

# **POWERSHELL**

## **For Beginners!**

**Master The PowerShell  
Command Line In 24 Hours**



**ALEX ARTUSO**

# **PowerShell For Beginners!**

## **Master The PowerShell Command Line In 24 Hours**

## **Copyright 2015 by Alex Artuso - All rights reserved.**

This document is geared towards providing exact and reliable information in regards 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.

- From a Declaration of Principles which was accepted and approved equally by a Committee of the American Bar Association and a Committee of Publishers and Associations.

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 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 the owned by the owners themselves, not affiliated with this document.

# Table of Contents

## **Introduction**

A step back into PowerShell History

Who are we speaking to?

How to use the book

## **Chapter 1: Setting up your PowerShell Playground**

Get the PowerShell running

Customize the Shell

Online Resources

## **Chapter 2: Hellp PowerShell!**

The Standard Console Window

The ISE Window

Confirm PowerShell Version

## **Chapter 3: Discover Commands**

Updating Help

Asking for help

Looking for Commands

Tab Completion

Wildcards

Summary Help

Interpreting Different types of parameters

Paramater Values

Type of Parameter Values

Adding value to strings

Getting examples

Getting common parameters

Access online help

## **Chapter 4: Working with Commands**

It's a shell

Elements of a command

Creating a command

Terminologies

Naming cmdlets

The Alias

Taking shortcuts with parameters

Show-Command

External Commands

Error Messages

Cmdlets typing rules

Parameters Typing Rules

## **Chapter 5: Piping Function**

CSV Export

XML Export

Compare Object Command

Out cmdlets

Convert Commands

Stop cmdlets

## **Conclusion**

# Introduction

For some time now, a lot of Windows PowerShell books and instructional materials, and even Windows PowerShell presentations, have been developed for people with VBScript experience. To understand the technical instructional language of the book or attend the classes, readers and attendees were required to have an understanding of scripting. Understandably, it limited the number of people who could learn from the Windows PowerShell literature that was being made available. And as much as certain Windows administrators want the Windows PowerShell to be the next VBScript it is not likely to happen.

The Windows PowerShell is not just a scripting shell. It is first and foremost a command-line shell where people can run command-line utilities. There is an option not to use Windows PowerShell for scripting but it's not possible to use PowerShell without running a command-line function. It's likely the first and main use of the shell for any person is to perform command-line functions. You can use it for scripting sometimes, but it's not possible to not use the shell for commands.

In this book, we will focus on discussing the key purpose of the Windows PowerShell - its command-line utilities. If you are interested in scripting then this book is not for you. But if you want to know everything there is to know about the PowerShell command-line functions, well at least as much of the basics useful for any beginner, then read on.

With this book you will learn all the command runs that make the Windows PowerShell such an effective tool for providing automation skills to any Windows Systems administrator. It has the most flexible and user-friendly methods of automating a range of administrative tasks. The Windows PowerShell is guaranteed to make any Windows administrator more efficient and effective at work.

## **A Step Back Into Windows PowerShell History**

Systems administrators have come a long way since the days of the Graphical User Interface (GUI). The first tool Windows administrators used to manage their company's Windows Systems. GUI was actually the inspiration for Windows PowerShell.

It was just too time consuming to accomplish tasks using GUI. A more efficient tool was required to reduce the work process and allow administrators to deliver work faster. For example, if in the past an administrator using GUI required 5 hours to add new users into a computer system Windows PowerShell was able to reduce the time to accomplish the task by an hour. The less than efficient GUI therefore caused and inspired the creation of its more efficient counterpart – the Windows PowerShell.

Expect the Windows PowerShell to improve through the years as features get more streamlined to service the needs of Windows administrators. In the future, all tasks related to automation and administration should be possible to manage within the shell in the quickest possible time using the most precise commands.

### **Who is the book speaking to?**

This book is for Windows Systems Administrators who are responsible for managing a company's integrated Windows systems. This book is also for Systems administrators who run commands and are in constant search for command tools to add to a command knowledge base. New Windows Systems administrators who have just jumped into the fray and are looking for a beginners guide to Windows PowerShell to get them started would also find this e-book handy, and so would anyone looking for a career in the field.

### **How to use this book**

Since the book is focused mainly on how to run commands on the Windows PowerShell suggest finding time every day to read through and go through the tests of each chapter. It is not recommended to rush through the book and read it in one sitting. It's not the best way to absorb all the technical information you'll be getting from reading it, much less acquire the skills it will teach you in every chapter.

Sure and steady is the best pace to be reading the book. Make sure to immerse yourself in each chapter and make the most of the practical tests available to really sink your teeth into the command functions of the shell.



# Chapter 1: Setting Your PowerShell Playground

Here are the basic requirements for setting up your Windows PowerShell playground. The space where you get to play around with the command functions of the Windows PowerShell. It is necessary to make sure that all the technical components are available and installed to ensure you get the most instruction from the book:

1. Windows with a PowerShell version 3 at the very least. The minimum would be the Windows 8 or Windows Server 2012.
2. “x64” Operating System. To check, go to start menu. Open the apps folder. Look for the Windows PowerShell folder. If inside you find the files Windows PowerShell and Windows PowerShell ISE then you have the right OS. That means that version of the Windows PowerShell is the required 64-bit version.

**NOTE:** The “x86” version that is the 32-bit operating system is not compatible with the instructional materials of this book. It is recommended to upgrade to the 64-bit version of the OS.

## Get the PowerShell Running

Although in most PCs the Windows PowerShell V3 is already pre-installed it's always good to check if the computer you are using actually has it. It's easy to check if the PowerShell V3 is installed in the computer.

Go to Start Menu, click on Apps, and open the Windows PowerShell ISE Folder. On the Windows PowerShell console type `$PSVersionTable` and press Enter. Check PS Version information that will appear on the screen. It should say version 3.0. If it does not then you will have to make an upgrade.

If you have to install an upgrade then visit <http://download.microsoft.com>. Go to search box and enter PowerShell 3. Download Windows Management Framework 3.0 which should have the PowerShell V3. Reminder to download the 64-bit package as it is the OS system that this book is compatible with.

**Note:** Windows PowerShell V3 has 2 components to choose from to engage the shell. The first is the PowerShell.exe. It is the text based console host. The second is the PowerShell\_ISE.exe. It is the graphic based console. It provides a more visual Integrated Scripting Environment. Both work for the purpose of this book. So, it's a matter of preference for the reader. The book however is geared towards the .exe console host version. The recommendation is to work with that console.

## Customizing the Shell for PowerShell.exe



If you have decided to use the text based PowerShell console, we highly recommend customizing the console features before starting work on the shell. Some changes to consider:

*Font:* Click on the PowerShell icon, the control box located in the upper left of the console. Select properties from the Menu, in the dialog box browse through tabs to change the default font to Lucida fixed-width font. This is the optimized font for the PowerShell commands function. To ensure the words, characters and symbols being typed in the console are readable increase the font size to more than the default font size 12.

*Color:* It is also possible to change the color of the console background and foreground. Choose your preferred colors.

*Window Size:* The window size and the screen buffer should have the same width values.

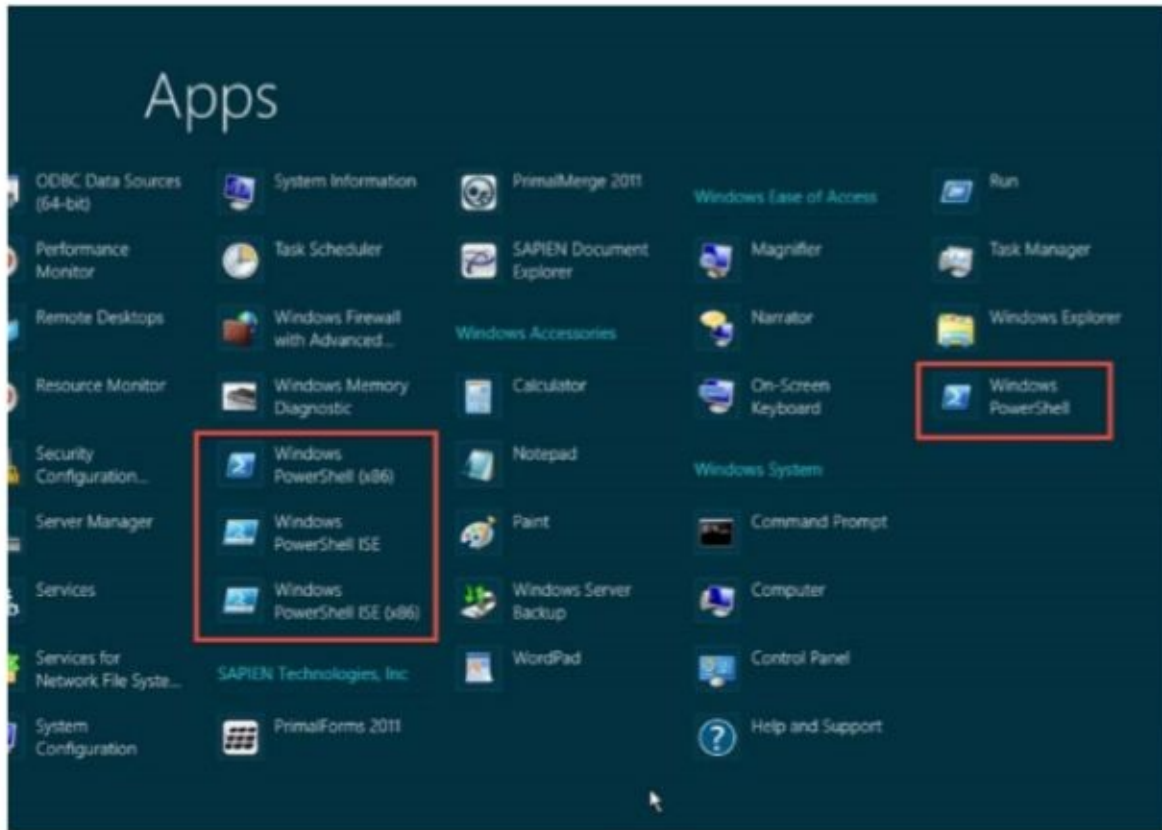
### **PowerShell V3 Online Resources**

For additional information on anything that concerns PowerShell V3 go to [MoreLunches.com](http://MoreLunches.com). The website provides relevant information on code listings which can be downloaded. You'll also find in the website answers to the book's test questions. Articles, videos, and links to blogs about PowerShell V3 can also be accessed in the website.



