BASICS WITH WINDOWS POWERSHELL

By Prometheus MMS

Copyright © 2016 by Prometheus MMS

All rights reserved. No part of the book may be reproduced in any form by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from the author or publisher, except by a reviewer who may quote brief passages in a review.

Published by Amazon kindle LLC

For more contact: mmsprometheus@gmail.com

Twitter: https://twitter.com/prometheus_mms

BASICS WITH WINDOWS POWERSHELL

INTRODUCTION

1.DATES AND TIMES

1.1 CHANGING A COMPUTER'S DATE AND TIME

1.2 LISTING DATE AND TIME INFORMATION

1.3 PERFORMING DATE ARITHMETIC

2. FILES AND FOLDERS

2.1 CREATING A NEW FILE OR FOLDER

2.2 DELETING A FILE OR FOLDER (OR OTHER TYPE OF OBJECT)

2.3 MOVING A FILE OR FOLDER

2.4 RENAMING A FILE OR FOLDER

2.5 REPLICATING (AND EXTENDING) THE DIR COMMAND

2.6 RETRIEVING A SPECIFIC ITEM

2.7 VERIFYING THE EXISTENCE OF A FILE OR FOLDER

3. SAVING AND IMPORTING DATA

3.1 APPENDING DATA TO A TEXT FILE

3.2 CHECKING FOR THE EXISTENCE OF A STRING VALUE

3.3 DISPLAY DATA AND SAVE THAT DATA WITH ONE COMMAND

3.4 Erasing the Contents of a File

3.5 Saving Data as an HTML File

3.6 Reading a Text File

3.7 Read in a Comma-Separated Values File

3.8 Reading in an XML File

3.9 Saving Data as a Comma-Separated Values File

3.10 Saving Data as an XML File

3.11 Saving Data Directly to a Text File

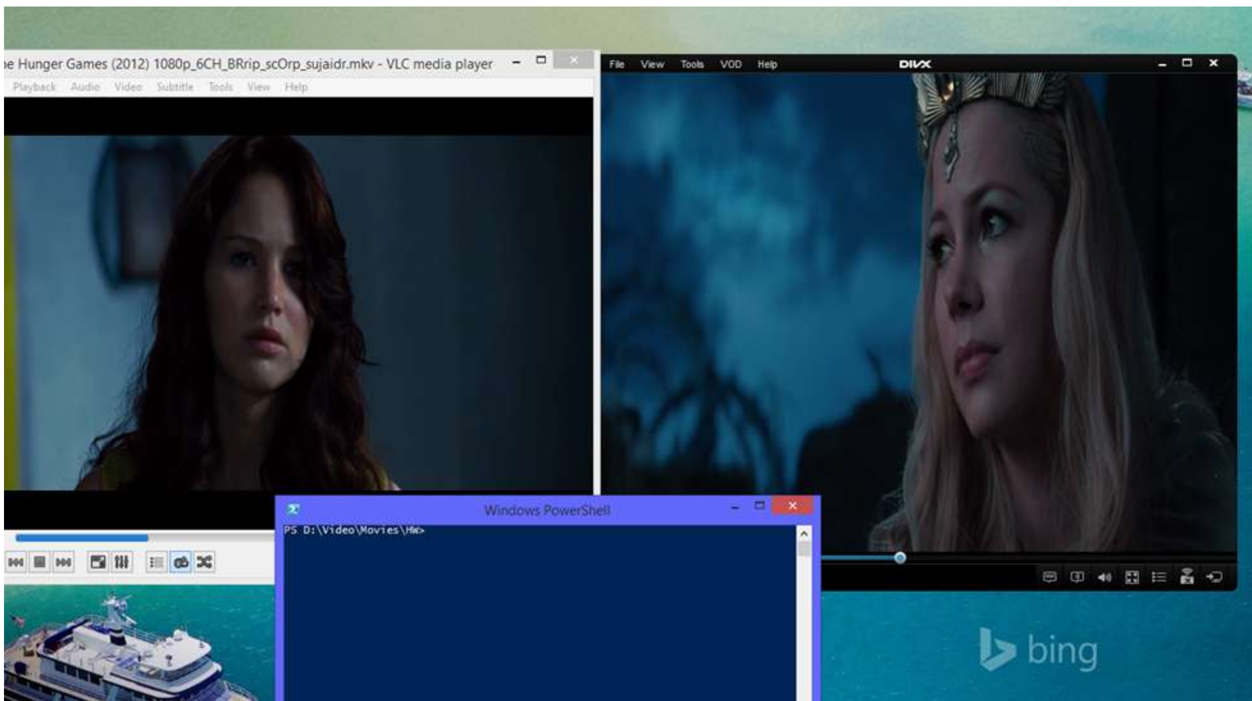
3.12 Saving Data to a Text File

4. A-Z Commands for Command Prompt

INTRODUCTION

Power shell Commands for Dates and Times Files and Folders, Saving and Importing Data

Microsoft's new command shell/scripting language. This series provides a task-based introduction to Windows PowerShell cmdlets: rather than focusing on the individual cmdlets themselves, the emphasis is on the tasks you can carry out using those cmdlets. These tasks include everything from reading and writing text files to managing event logs to sorting and filtering data.



