripts just to run them while checking out, then it's not well worth the effort. Why not consist of a button in the PowerShell IDE which allows me to "Sign & Execute" my script attempt? And if I don't have a digital certificate then open a dialog field to acquire the records to magically make this happen. Or even better, just permit me to create and run scripts besides all the nonsense. How about a large toggle that says "Unsafe Mode" versus "Safe Mode"?

A shell is an interface, regularly a simple command line, for interacting with

a running system. PowerShell specifically includes a scripting language and helps gadget administrators automate duties on their necks, configure devices, and usually remotely manipulate a gadget. A framework like PowerShell has many neck security benefits, due to the fact that it can facilitate tediously, but necessary tasks such as updating and configuring large amounts of devices.

But similar features that make PowerShell versatile and convenient to use - it depends on instruction to units at certain points in a neck - also make it an attractive device for attackers.

When Microsoft first developed PowerShell for release in 2006, it directly identified the framework's viable security implications. "We honestly knew that PowerShell was going to be [attractive]. The attackers have satisfaction with the job as well," says Lee Holmes, a sketch engineer, a chief software program for PowerShell, and a security architect for Azure Management Group for Microsoft. "But we have been laser-focused on Powderly Protection since the first version. We have consistently approached this in terms of large machine safety."

Outside observers also incurred these viable losses. Companies such as Symantec targeted PowerShell's practical ability to quickly spread the virus for the duration of the neck. Before it was even formally released, however, Microsoft took steps to make it more difficult for attackers to quickly stop the framework through precautions, such as banning Who has given what instructions and is required to sign the script via default — including the process of digitally signing to certify that a lower The brand is legitimate, so attackers cannot just freely enter something they want. But initially, it was made significantly less of a hacker target at first because of its restricted distribution of PowerShell, after it was recognized by Windows that started making it fashionable with Windows 7 in 2009. Microsoft made it an open-source tool for the rest of the year.

Penetration testers and researchers David Kennedy and Josh Kelly wrote, "We will be the first to properly acknowledge the utility and strength of Powderly. The ability to conduct superior duties is a major upside in

Microsoft-based work structures." The first Deco security conference speaks about PowerShell, returning in 2010. But "PowerShell additionally gives hackers full-flow programming and scripting language at their disposal on all work systems by default ... [which] poses a good size security risk."

## **Domino effect**

Microsoft's security precautions saved hackers from using PowerShell for total acquisitions, although attackers increasingly saw that they wanted to use it for positive attack steps, such as remotely setting the settings on a unique device Adjusting, or initiating malicious downloads, even if they cannot remember. On PowerShell to do everything. For example, the Odinaf hacker group took advantage of the malicious PowerShell script as its sting of attacks on banks and various financial establishments last year. And the popular "W97M.Downloader" Microsoft Word macro properly uses the PowerShell prompt to spread Trojan malware.

An unavoidable feature attacker discovered was that PowerShell did not provide particularly comprehensive logs of its activity, as well as most Windows users not setting PowerShell to record in logs. As a result, attackers may want to misuse the framework with a simple approach, in addition to a victim mechanism somehow flagging the effort.

Recent changes, however, have made the attacks less complex to detect. PowerShell 5.0, launched last year, brings a full suite of manifold logging tools. In a system attack recorded through response company Mandiant, which collaborates with Microsoft's team several times in PowerShell Research, Advanced Persistent Threat 29 (also accepted as Cozy Bear, a team that has been linked to the Russian government) has used a multi-dimensional attack that incorporates a PowerShell element.

As they were removing the machines, they were again compromising, "Holmes says of Mendicant's defense efforts." They knew the attackers were using Python and a combination of command-line tools and system utilities and PowerShell - all this stuff is out. Are you there So they update the gadget to use a more recent version of Powersley that has extended logging, and suddenly they want to look at the logs and see exactly what [the attackers] were doing, what machines they have Adding to, he carried out each

command. It certainly took away the veil of secrecy. "

While this is no panacea, and will not protect attackers, new focal points on detection and detection of logging aids. It is a basic step that helps an evacuation and response after an attack ends, or if it persists for a long time.

"PowerShell built the ability for users to determine which logging they select or require, and with that, they have added a pass by the policy. As an attacker you can both Whether you're going to log the event or you're going to bypass the command, it's an important Crimson flag, "says Michael Viscous, CTO, Carbon Black, Mr. tracks PowerShell trends in the stage of intelligence research. "From that point of view, the developers of PowerShell have stopped the type of attackers to make a decision. Nevertheless, if an attacker allows you to log their activities, assuming they have any type of privilege on the machine Granted, they can take the bus. They log in later. It's a road, but it's usually more inconvenient. "

And PowerShell's recent defense improvements go to the previous log. The framework recently introduced "constrained language mode," to manage additional instructions on what PowerShell customers can implement. Signature-based antivirus has been able to flag and block malicious PowerShell scripts for years, and the launch of Windows 10 accelerated the capabilities of these services through integrating the Windows antimalware scan interface for deeper running machine visibility. The large-scale security enterprise has additionally attempted to determine what every day basic processes for PowerShell should look like, indicating malicious behavior due to the fact. The time and source it takes, in my opinion, establishes that baseline, however, on the basis that it is different for each organization's neck.

Penetration testers - security experts are paid to sabotage necks so that businesses can plug holes that they have paid increasing interest to for PowerShell. "The past 12 months appear to have produced in the pastime from the Pentest community, with conscientious shielding conscientiously paying extra interest for PowerShell-related attacks," Will Schroeder, a conservation researcher who researches PowerShell capabilities.

## **Shell game**

Like any security mechanism, however, these measures provoke invasive innovation. At the Black Hat and Defcon security conferences in the final month of Las Vegas, Microsoft's Holmes gave more than one presentation on which strategy attackers might want to use to hide their pastime in PowerShell. He also confirmed how customers can set up their systems to maximize devices to gain visibility and protect their behavior to minimize misunderstanding attackers.