## Chapter 2: Say Hello To PowerShell

It's been mentioned in the introduction that there are 2 types of consoles available to the Windows PowerShell user. Well, to be exact there are actually 4 console options. To preview go to Start, open All Apps. Figure 2.1 shows the desktop screen view after clicking on Apps.



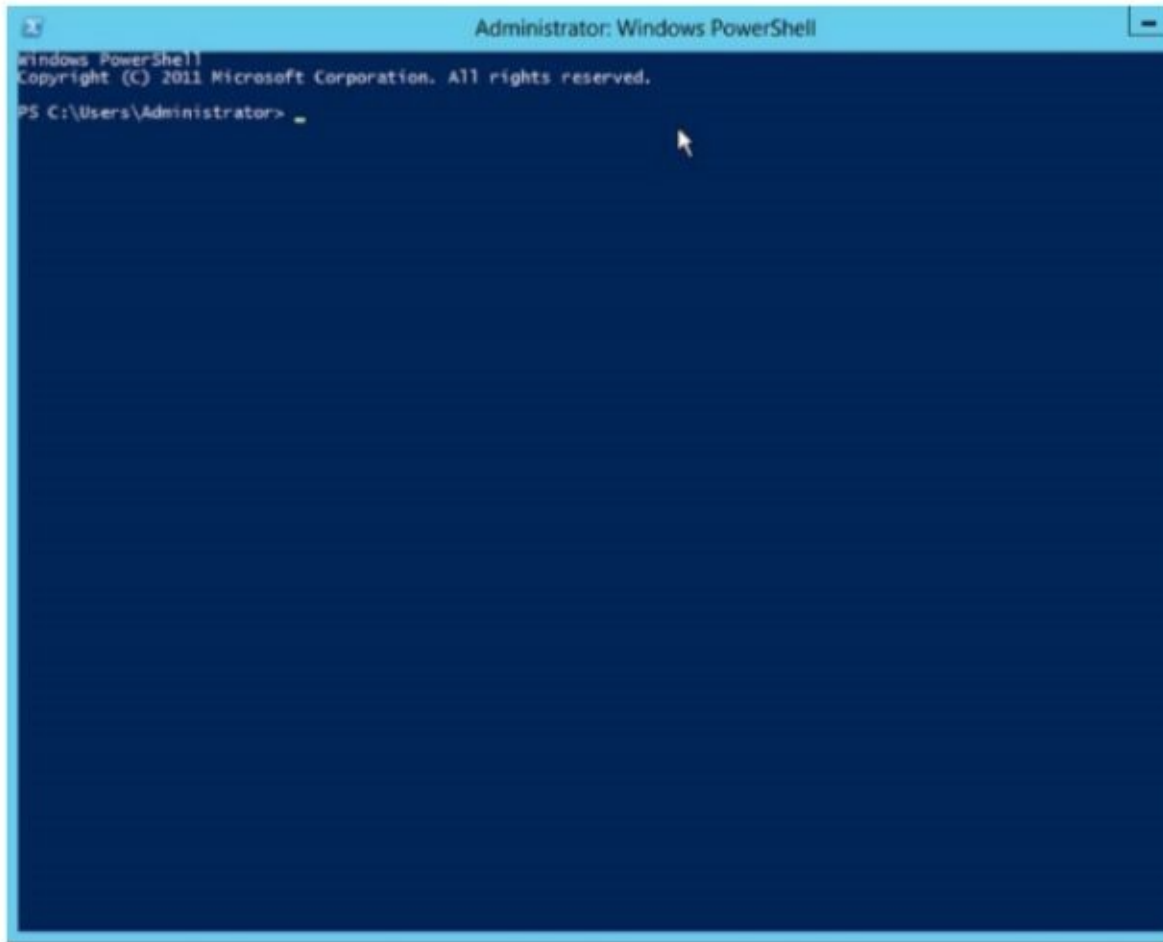
**Figure 2.1**

We said earlier that there are 2 types of consoles available on the Windows PowerShell. These are the text based and the graphics based consoles. It was also mentioned that there are 2 kinds of OS systems that power the Windows PowerShell. These are the 64-bit OS system and the 32-bit OS system.

On the desktop view of the PowerShell icons the 32-bit system text and graphics based consoles are the ones with the (x86) sign while the 64-bit versions are the ones without.

**NOTE:** For the purpose of the book, it is recommended to work with the 64-bit system using the standard .exe text based console.

### The Standard PowerShell Console Window (Text Based)



**Figure 2.2**

Throughout the book we will be using this PowerShell console. See figure 2.2. The reason being it is a simpler and less complicated console than the graphic based option.

### ***Configuring the text based console***

All tabs required to custom tailor the console can be found in the Properties tab on the control box of the window. The control box is the windows logo located in the upper left corner of the window. The properties folder will have the following tabs:

**Options:** Recommend to increase the size of the Command History Buffer Size. Important because you would like to be able to read clearly information on the commands typed in the past.

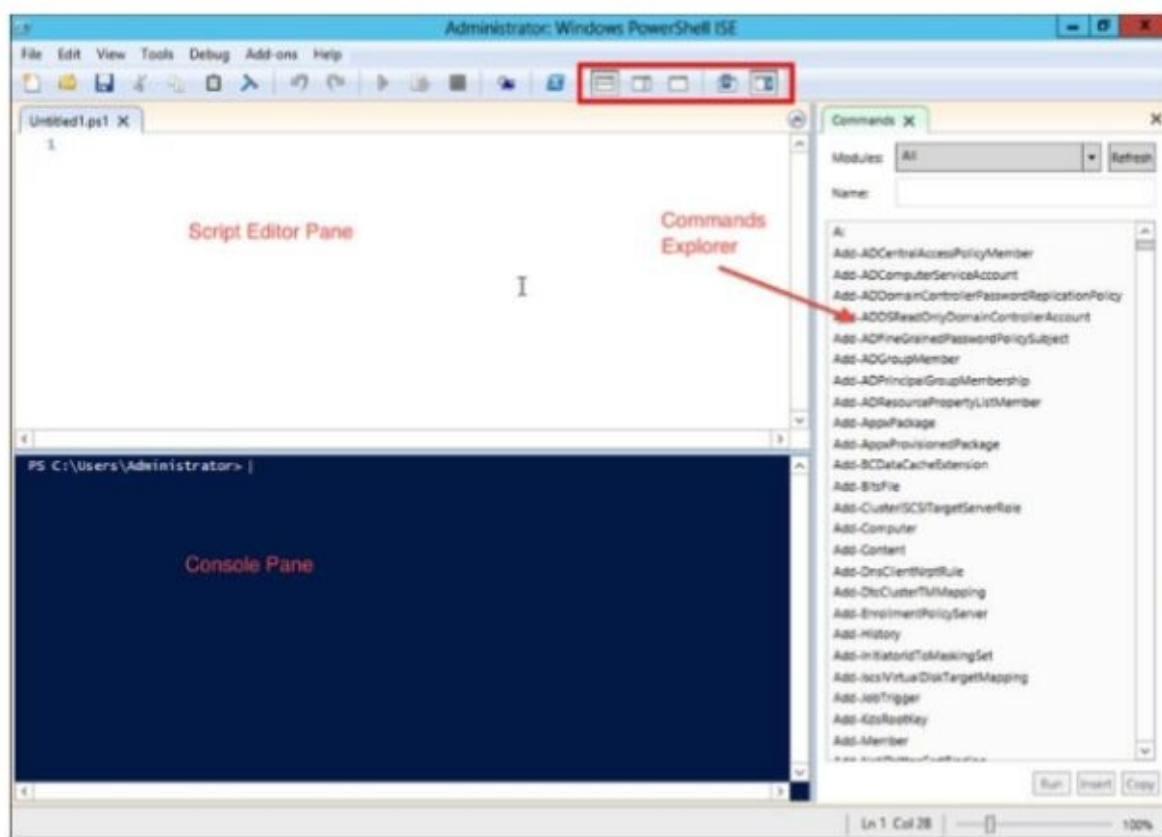
**Font:** The preferred font is Lucida fixed width-font. Font size should be more than the 12 default size. Since much about working with command runs is typing accuracy you would like to make sure every letter and symbol you are typing is accurate. And the best way to check is with a big font that allows you to see the exact character on the screen.

**Layout:** Width sizes should have the same numbers so that the window fits exactly in the screen. Avoid extending the console size too much to the point where certain areas are no longer visible on the screen. It ensures all information typed on the screen is readable.

**Colors:** If you prefer to see different background, font, and foreground colors, go ahead. Color schemes are up to you but in case you tweaked them to the point that you are no longer happy with what you see, there is always the option to go back to the default blue and white colors.

## The Integrated Scripting Environment (ISE) Console

The upside of using the graphics based console (ISE) is that it supports double-byte character sets and, since it is meant to be more visual, it is a niftier PowerShell console to work with. Another advantage is it does more to help create PowerShell commands and scripts for advance users. And unlike the text based version you can copy-paste items on the console. See figure 2.3 below for the screen view of the ISE console.



**Figure. 2.3 The Integrated Scripting Environment (ISE)**

The ISE has limitations though, some of which are: it does not support transcription, it takes longer than the text based console to get working, and it needs the Windows Presentation Foundation (WPF) for it to work. This means the hardware or the computer should have GUI installed otherwise the ISE will not function.