Applies To: Windows PowerShell 2.0, Windows PowerShell 3.0, Windows PowerShell 4.0

Graphical user interfaces use some basic concepts that are well known to most computer users. Users rely on the familiarity of those interfaces to accomplish tasks. Operating systems present users with a graphical representation of items that can be

browsed, usually with drop-down menus for accessing specific functionality and context menus for accessing context-specific functionality.

A command-line interface (CLI), such as Windows PowerShell, must use a different approach to expose information, because it does not have menus or graphical systems to help the user. You need to know command names before you can use them. Although you can type complex commands that are equivalent to the features in a GUI environment, you must become familiar with commonly-used commands and command parameters.

Most CLIs do not have patterns that can help the user to learn the interface. Because CLIs were the first operating system shells, many command names and parameter names were selected arbitrarily. Terse command names were generally chosen over clear ones. Although help systems and command design standards are integrated into most CLIs, they have been generally designed for compatibility with the earliest commands, so the command set is still shaped by decisions made decades ago.

Windows PowerShell was designed to take advantage of a user's historic knowledge of CLIs. In this chapter, we will talk about some basic tools and concepts that you can use to learn Windows PowerShell quickly. They include:

- Using Get-Command
- Using Cmd.exe and UNIX commands
- Using External Commands
- Using Tab-Completion
- Using Get-Help

for example, they mention that using double quotes you can evaluate variables inside the strings:

$x = "hello"

"$x world"

results in "hello world"

but

$y = "hello","goodbye"

"$y[0] world"

does not. So, trial and error commence and we try:

"${y[0]} world" which also fail.

It would be a lot better if they bothered to explain exactly how the various basic tokens are built up, for example what goes into a script block? will for example:

$x="hello"

0..4 | foreach-object -process { "$x world $_" }

Give the expected result? Yes, it does!

But for some reason using $args inside foreach does not - because presumably foreach-process has it's own arguments which replaces the $args variable, probably best then to store the $args into another variable and then use it? Yes, that works.

Things would be a lot easier if they bothered to have a few chapters to explain these basic stuff rather than having each of us finding out by experimentation.

Seems they skip immediately to the fun part and drop the tedious basic building blocks of the scripting language.

Also, if they could allow a character to escape the meaning of $ inside double quoted strings so you could write stuff like:

```
"\$x = '$x"'
```

To get the output

```
$x = 'Hello'
```

but that is not just documentation, that is change of the language. Yes, I know you can get the same
string by doing:

```
("$"+"x = '$x"')
```

but it is easier to write "\$x = '$x'" to achieve the same thing.

About Windows PowerShell

Discoverability

Windows PowerShell makes it easy to discover its features. For example, to find a list of cmdlets that view and change Windows services, type:

get-command *-service

After discovering which cmdlet accomplishes a task, you can learn more about the cmdlet by using the Get-Help cmdlet. For example, to display help about the Get-Service cmdlet, type:

get-help get-service

To fully understand the output of that cmdlet, pipe its output to the Get-Member cmdlet. For example, the following command displays information about the members of the object output by the Get-Service cmdlet.

get-service | get-member

Consistency

Managing systems can be a complex endeavor and tools that have a consistent interface help to control the inherent complexity. Unfortunately, neither command-line tools nor scriptable COM objects have been known for their consistency.

The consistency of Windows PowerShell is one of its primary assets. For example, if you learn how to use the Sort-Object cmdlet, you can use that knowledge to sort the output of any cmdlet. You do not have to learn the different sorting routines of each cmdlet.

In addition, cmdlet developers do not have to design sorting features for their cmdlets. Windows PowerShell gives them a framework that provides the basic features and forces them to be consistent about many aspects of the interface. The framework

eliminates some of the choices that are typically left to the developer, but, in return, it makes the development of robust and easy-to-use cmdlets much simpler.

Interactive and Scripting Environments

Windows PowerShell is a combined interactive and scripting environment that gives you access to command-line tools and COM objects, and also enables you to use the power of the .NET Framework Class Library (FCL).

This environment improves upon the Windows Command Prompt, which provides an interactive environment with multiple command-line tools. It also improves upon Windows Script Host (WSH) scripts, which let you use multiple command-line tools and COM automation objects, but do not provide an interactive environment.

By combining access to all of these features, Windows PowerShell extends the ability of the interactive user and the script writer, and makes system administration more manageable.