Many Windows PowerShell users prefer to start with the text based console and move up to using the ISE version. Since the ISE was designed for a more optimized command run and scripting experience the eventual shift to the ISE has

been observed from all Windows administrators using the PowerShell.

Since the book was written for PowerShell beginners it is only appropriate to begin with the text based console. There will be plenty of time to transition to the ISE console when you get better at working with PowerShell. For a start, might as well get comfortable with the PowerShell.exe first.

## No Typos!

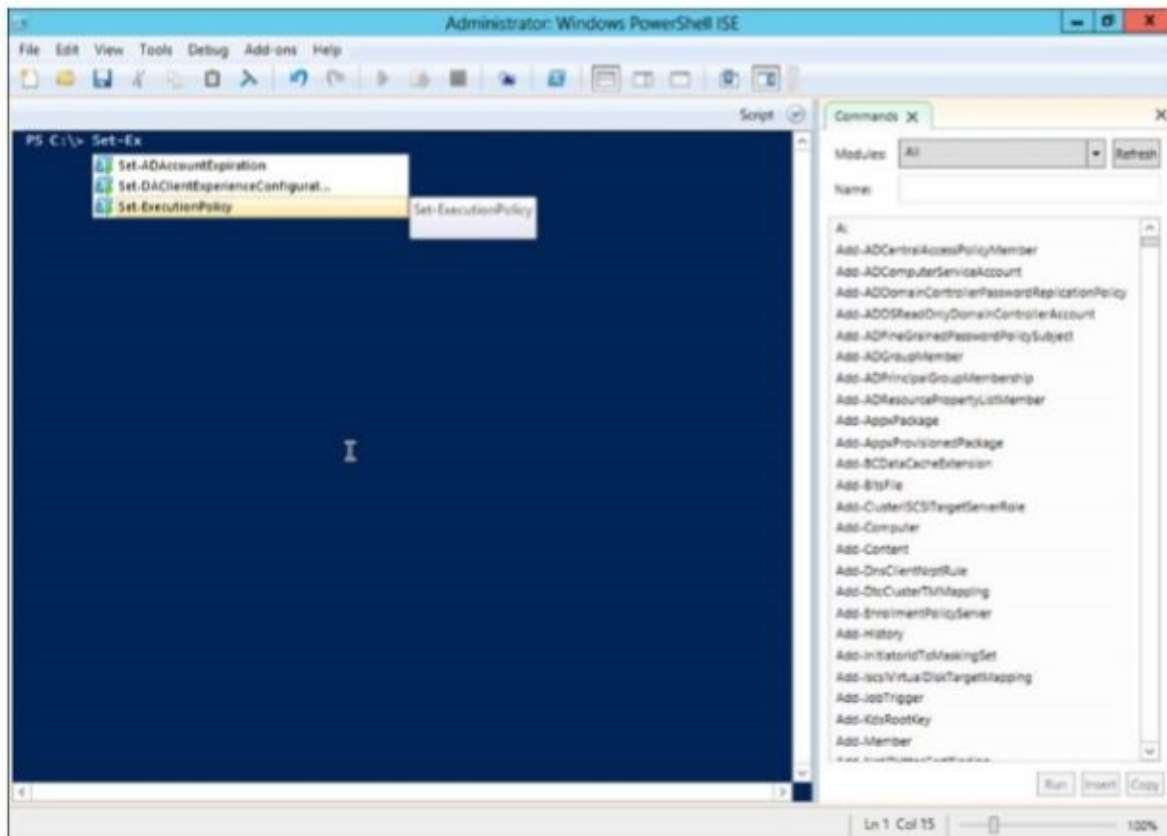
Since the PowerShell is a ‘command-line interface’, expect to do a lot of typing. And that means accurate, error free typing. Literally no typos allowed because otherwise the console will not be able to generate the right answers and responses.

Let’s give it a try. You might as well start getting comfortable with the tab selection function of the console. Let’s go through several exercises:

1. Get-S, press Tab several times, and then press Shift-Tab. PowerShell will cycle through files. The intention is for the user to scan through the options to get at the right tab required. When done, press Esc to clear the console.
2. Press Dir space c:\ and then press Tab. PowerShell will cycle through folders much like in exercise 1. Press Esc again to clear the console when done.
3. Set-Execu then press Tab, add space and a dash (-), press Tab several times. This time PowerShell will cycle through parameters. Again press Esc to clear the console.

You’ve just learned the importance of tab completion. Pressing the tab will generate the various options available in the shell. It gives the user information on a variety of commands, functions, or parameter names inscribed inside the shell.

***Intellisense:*** The pop-up menu that appears on the right side of the PowerShell ISE console is referred to as *Intellisense*. See Figure 2.4 below. Scroll up or down on the menu to look for the item to search then hit Tab and continue typing to get desired information or result.



**Figure 2.4 Intellisense**

**NOTE:** We cannot stress this enough: Accuracy in typing is the single most important skill to acquire while working with the PowerShell console. There have been instances when a simple typo error caused the entire system to fail. To ensure functionality of the PowerShell console it is best to be very alert and cautious when typing in each character. Double checking words and symbols is a must.

## How to confirm for your version of the PowerShell

Let's make sure the computer is actually running on at the very least the V3 version of the PowerShell. On the console type this command: `$PSVersionTable`

```
PS C:\> $PSVersionTable
```

Name	Value
PSVersion	3.0
WSManStackVersion	3.0
SerializationVersion	1.1.0.1
CLRVersion	4.0.30319.17379
BuildVersion	6.2.8250.0
PSCompatibleVersions	{1.0, 2.0, 3.0}
PSRemotingProtocolVersion	2.2

**Figure 2.5 PowerShell V3 detailed information**

Figure 2.5 above should appear on the console. The value on the item *PSVersion*



should say 3.0 or higher. If it does not indicate that then you'll have to upgrade or download the V3 version at a minimum. The table also provides information on all the PowerShell related technologies the version is compatible with.

## **PowerShell Laboratory**

Let's go through a few console configuring tests to make sure your console properties are primed for the next lessons:

1. Customize the color of the console to your specification.
2. Customize the font to your specification. Be reminded though that our recommended font is Lucida font and the size should be bigger than the default font size 12. You can choose to work with a different font if that is the preference and if Lucida is not satisfactory.
3. Check for readability of characters and symbols. Type a quote ( ' ) and a backtick ( ` ). Check to see if you can distinguish between the two.
4. Type a parentheses ( ), type a square bracket [ ], type an angle bracket < >, type a curly bracket { }. Make sure the symbols are easily identifiable. If not change the font or increase the font size so that they are easily readable.

If you have accomplished the test and are satisfied with the look of the PowerShell console then it is time to move on to the next Chapter.

# Chapter 3: Discover Commands with the Help System

This book will not have all the answers for all the possible PowerShell questions there is to be asked. The goal of the book is to provide the administrator relevant information on how the shell functions as a command-line utility for the purpose of better understanding its features. The book wants to make sure the PowerShell is a useful tool in automating and managing an integrated Windows system using command runs.

The book is not a manual that contains all the answers. As such, we highly recommend expert and detailed knowledge of the PowerShell's most important discoverability feature – **the help system**.

The PowerShell Help System provides detailed description of the Windows PowerShell functions, scripts, modules and cmdlets or commands. It has the ability to explain the intricacies of the Windows PowerShell language as well as explain concepts of each PowerShell item, symbol, character and element.

We recommend cozying up to the Help System, spend as much time as you can with it, and really get to know the minutest detail and the tiniest bit of information that the Help System can provide. If you invest quality time with the Help System it will reward you with near expert knowledge of all of PowerShell's intricate and very useful details, making you a veritable PowerShell wizard. Plus expert knowledge of PowerShell means an easier life for every PowerShell user. It literally has all the answers to any and every PowerShell question.

You do not know the command to a task? The Help System will tell you. Do not understand the definition of certain terms? The Help System has the answers. Having problems getting the accurate syntax for a command? The Help System knows the syntax to each command. Want to create a string of individual commands but do not know how? The Help System can guide anyone through the step by step procedure of properly stringing multiple commands together.

Yes, you got it right. The Help System is the PowerShell system manual. Therefore, it is smart to read it front and back to fully understand how the shell works. That is the only way to make the experience of using it a pleasant and productive one. Otherwise you'll be stuck. So do not be stubborn and be a smart PowerShell user. Read the Help System. Master the Help System.

## Updating the Help

Now that we know how important the Help System is for the PowerShell user, time to acquire a copy. Yes, the Help System is not pre-installed in the PowerShell.

The reason being is it goes through frequent updates. So frequent in fact that it's more practical to ask users to install the Help System version **after** purchase of a PowerShell. That way they will be able to get the very latest version of the Help System on the day of purchase.

Updating the Help System should be the first order of business after acquiring a Windows PowerShell. It is the first task to accomplish before starting any work with the shell.

So let's have a go at it then. An important reminder - before typing the command Update Help on the console make sure to check if you are running the PowerShell as an administrator. Check the top bar of the console it should say, 'Administrator: Windows PowerShell'. See Figure 3.1.



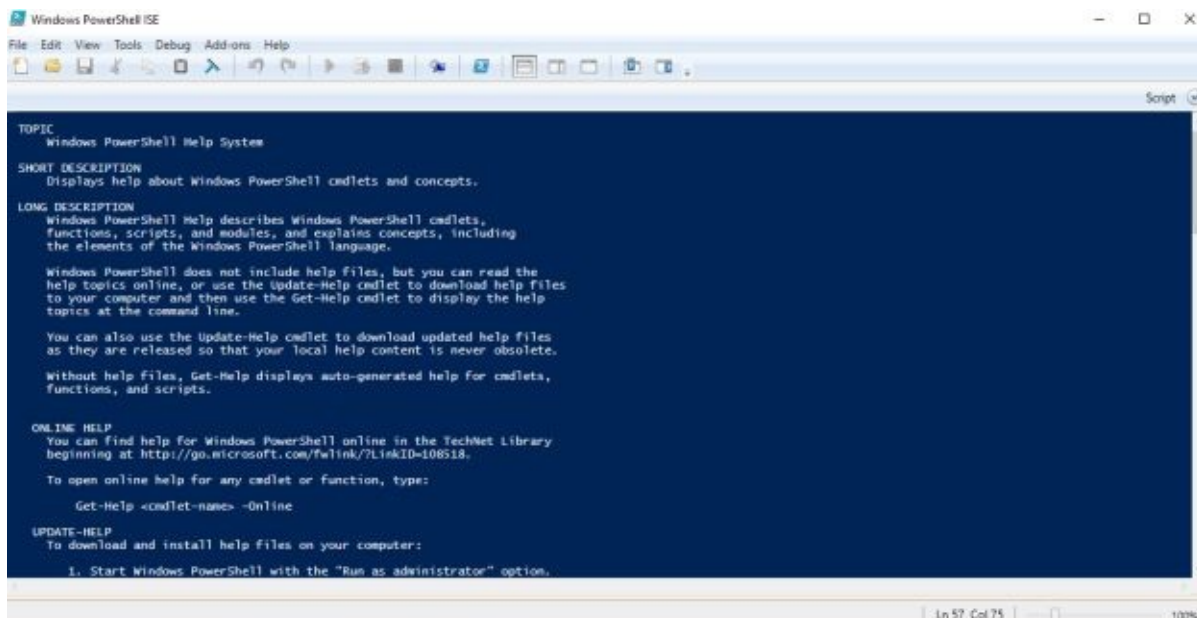
**Figure 3.1**

If it does not say 'Administrator', then run the shell as an administrator. Only if it says 'Administrator' should you type the command Update Help again. It is important that the computer is connected to the internet for this task. After typing Update Help, give the PowerShell time to update the Help System. It will take a few minutes. After that, you can proceed to work with the PowerShell.

**NOTE:** Of all the habits to develop as a PowerShell user, it is the habit of updating the Help System every month that is the most important. Remember that there will always be a constant stream of new information and features that will be generated for the PowerShell on a regular basis. Make it a habit to update help on a regular basis.

## Asking for Help

It's very simple to start using the PowerShell Help System, type in the cmdlet Get-Help on the PowerShell console. You should see all relevant information about the Help System: descriptions, online Help, update Help, get Help. All the necessary command functions for Help should be in the page also. See Figure 3.2.



**Figure 3.2**

Some sources will suggest using the command *Help* or *Man*, short for ‘manual’ as alternative commands. Both work. They register the same result as Get-Help. They are perfectly reasonable alternatives.

## Ask Help to Look for Commands

To manage expectations, the Help system does not provide information on the type of commands that are in its system. It knows it has a range of Help topics in its system. It is great however Microsoft created Help topics for all commands it has created. So expect any request for commands from Help to be answered provided of course you know what command you are looking for.

A bit of information about commands, all commands have a parameter that follows it. This is a mandatory formula. A parameter is the word, phrase or symbol that is typed after the command or cmdlet to identify the item that the shell is being asked to retrieve or search for.

For the Get-Help or Help cmdlet, the parameter is a name parameter. It means when asking for help from the Help system just type in the name of the topic you would like it to give you information on.

For example, the task is to clear an ‘event log’. You would like to ask Help for the command run or cmdlet on how to accomplish clearing an ‘event log’.

Type Help space and log or Help space event on the console. Add the wildcard symbol \* because you are searching for all possible options. If you do not add the wildcard symbol \* the result will be limited. It should look like this:

Help \*log\*

Help \*event\*

The list of results should appear on the console screen. See Figure 3.3 below.

Name	Category	Module	Synopsis
Clear-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
Get-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
Limit-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
New-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
Remove-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
Show-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
Write-EventLog	Cmdlet	Microsoft.PowerShell.M...	...
Disable-AppBackgroundTaskDiagn...	Cmdlet	AppBackgroundTask	Disable-AppBackgroundTaskDiagnosticLog...
Enable-AppBackgroundTaskDiagno...	Cmdlet	AppBackgroundTask	Enable-AppBackgroundTaskDiagnosticLog...
Get-AppxLog	Function	Appx	...
Export-BinaryMiLog	Cmdlet	CimCmdlets	Export-BinaryMiLog...
Import-BinaryMiLog	Cmdlet	CimCmdlets	Import-BinaryMiLog...
Get-MpThreatCatalog	Function	Defender	...
New-AutoLoggerConfig	Function	EventTracingManagement	...
Get-AutoLoggerConfig	Function	EventTracingManagement	...
Set-AutoLoggerConfig	Function	EventTracingManagement	...
Start-AutoLoggerConfig	Function	EventTracingManagement	...
Remove-AutoLoggerConfig	Function	EventTracingManagement	...
Get-DtcLog	Function	MsDtc	...
Reset-DtcLog	Function	MsDtc	...
Set-DtcLog	Function	MsDtc	...
Clear-PcsvDeviceLog	Function	PcsvDevice	...
Get-PcsvDeviceLog	Function	PcsvDevice	...
Get-LogProperties	Function	PSDiagnostics	...
Set-LogProperties	Function	PSDiagnostics	...
Start-StorageDiagnosticLog	Function	Storage	...
Stop-StorageDiagnosticLog	Function	Storage	...
Get-WindowsUpdateLog	Function	WindowsUpdate	...

**Figure 3.3**

Notice how all of the items in the list refer to event logs topics. Eleven of which are actual command runs or cmdlet and the rest are functions. Now choose the item that best suits your requirement. The top item Clear-Eventlog looks like it is the best choice. Now you have the accurate cmdlet to ask Help for information on the topic: Help Get-Eventlog. Follow the same formula when asking Help for information on other cmdlets.

## Tab Completion

We discussed this in the early chapters. It's good to be reminded again to get into the habit of pressing the tab when typing in the console. Tab completion allows you to complete the word you are typing to its closest match. It shows the many command, functions, scripts options available in the shell. It adds to your knowledge of what is inside the shell. If the information a tab generates is not accurate it is not necessary to type another word just press the tab again and it will cycle through the options. Stop when the tab shows the accurate word. Tab completion makes the work so much easier and faster.

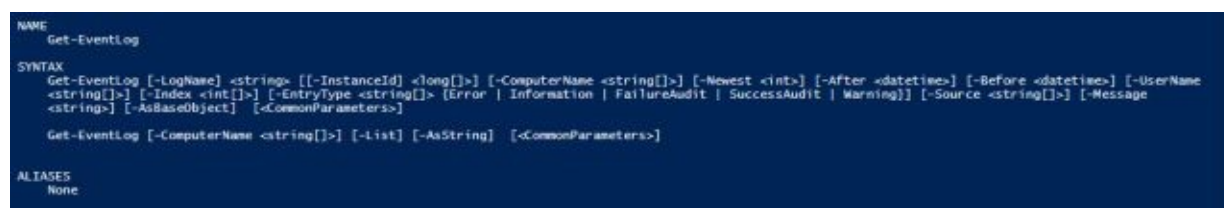
## Wildcard

Much like tab completion the \* wildcard symbol completes the entire word you're asking Help to give you information on. Just add in the \* wildcard symbol after a letter and Help will be able to narrow down the choice to the exact Help topic of the search.

For example, type into the console Help EventL\*. You'll find that Help did not go through all of the possible event log items it has in its system like it did when you typed in Help log or Help event. Instead it went directly to the Help file of EventLog. The \* wildcard symbol cuts to the chase and brings you directly to the topic.

## Summary Help

If you typed in `Help EventL*` then below is how the console screen should look like.



```
NAME
    Get-EventLog

SYNTAX
    Get-EventLog [-LogName <string>] [-InstanceId <long[]>] [-ComputerName <string[]>] [-Newest <int>] [-After <datetime>] [-Before <datetime>] [-UserName <string[]>] [-Index <int[]>] [-EntryType <string[]> [Error | Information | FailureAudit | SuccessAudit | Warning]] [-Source <string[]>] [-Message <string>] [-AsBaseObject] [<CommonParameters>]

    Get-EventLog [-ComputerName <string[]>] [-List] [-AsString] [<CommonParameters>]

ALIASES
    None
```

**Figure 3.4**

Figure 3.4 is the Summary Help. It describes the name of the command. In this case `Get-EventLog`. And it provides information on the syntax of the command. Syntax is the use of the command.

## Interpret a Help File

Looking at the summary help provided in figure 3.4 notice how there are 2 different parameter sets underneath syntax. This means there are 2 unique ways to run this particular command depending on the parameter set chosen. Parameters are all the items enclosed in a `[ ]` square bracket.

Both parameter sets starts with `Get-EventLog`. That's how we identify there are 2 sets of parameters. The first one has more parameters than the second one. Notice also how the 2 parameter sets have a parameter in common. They both have the parameter `[Computer Name]`. Conversely, there will always be special parameters that will exist only in one parameter set. `[AsString]` and `[List]` are the unique parameters in the shorter parameter set. And several parameters in the longer parameter set cannot be found in the shorter parameter set.

A command with a syntax that has 2 parameter sets means it is possible to use both sets to run the command provided you only use the shared parameters. If you use a special parameter which only exists in one parameter set in the case of the above example, `[AsString]`, then you're stuck with working with the parameters found in that parameter set. To work with multiple parameter sets and to have access to a larger pool of parameters best to avoid using unique parameters not shared by multiple parameter sets.

## Different Types of Parameters

Let's define and distinguish parameters to gain a better understanding of their function.

**Mandatory Parameters:** Any parameter that is not inside a [ ] square bracket is mandatory. In Figure 3.4 *string* is a mandatory parameter.