Object Orientation

Although you interact with Windows PowerShell by typing commands in text, Windows PowerShell is based on objects, not text. The output of a command is an object. You can send the output object to another command as its input. As a result, Windows PowerShell provides a familiar interface to people experienced with other shells, while introducing a new and powerful command-line paradigm. It extends the concept of sending data between commands by enabling you to send objects, rather than text.

Understanding Important Windows PowerShell Concepts

Commands Are Not Text-Based

Unlike traditional command-line interface commands, Windows PowerShell cmdlets are designed to deal with objects - structured information that is more than just a string of characters appearing on the screen. Command output always carries along extra

information that you can use if you need it. We will discuss this topic in depth in this document.

If you have used text-processing tools to process command-line data in the past, you will find that they behave differently if you try to use them in Windows PowerShell. In most cases, you do not need text-processing tools to extract specific information. You can access portions of the data directly by using standard Windows PowerShell object manipulation commands.

The Command Family Is Extensible

Interfaces such as Cmd.exe do not provide a way for you to directly extend the built-in command set. You can create external command-line tools that run in Cmd.exe, but these external tools do not have services, such as Help integration, and Cmd.exe does not automatically know that they are valid commands.

The native binary commands in Windows PowerShell, known as cmdlets (pronounced command-lets), can be augmented by cmdlets that you create and that you add to Windows PowerShell by using snap-ins. Windows PowerShell snap-ins are compiled, just like binary tools in any other interface. You can use them to add Windows PowerShell providers to the shell, as well as new cmdlets.

Because of the special nature of the Windows PowerShell internal commands, we will refer to them as cmdlets.

Note:

Windows PowerShell can run commands other than cmdlets. We will not be discussing them in detail in the Windows PowerShell User's Guide, but they are useful to know about as categories of command types. Windows PowerShell supports scripts that are analogous to UNIX shell scripts and Cmd.exe batch files, but have a .ps1 file name extension. Windows PowerShell also allows you to create internal functions that can be used directly in the interface or in scripts.

Windows PowerShell Handles Console Input and Display

When you type a command, Windows PowerShell always processes the command-line input directly. Windows PowerShell also formats the output that you see on the screen. This is significant because it reduces the work required of each cmdlet and ensures that you can always do things the same way regardless of which cmdlet you are using. One example of how this simplifies life for both tool developers and users is command-line Help.

Traditional command-line tools have their own schemes for requesting and displaying Help. Some command-line tools use /? to trigger the Help display; others use -?, /H, or even //. Some will display Help in a GUI window, rather than in the console display. Some complex tools, such as application updaters, unpack internal files before displaying their Help. If you use the wrong parameter, the tool might ignore what you typed and begin performing a task automatically. When you enter a command in Windows PowerShell, everything you enter is automatically parsed and pre-processed by Windows PowerShell. If you use the -? parameter with a Windows PowerShell cmdlet, it always means "show me Help for this command". Cmdlet developers do not have to parse the command; they only need to provide the Help text.

It is important to understand that the Help features of Windows PowerShell are available even when you run traditional command-line tools in Windows PowerShell. Windows PowerShell processes the parameters and passes the results to the external tools.

Note:

If you run angraphic application in Windows PowerShell, the window for the application opens. Windows PowerShell intervenes only when processing the command-line input you supply or the application output returned to the console window; it does not affect how the application works internally.

Windows PowerShell Uses Some C# Syntax

Windows PowerShell has syntax features and keywords that are very similar to those used in the C# programming language, because Windows PowerShell is based on the .NET Framework. Learning Windows PowerShell will make it much easier to learn C#, if you are interested in the language.

If you are not a C# programmer, this similarity is not important. However, if you are already familiar with C#, the similarities can make learning Windows PowerShell much easier.

Easy Transition to Scripting

Windows PowerShell makes it easy to transition from typing commands interactively to creating and running scripts. You can type commands at the Windows PowerShell command prompt to discover the commands that perform a task. Then, you can save those commands in a transcript or a history before copying them to a file for use as a script.

Understanding Important Windows PowerShell Concepts

Commands Are Not Text-Based

Unlike traditional command-line interface commands, Windows PowerShell cmdlets are designed to deal with objects - structured information that is more than just a string of characters appearing on the screen. Command output always carries along extra information that you can use if you need it. We will discuss this topic in depth in this document.

If you have used text-processing tools to process command-line data in the past, you will find that they behave differently if you try to use them in Windows PowerShell. In most cases, you do not need text-processing tools to extract specific information. You can access portions of the data directly by using standard Windows PowerShell object manipulation commands.

The Command Family Is Extensible

Interfaces such as Cmd.exe do not provide a way for you to directly extend the built-in command set. You can create external command-line tools that run in Cmd.exe, but

these external tools do not have services, such as Help integration, and Cmd.exe does not automatically know that they are valid commands.