**Optional Parameters:** These are the parameters in [ ] square brackets. To run a command it is not mandatory to use every parameter on the set. In fact most are optional because the information can be provided by the system. Notice that in the short parameter set in Figure 3.4 all the parameters are optional.

**Positional Parameters:** Some parameters appear in all syntax language strings that it is not necessary to type them in every time a command is run. These are positional parameters. In Figure 3.4 the positional parameter is *[InstanceId]* because it has a [ ] square bracket within another set of [ ] bracket. This means you can keep that parameter but it is necessary to add value to the position <long[]>].

## Parameter Values

For information on understanding parameter values or the input that a parameter requires to run a command refer to Figure 3.5 below:

### SYNTAX

```
Get-EventLog [-AsString] [-ComputerName <string[]>] [-List] [<CommonParameters>]
```

```
Get-EventLog [-LogName] <string> [[-InstanceId] <Int64[]>] [-After <DateTime>] [-AsBaseObject] [-Before <DateTime>] [-ComputerName <string[]>] [-EntryType <string[]>] [-Index <Int32[]>] [-Message <string>] [-Newest <int>] [-Source <string[]>] [-UserName <string[]>] [<CommonParameters>]
```

## Figure 3.5 Parameter Values

First, *switches*. A switch is a parameter that is stand alone and will not require a value. In Figure 3.5 an example of a switch is [AsString]. In dealing with the full syntax of a command a switch will be clearly indicated as a switch so that administrators know not to put a value on them.

For parameters that require a value input this is how to identify them:

1. When a parameter is inside a [ ] square bracket separated by a space from another parameter.

Example: [LogName] <string>

When a parameter is stand alone and does not require a value this is how to identify them:

1. When a parameter is followed by an equal sign, colon, square bracket, etc.

Example: [AsString]



## **Types of Parameter Values**

- 1. **DateTime:** A parameter value that can be seen as a date. Therefore expect it to be a string of numbers. Example: 12-01-2012. DateTime is often dictated by the regional settings of the computer.
- 2. **String:** Combination of letters and numbers. If there is a space between a string should be encased in quotation marks.
- 3. **Int, Int64, Int32:** Int stands for integer. Not a decimal but a whole number.

## Adding value to a string with a [ ] square bracket

[ComputerName <string>[ ] >]

The [ ] square bracket in the above example means the parameter can add an array or list of strings. Example:

### ***Single Value:***

Get –EventLog Security –computer Station-A2

***Multiple Values:*** Each value should be separated by a comma & no space between.

Get –EventLog Security –computer Station-A2,AB5,Files02

***Multiple Values with space between:*** Should be enclosed in quotation marks:

Get –EventLog Security –computer “Station-A2”, “AB5”, “Files02”

***Text File Values:*** These values should be typed in single lines each.

Get –EventLog Security –computer

Station-A2

Files02

Files07

Files30

AB5

AB8

If you want to read the contents of files the most effective way to do so is to force PowerShell to show the contents via the Get-Content cmdlet. The below syntax does exactly that. Notice the command to get content is in a parenthesis. That is the symbol that can get PowerShell to execute a command within a given parameter.

Get –EventLog Security –computer (Get-Content names.txt)

## Command Help to Get Examples

If you are the sort of administrator who likes tinkering around with the systems and love to learn by trial & error then asking Help to give you examples of commands will be something of interest. This is the syntax to get Help to show command examples:

```
Help Get-EventLog -examples
```

Go ahead and test the list of cmdlet examples that Help has provided. Just make sure you are working on a computer that is not connected to your company's systems so that your Windows production line does not get affected in case you encounter errors while testing the examples list.

## Command Help to Get Common Parameters

Looking back at Figure 3.5, the example presented for interpreting parameter values. If we look closely at the 2 sets of parameters on the syntax we will notice that they both end with [`<CommonParameters>`]. [`<CommonParameters>`] is a set of eight parameters that will always be found in every cmdlet.

Let's command Help to give us more information on [`<CommonParameters>`]. Type the command below on the console:

```
Help *common*
```

The result should be `About_common_parameters`. There should not be any other options because of the `*` wildcard symbol. Plus there are no other Help topics on common parameters. That is it.

Look through `About_common_parameters` and see the 8 parameters found in [`<CommonParameters>`]. They are:

```
-Verbose
```

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

About\_common\_parameters is just one of the 'about' topics available in the Help System. If you wish to see all the other 'About' topics then type in the console:

Help about\*

Go ahead and feel free to tinker around and sift through the myriad more documentation PowerShell has tucked away in Help. Always an advantage for any administrator to have information on the piles of information that PowerShell has in storage.

## **How to Access Online Help**

From the beginning we have recommended developing the habit of updating the Help System on a monthly basis. The Help System is being constantly updated and will remain to do so for as long as PowerShell is being utilized. The best way to update Help is to access the online version of Help. It is and will always be the most updated version. Everyone involved in the development of PowerShell add and contribute to updating the Help System online. It would be good to have all that new information added to your PowerShell's Help System on a regular basis. Below is the command for accessing PowerShell Online Help:

Help Get-EventLog -online

Make it a habit to visit Online Help and update your Help System on a monthly basis. You will thank yourself for it when encountering new & complex Windows PowerShell command & scripting tasks.

## Laboratory Test

- a. Make sure you have successfully installed Update-Help.
- o. Find cmdlets that can output into files.
- o. Find cmdlets that can output to a printer.
- l. What is the command for writing an event log?
- o. Command Help to get the recent 50 entries from the security event log.

**Reminder:** Visit online sources like [MoreLunches.com](http://MoreLunches.com) to find the answers if stuck with the test. In fact much like Updating Help it is highly recommended to make it a habit to check reliable online resources on a regular basis also to get the most recent news about Windows PowerShell. Keeping abreast with everything new about PowerShell will translate to improved work efficiency.

# Chapter 4: Working with Commands – Finally!

Every systems administrator, web developer, IT professional have one thing in common. Everyone, every single one of us went through and started with one task at the onset of our careers. That task is running commands. And there is no other better computer system that provides command-line support out there but Windows PowerShell.

That is what PowerShell was developed for – to run commands. It is not accurate when people say Windows PowerShell is a scripting tool. It can do script but it was developed first and foremost to run commands. Any other additional functions it can deliver on are secondary to its ability and purpose to run command-lines.

## It's a Shell

Like other shell's, for example Cmd.exe, MS-DOS, Unix Bourne, and Unix Bash, the way to use PowerShell is very simple. Type a command, add the desired parameters to tell the command how to behave, press Enter and see the output.

To make a PowerShell script is very easy. Just type up a script and add .PS1 in the end of the file. Voila! You now have a PowerShell script. It is possible to create very complex PowerShell scripts, as complex as you would like them to be. It is also very possible for PowerShell to be able to run that script and achieve the result the script was created for.

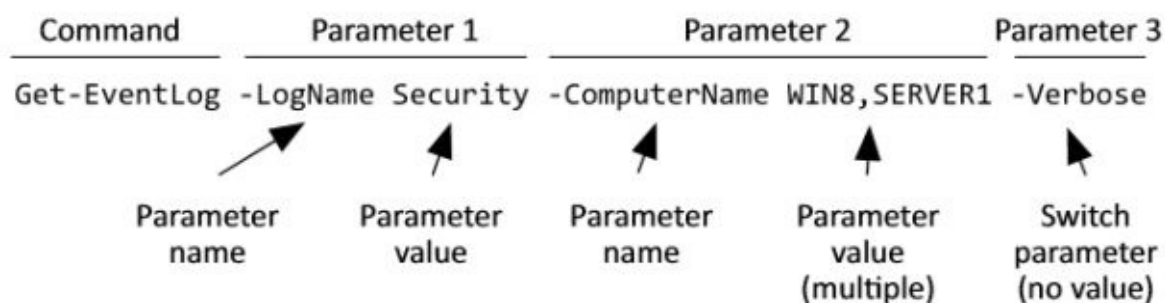
A lot of IT professionals with advanced skills have been known to throw in the most complex scripts imaginable with PowerShell being able to run the command. That is how dependable and equally complex PowerShell is. It will be able to give something back no matter what you throw at it.

But if you are a systems administrator who only wants to solve the day to day tasks generated by a company's integrated computers systems. The ability to create complex scripts is not as important as the ability to run commands that solve system questions and glitches. So mastery of the command runs of PowerShell is so much more important than challenging PowerShell with complicated script combinations.

If we are going to be realistic and practical about PowerShell getting really good at

running commands and getting familiar with PowerShell functions and features is so much more important than creating the most impressive script that can be created. Make good use of your time and understand the real priorities.

## The Elements of a Command



**Figure 4.1**

Above is a detailed listing of each element or item that makes up a command. It is imperative in the understanding of PowerShell and its functions to understand the composition of a command. So, let's start. Refer to Figure 4.1 above.

1. Command: Also referred to as the cmdlet. All cmdlets have a verb-noun format.
2. Parameter 1 Parameter Name: LogName is the parameter name of parameter 1. If a parameter name contains a punctuation or space it will be necessary to add quotation marks.
3. Parameter 1 Parameter Value: Security is the parameter value of Logname.
4. Parameter 2 Parameter Name: ComputerName is the name of Parameter 2. If a parameter name contains a punctuation or space it will be necessary to add quotation marks.
5. Parameter 2 Parameter Value: Win8 and Server1 are the values of ComputerName.
6. Parameter 3 Parameter Name: Verbose is the name of Parameter 3. It is a common parameter. It does not get a value.

## Rules for Creating a Command



1. Always add a space between the command name and the first parameter name.
2. Always start add a dash before every parameter name. Do not add a space between the dash and the parameter name.
3. Always separate the parameter name from its value with a space.
1. Always separate a value from the next parameter name with a space.
2. Typing a command will never be case-sensitive.

Again, perfection in typing a command results in desired error free result. Practice, practice, practice typing skills and know by heart the rules for creating a command. When you do working with PowerShell will always be smooth sailing.

## Terminologies

1. **Cmdlet:** It is pronounced, command-let. It is a command utility exclusive to PowerShell and therefore can only be operational in PowerShell. The framework language is .NET
2. **Function:** Almost the same as cmdlet except it is not written in the .NET framework. Instead it is written using PowerShell's own scripting language.
3. **Workflow:** It is compatible with PowerShell's workflow system. The workflow is also a function.
1. **Application:** a command-line utility which is externally executable.
2. **Command:** The term that refers to cmdlet, function, workflow, and application.

## Microsoft Recommended Way to Name Cmdlet

Start with a Verb then follow it up with a dash ( - ) then end with a Noun. Example: Get-EventLog. Having a formula for naming cmdlets makes guessing at the right command to use easier. There are only around 100 verbs accessible on PowerShell and only a few are used on a regular basis. Confirming the verb via Help or Get -Command is a quick task as well.

## The Alias

Most of the time complete command names are very long since they are often a list of words strung together. It can be a bit tedious to type a command with a string of 15 word combinations. That's where an alias comes in handy. Aliases are nicknames for commands, the abbreviated version of an otherwise very lengthy complete version. Here's how to get the alias of a command: `get alias -Definition *Get-Service*`

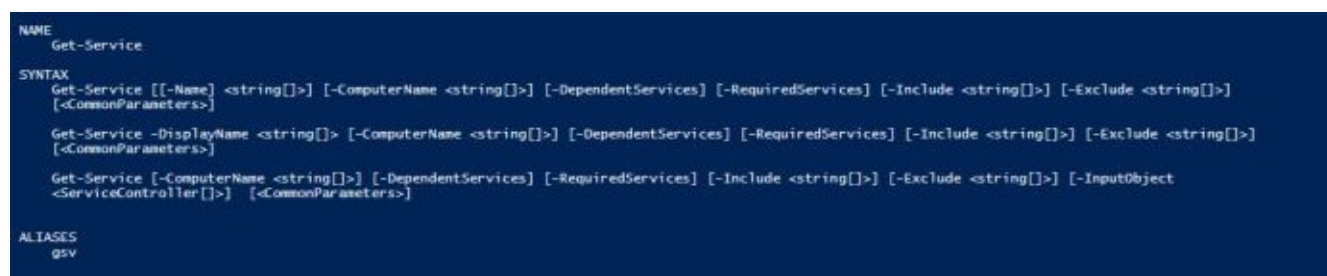
The result should look like this:

Capability	Name
cmdlet	gsv -> Get-Service

The alias of the command `Get-Service` is `gsv`. Reminder the use of an alias does not change anything about the command. Only the name gets an abbreviated version all other elements and parameters are the same the only thing different is the alias.

In dealing and working with aliases there will be instances where no matter how much you try decrypting the alias the command name it represents will escape you. During these moments ask for assistance from `Help` to clarify what command name the alias stands for. Using the example above, go ahead type: `help gsv`

The console should reflect Figure 4.2 below.



```
NAME
    Get-Service

SYNTAX
    Get-Service [[-Name <string[]>] [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include <string[]>] [-Exclude <string[]>]
    [-CommonParameters]
    Get-Service -DisplayName <string[]> [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include <string[]>] [-Exclude <string[]>]
    [-CommonParameters]
    Get-Service [-ComputerName <string[]>] [-DependentServices] [-RequiredServices] [-Include <string[]>] [-Exclude <string[]>] [-InputObject
    <ServiceController[]>] [-CommonParameters]

ALIASES
    gsv
```

**Figure 4.2**

Figure 4.2 shows very clearly the complete name of the command, the parameter sets that make up the command, and its alias. When stumped for the meaning of an alias going to `Help` and asking for information is always the best solution.

There will be certain tasks or projects that will make you wish you can create your

own aliases. Good news! You can in fact customize an alias for specific projects. Just type in New-Alias and the alias that you'd like to create. Note however that the new alias you created will only be good for the session on the day it was created. You have to create it again if you require it for the next session.

## Taking the Short Cut with Parameters

Since we are on the topic of short cuts you'd be happy to know that it is possible to create shorter versions of parameters. Here are the varied ways to do it:

**Truncating:** PowerShell has the capacity to recognize every product name even if only the first 5 letters are typed in. Just make sure to press the Tab so that PowerShell can complete typing the name and you can check the right parameter name is on the set.

**Parameter Alias:** Here is an example on how to find out the alias of a parameter name. The boldfaced words are replaceable with the command name and parameter name that you would like to check for alias in the future. PS C:\> (get-command **get-eventlog** | **select** -ExpandProperty parameters).**computername.aliases**

**Positional Parameters:** When a parameter is enclosed in a [ ] square bracket then it is a positional parameter. Here's a quick test, identify the 2 positional parameters in figure 4.3 below.

### SYNTAX

```
Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-Exclude  
<String[]>] [-Force [<SwitchParameter>]] [-Include <String[]>] [-Name  
[<SwitchParameter>]] [-Recurse [<SwitchParameter>]] [-UseTransaction  
[<SwitchParameter>]] [<CommonParameters>]
```

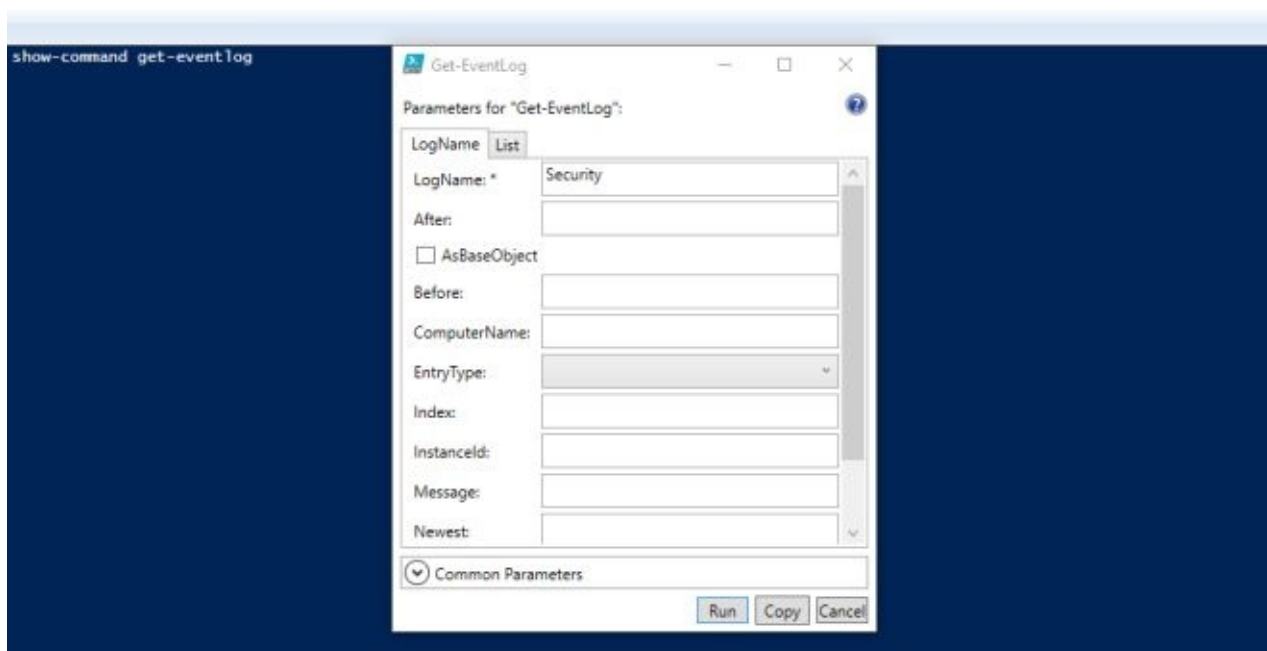
**Figure 4.3**

If you answered [-Path] and [-Filter] then you are right. When it comes to positional parameters it is not mandatory to type in a parameter name which can

take time. There is an option to keep them as they are.

## Show-Command

A graphical representation of the syntax format where you get to input the values on each parameter in a given parameter set. See Figure 4.4 provided.



**Figure 4.4** Show Command Box

Instead of manually typing every word and symbol of a syntax or parameter go to the Show-Command option. It allows entry of required fields on the box and it can either be asked to run the command or it can be copied to the console after the box has been filled up.

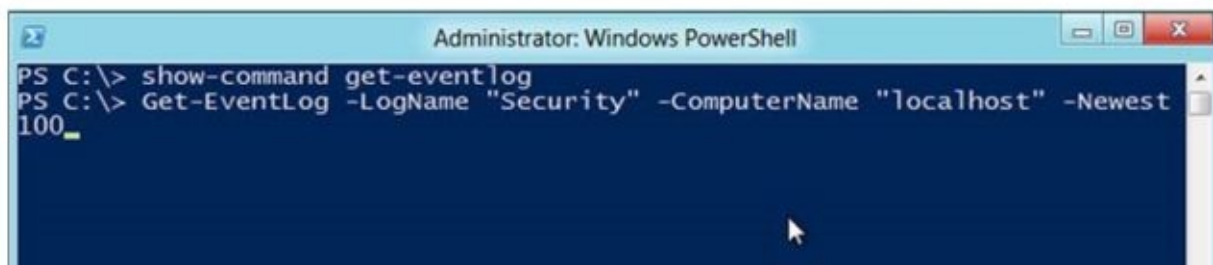
This makes work so much easier since it creates the parameters set for you. All you have to do is to input the values and names on each parameter. The possibility of an error occurring while manually typing the parameters of a syntax or parameter set is very high. The Show-Command box lessens the possibility of a mistake since it will be the one to digitally add the right symbols, names and words on the parameter set.

If you are having difficulty getting the right syntax for a parameter set, just type in show-command space, then the command name and press enter. The console

screen should look like figure 4.5. Add in the values on each box.

As mentioned earlier, there are 2 options to use show-command. The first is to run the command. After completing the requirements just click on the run tab below the box and it will run the command directly.

The second option is to copy and paste it to the console. Click on copy and go back to the shell and paste the command –ctrl + V. The great part about the 2<sup>nd</sup> option is you will be able to see the complete format of the syntax with complete values and names for each parameter. This is a great way to get familiar with a command's full and detailed syntax information. See Figure 4.5 for the complete syntax for the command get-eventlog.



**Figure 4.5**

## External Commands

Every newly purchased Windows computer will have at least 400 cmdlets built in to the system. Inside each new Windows server operating system are thousands more built in cmdlets. There is also provision in the PowerShell for administrators to create their own cmdlets that they can add into the existing ones if the need arises. Windows PowerShell was designed to be very open, inclusive, and compatible to cmdlets developed and used for other similar programs.

If an administrator has for example prior experience with Nslookup, Ipconfig, Net and the like. And have in fact sets of cmdlets from these programs, then it is very possible to export them to PowerShell and use them as they would in the previous programs or shells.

A bit of a caveat: do not expect all the external commands exported to PowerShell

to work flawlessly. Errors will be common since they are not commands native to the shell. In the past, especially with the PowerShell Version 1 & 2, it was necessary to add accurate parameters and manually make changes to the parameters of the external command so that it works seamlessly inside the shell. This was particularly true for the more complicated external commands with multiple strings of parameters.

But take heart. Those days of manually correcting external commands are over. Ever since PowerShell version 3 was released external commands that would not properly run on the PowerShell became fully functional. *It was solved by adding **2 dash lines in front** of the external command before hitting the run button.*

## Interpreting an Error Message

A screenshot of a Windows PowerShell console window titled "Administrator: Windows PowerShell". The prompt is "PS C:\>". The user has entered the command "get command". The output is a red error message: "The term 'get' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again. At line:1 char:1". Below this, there is a list of error details: "+ get command", "+ CategoryInfo : ObjectNotFound: (get:String) [], CommandNotFoundException", and "+ FullyQualifiedErrorId : CommandNotFoundException". The prompt "PS C:\>" is visible at the bottom.

```
PS C:\> get command
The term 'get' is not recognized as the name of a cmdlet, function,
script file, or operable program. Check the spelling of the name, or if a
path was included, verify that the path is correct and try again.
At line:1 char:1
+ get command
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (get:String) [], CommandNot
FoundException
+ FullyQualifiedErrorId : CommandNotFoundException

PS C:\>
```

**Figure 4.6**

PowerShell error messages are the red text that gets reflected on the console when PowerShell cannot understand the command that has been sent to it. The great thing about error messages is it is very clear about what PowerShell considers wrong and more often than not will have suggestions on how to solve the issue. Just like in the example provided above.

When PowerShell sends an error message and it is not clear how to address it there are 3 ways to get back on track:

1. Ask Help for the answers.
2. Make sure all the parameter information on the syntax is accurate.
3. Go to Show Command to check for the accurate syntax information.

## **Cmdlet Typing Rules**

Although PowerShell could not care less about whether the letters on the command are in lower or upper case it does care a whole lot about the spaces and dashes in a cmdlet or command. Get it wrong and you'll see red text plastered all over your console.

Here are the mandatory rules to memorize when typing a command or cmdlet:

1. Verb–Noun format: Every cmdlet or command will always, always have a verb plus noun format.
2. The dash: Every cmdlet or command noun should be connected with a dash.
3. No space between: Every cmdlet or command shall never be separated by a space before or after the dash.

Remember when typing a command or cmdlet the format is: verb, dash, noun.

## **Parameters Typing Rules**

The same is true for typing parameters. Watch out for the all-important space & dash. Here are the mandatory typing rules for parameters:

1. All parameters with no value get a dash in front of it.
2. A cmdlet and a parameter should have space between them.
3. A parameter and another parameter after it should have space between them.

## **PowerShell Laboratory**

Here are a few test questions to refresh and practice the lessons for Chapter 4. If



stumped for answers look up online sources for a quick look.

1. Gather a list together of all cmdlet type of commands.
2. Provide a list of 50 aliases.
3. Find out via Help more about Windows Firewall rules.



# CHAPTER 5: Piping Function

The great thing about running commands in PowerShell is its ability to string together multiple sets of commands in a single neat syntax. In the previous chapters we discussed everything there is to know about single commands. It is now time to learn about how to bring together 2 or more sets of command syntax to command the shell to work on multiple tasks.

Piping is what we call this ability of the shell to string together multiple commands. This is because the | pipe symbol is placed in between each set of command syntax that are being put together. It is referred to as the *pipeline*.

The pipeline serves as a pathway for the previous command to pass through so that the next command can receive it and have something to work on. The second command then does the same thing. It passes through the pipeline so that the third command after it has material to process.

The pipeline is actually one of the best and distinct features of Windows PowerShell other command-run utilities are not equipped with the same capability to run multiple commands in one syntax.

## Comma-Separated values (CSV) export

Let's say that the task is to get detailed information on how a system's CPU is being utilized and information on its memory. The multiple command string for that would be:

```
Get-Process | Export-csv procs.csv
```

In the syntax above there are 2 sets of commands. The first command is *Get-Process* and the second command is *Export-csv procs.csv*.

The Get-Process command is asking PowerShell to provide information on the computer's processes. For example: CPU utilization and computer memory.

While the second command, *Export-csv procs.csv*, is asking PowerShell to export to a CSV file via | pipeline all of the Get-Process information on CPU utilization and memory.

Let's type in only the command *Get-Process* on the shell. It should result in figure 5.1 below.

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
45	5	564	2076	18	0.00	1352	coherence
29	5	612	1876	38	0.02	1436	coherence
33	6	756	1028	39	0.02	1444	coherence
100	10	2660	10848	94	2.61	1220	conhost
154	10	1620	2948	46	0.13	396	csrss
196	13	1840	3608	47	0.89	460	csrss
81	7	1084	3808	53	0.02	2056	dllhost
105	9	1616	5336	40	0.03	2820	dllhost
172	18	49016	26712	150	1.44	760	dsm
1511	95	29212	39916	425	8.20	1288	explorer
0	0	0	20	0	0.00	0	Idle
631	17	2900	5796	35	0.58	556	lsass
446	30	56320	15380	181	22.33	1596	MsMpEng
520	38	104620	111024	699	9.09	1776	powershell
276	26	3792	8368	105	0.41	3008	prl_cc
121	11	1612	4332	76	0.08	1476	prl_tools
90	11	1228	3344	51	0.05	1424	prl_tools_ser...
83	10	3868	7892	91	0.31	812	regedit
491	29	14480	8180	615	0.20	2500	SearchIndexer
195	11	3452	5348	32	0.98	548	services
36	2	280	788	4	0.05	288	smss
328	16	3048	5820	47	0.09	1080	spoolsv
583	37	13512	14056	1386	2.13	404	svchost
295	12	2116	6240	36	0.13	632	svchost
313	14	2708	5372	34	0.55	676	svchost
635	26	14036	12976	118	0.53	736	svchost
319	23	13244	9668	93	5.05	856	svchost
574	28	7736	8748	133	0.89	892	svchost
1071	44	11628	13988	134	3.17	932	svchost

**Figure 5.1**

The set of information on the table in Figure 5.1 tells of CPU utilization and systems memory use.

Now, to ask PowerShell to export it to a CSV file so that it can be transferred to a Microsoft excel file for presentation purposes, type in the command run below:

Get-Process | Export-csv c:\procs.csv

Let's look at the detailed information on computer processes that the dual

command `Get-Process | Export-CSV` is able to provide by previewing the list on notepad.

Figure 5.2 should be the result on the console:



**Figure 5.2**

The first line distinguishes the information provided as 'Systems Diagnostics Process'. The second line identifies the nature of the information per table. The third line provides a detailed listing of processes information running in the computer. Above is the information extracted from the dual command-run `Get-Process | Export-CSV c:\procs.csv`.

Now make sure to save that information into a CSV file. Once that's accomplished the file is ready to share with any colleague who requires the information transferred to a presentation friendly Microsoft excel file.

Is it possible to import the CSV file to PowerShell? Of course it is. Just type in `Import-CSV c:\procs.csv` to the shell and the result should look much the same way as in Figure 5.2.



## Extensible Mark Up Language (XML) Export

Another way to export and import files is via XML or Extensible Mark Up Language. Some administrators prefer using XML because it is not a flat file like CSV and more often than not it generates a richer array of information than CSV.

To export an XML file the command run is much the same except the 2<sup>nd</sup> command run is different: `Get-Process | Export-cliXML C:\procs.csv`

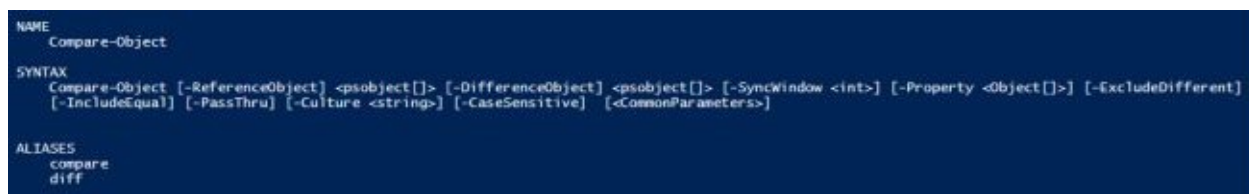
To import the file the command run should be: `Get-Process | Import-cliXML C:\procs.csv`

Suggest testing the XML export command run for familiarity's sake. Make sure to run the import XML command also and check the result on the Notepad to see if the process information can be previewed.

## The Compare Object Command

PowerShell has a very nifty feature called the compare object command. It allows file comparisons between 2 sets of files. With PowerShell it is possible for example to compare the CSV & XML generated process information recently worked on, place them side by side and compare the differences between the 2 files.

Type the command `Help Compare-object` on the shell. See command run result in Figure 5.3.

A screenshot of a PowerShell terminal window with a dark blue background and white text. It displays the help information for the 'Compare-Object' command. The text is organized into sections: 'NAME' (Compare-Object), 'SYNTAX' (a detailed command line with various parameters in angle brackets), and 'ALIASES' (listing 'compare' and 'diff').

```
NAME
    Compare-Object

SYNTAX
    Compare-Object [-ReferenceObject] <psobject[]> [-DifferenceObject] <psobject[]> [-SyncWindow <int>] [-Property <Object[]>] [-ExcludeDifferent]
    [-IncludeEqual] [-PassThru] [-Culture <string>] [-CaseSensitive] [-CommonParameters]

ALIASES
    compare
    diff
```

**Figure 5.3**

Compare-object has 2 aliases indicated above. They are compare and diff. This

means there are 2 additional possible command runs for compare-object, and they are the aliases diff & compare.

Compare-object is self-explanatory. It is the ability of PowerShell to take similar information from 2 computers and bring them together in one file so that the difference between the 2 sets of information can be determined.

Close scrutiny of the syntax shows 3 important parameters of compare-object – *referenceobject*, *differenceobject*, and *property*.

For example, the task is to compare the process information from 2 computers. The first computer will be referred to as the reference computer where the *referenceobject* will be extracted from. Because it is called *referenceobject*, this means the information it has is the standard for accuracy which the *differenceobject* will be measured against.

The second computer will be the difference computer where the *differenceobject* will come from. The information from *differenceobject* is what's being checked for accuracy.

To exercise the compare-object command run 2 sets of computers will be required. Once you have 2 computers identify which computer will be the reference computer. Remember this is the computer with the accurate information or the baseline standard. Then run this command on the reference computer: Get-Process | Export-cliXML reference.xml

The next step is to transfer the file to the 2<sup>nd</sup> computer by running this command on computer 2: Diff –reference (Import-cliXML reference.xml) –difference (Get-Process) –property name

The command is a bit complicated since it is a combination of a series of commands. A bit of explanation, every time a parenthesis is used in a command syntax that means the command inside the parenthesis will be executed first and the processed information sent to the command run before it.

In the example above since Import-cliXML reference.xml is in parenthesis PowerShell will import the reference cliXML first before sending it to Diff –



reference. The same is true for the command (Get-Process). PowerShell will process it first before sending it to –difference.

The parameter –property name at the end tells PowerShell to show only the processes that are different in the reference set and the difference set. <= is the indicator for the process that can only be found in the reference set. => is the indicator for the processes that are found only in the difference set. See figure 5.3 for graphic example.

```
PS C:\> diff -reference (import-clixml reference.xml) -difference (get
-process) -property name
```

name	SideIndicator
----	-----
calc	=>
mspaint	=>
notepad	=>
conhost	<=
powershell_ise	<=

**Figure 5.4**

The above table indicates that the difference computer or computer 2 has calc, mspaint, and notepad while the reference computer or computer 1 does not. And while computer 1 or the reference computer has conhost and powershell\_ise the difference computer does not.

For administrators, the compare-object command run of the PowerShell can be very useful for checking on a regular basis the internal workings of an array of computers against a standard baseline or reference object. PowerShell's compare-object command run therefore is the perfect tool for efficient systems management.

## PowerShell Out Cmdlets

PowerShell has several cmdlets that allows for output of commands to files, printer, or even the option to delete or nullify. These are referred to as the out cmdlets. Go to the shell and type in Help Out\*. The following out cmdlets should be the result

1. **Out-null:** The object in the command syntax whether single command or pipeline command that will be nullified or deleted.

Example: PS C:\>get-newitem | out-null

2. **Out-default:** The object in the command syntax that will be sent to the host or the shell. The information reflected on the shell.

3. **Out-host:** Similar to Out-Default.

4. **Out-file:** Processing a list of items generated from a command so that the output is a .txt file.

Example:

```
PS C:\ get-process | outfile -filepath C:\docs\process.txt
```

```
PS C:\ get-process | out-file C:\docs\process.txt
```

5. **Out-printer:** Asking PowerShell to print a .txt file via pipeline.

Example:

```
PS C:\ get-content $pshome\about_signing.help.txt |
```

Out-Printer

6. **Out-string:** Searching for and manipulating data to output as string. Example 1 sends a .txt file to the shell as a single string. Example 2 takes regional settings information from a computer and converts it to strings.

Example:

1. PS C:\ get-content C:\docs\test1.txt | out-string
2. PS C:\> \$cult = get-culture | select-object \* out-string -inputobject \$cult -width 100

## The Convert Command

Before discussing the convert command-run of PowerShell it is necessary to differentiate between the export command and the convert command.

Export in PowerShell terms means converting data into a format say CSV, then saving that converted format into a file for storage. Convert on the other hand means conversion to a different format only not necessarily saving it into a file.

Is it possible to convert a command to an HTML format? Yes. The command syntax would be:

```
Get-Service | ConvertTo-HTML
```

The command will convert Service to its HTML format. Is it possible to export the Get-Service HTML format to a .txt file? Yes it is.

As a test, determine the pipeline command that will enable saving the Get-Service HTML format to .txt. The answer is accurate if you typed in:

```
Get-Service | ConvertTo-HTML | Out-file services.html
```

The service HTML data should be saved to a .txt file format easily with the above pipeline command run.

Other ConvertTo- cmdlets available on the PowerShell are:

1. **ConvertTo-CSV:** The command converts any data on the shell to a CSV format.
2. **ConvertTo-Json:** Converts any object to a JavaScript Object Notation (Json) string format. The conversion results in properties becoming field names, removal of methods and conversion of field values into property values.
3. **ConvertTo-XML:** For conversion of pipelined objects to an XML based representation. Will only work if there are multiple commands in a pipeline.

## Dangerous! cmdlets

Crashing a computer system is a valid option if for example the user would like all the data & systems inside the computer to never be seen again. That is always a valid option if the purpose is to clean up a computer and wash your hands clean of what is inside it forever.

But that is a rare instance and the reasoning behind should be so valid as to completely annihilate all the information acquired and stored in the device. More often than not, we would like to extend the life span of our computers devices for as long as possible. After all, the computer is such a personal device and typically contains some of the most important personal and work related information people cannot do without.

Therefore, understanding the cmdlets that pose a danger to the continued functionality of computers is very important. This will prevent the unfortunate possibility of a computer crash.

The dangerous cmdlets to avoid running in the shell are:

1. Get-Process | Stop-Process
2. Get-Service | Stop-Service

**Please do not test run the commands.** Run them in the shell and it will proceed shutting down your computer permanently. **DO NOT** ask the shell to run these commands ever.

If for some reason you have accidentally run the command PowerShell has a failsafe device that will ask, 'Are you sure?' before proceeding with the Stop-Process command.

This is because PowerShell is designed to measure the *impact level* of every command. This is to make sure that when the stop command is requested they do not pose any threat to the integrity of the computer data and systems. To check for a computers impact level settings run this command on the shell:

```
PS C:\ $confirmpreference
```

The preferred resulting answer should be *High*. This means when PowerShell runs a command that is detrimental to the integrity of the computer systems and data PowerShell will first ask, 'Are you sure?' before running the command. This of course provides the option to abort the command and save the computer from crashing.

Make sure every computer's PS C:\ \$confirmpreference settings is set on high to ensure it is safe from inadvertent shut down.

## PowerShell Laboratory:

Here are a few more practical tests to run on the shell recapping the lessons in this chapter:

1. What is the difference between a CSV and XML format? Run a command that exports to CSV.
2. Run a pipeline command and print the file into a printer.
3. Convert a command pipeline into Json.
4. Export a command pipeline into a .txt file.
5. Use convert & export in one pipeline command run.



# Conclusion

Congratulations! You have completed the book. Hopefully it has provided functional basic information you require to successfully run commands on PowerShell. Based on the lessons from the 5 chapters you should already be equipped to tinker inside the Help System of PowerShell for any information you require about cmdlets, functions, modules and scripts. Remember if unsure about what a cmdlet, script, or terms means Help has all the definitions that will inform on the matter.

Commands have a specific formula that needs to be entered in the console accurately so that the shell runs the right command to generate the right result. The best way to get good at perfecting the command formula is to keep on practicing your typing skills. Remember PowerShell has 400++ built in commands that you can work on. It is to the advantage of any administrator working with the PowerShell to know all of them to make accomplishing tasks and systems management a more effective work process.

Knowing all of the hundreds of cmdlets available inside the PowerShell system will allow the user to build pipeline commands that can be very beneficial to fast tracking tasks. They allow PowerShell to accomplish tasks and deliver results consecutively. Have fun with the pipeline command feature of PowerShell. It's a wonderful skill to acquire and shows in-depth understanding of PowerShell cmdlets.

Most important of all, remember not to run any Stop commands on the shell to prevent the systems from completely shutting down.

And as important as not commanding PowerShell to go through a Stop run is developing the habit of Updating the Help System. Help is being regularly and constantly updated by its creators and even actual PowerShell users who contribute new updates every day.

The only way to keep up with the ever-evolving Windows PowerShell is to have a regular schedule in updating its Help System. It is after all the lifeline of any systems administrator caught in a bind with a PowerShell task. Always remember Help will have the answer, provided of course you keep it updated.

As mentioned in the early chapters of the book there are a range of resources available online if you would like more advanced information on PowerShell. Just visit the many number of websites and online resources that offer the latest news and information about Windows PowerShell.

Good luck!