The native binary commands in Windows PowerShell, known as cmdlets (pronounced command-lets), can be augmented by cmdlets that you create and that you add to Windows PowerShell by using snap-ins. Windows PowerShell snap-ins are compiled, just like binary tools in any other interface. You can use them to add Windows PowerShell providers to the shell, as well as new cmdlets.

Because of the special nature of the Windows PowerShell internal commands, we will refer to them as cmdlets.

Note:

Windows PowerShell can run commands other than cmdlets. We will not be discussing them in detail in the Windows PowerShell User's Guide, but they are useful to know about as categories of command types. Windows PowerShell supports scripts that are analogous to UNIX shell scripts and Cmd.exe batch files, but have a .ps1 file name extension. Windows PowerShell also allows you to create internal functions that can be used directly in the interface or in scripts.

Windows PowerShell Handles Console Input and Display

When you type a command, Windows PowerShell always processes the command-line input directly. Windows PowerShell also formats the output that you see on the screen. This is significant because it reduces the work required of each cmdlet and ensures that you can always do things the same way regardless of which cmdlet you are using. One example of how this simplifies life for both tool developers and users is command-line Help.

Traditional command-line tools have their own schemes for requesting and displaying Help. Some command-line tools use /? to trigger the Help display; others use -?, /H, or even //. Some will display Help in a GUI window, rather than in the console display. Some complex tools, such as application updaters, unpack internal files before displaying their Help. If you use the wrong parameter, the tool might ignore what you typed and begin performing a task automatically. When you enter a command in Windows PowerShell, everything you enter is automatically parsed and pre-processed

by Windows PowerShell. If you use the -? parameter with a Windows PowerShell cmdlet, it always means "show me Help for this command". Cmdlet developers do not have to parse the command; they only need to provide the Help text.

It is important to understand that the Help features of Windows PowerShell are available even when you run traditional command-line tools in Windows PowerShell. Windows PowerShell processes the parameters and passes the results to the external tools.

Note:

If you run an graphic application in Windows PowerShell, the window for the application opens.

Windows PowerShell intervenes only when processing the command-line input you supply or the application output returned to the console window; it does not affect how the application works internally.

Windows PowerShell Uses Some C# Syntax

Windows PowerShell has syntax features and keywords that are very similar to those used in the C# programming language, because Windows PowerShell is based on the .NET Framework. Learning Windows PowerShell will make it much easier to learn C#, if you are interested in the language.

If you are not a C# programmer, this similarity is not important. However, if you are already familiar with C#, the similarities can make learning Windows PowerShell much easier.

Learning Windows PowerShell Names

Learning names of commands and command parameters is a significant time investment with most command-line interfaces. The issue is that there are very few patterns, so the only way to learn is by memorizing each command and each parameter that you need to use on a regular basis.

When you work with a new command or parameter, you cannot generally use what you already know; you have to find and learn a new name. If you look at how interfaces grow from a small set of tools with incremental additions to functionality, it

is easy to see why the structure is nonstandard. With command names in particular, this may sound logical since each command is a separate tool, but there is a better way to handle command names.

Most commands are built to manage elements of the operating system or applications, such as services or processes. The commands have a variety of names that may or may not fit into a family. For example, on Windows systems, you can use the net start and net stop commands to start and stop a service. There is another more generalized service control tool for Windows that has a completely different name, sc, that does not fit into the naming pattern for the net service commands. For process management, Windows has the tasklist command to list processes and the taskkill command to kill processes.

Commands that take parameters have irregular parameter specifications. You cannot use the net start command to start a service on a remote computer. The sc command will start a service on a remote computer, but to specify the remote computer, you must prefix its name with a double backslash. For example, to start the spooler service on a remote computer named DC01, you would type sc \\DC01 start spooler. To list tasks running on DC01, you need to use the /S (for "system") parameter and supply the name DC01 without backslashes, like this: tasklist /S DC01.

Although there are important technical distinctions between a service and a process, they are both examples of manageable elements on a computer that have a well-defined life cycle. You may want to start or stop a service or process, or get a list of all currently running services or processes. In other words, although a service and a process are different things, the actions we perform on a service or a process are often conceptually the same. Furthermore, choices we may make to customize an action by specifying parameters may be conceptually similar as well. Windows PowerShell exploits these similarities to reduce the number of distinct names you need to know to understand and use cmdlets.

Cmdlets Use Verb-Noun Names to Reduce Command Memorization

Windows PowerShell uses a "verb-noun" naming system, where each cmdlet name consists of a standard verb hyphenated with a specific noun. Windows PowerShell verbs are not always English verbs, but they express specific actions in Windows PowerShell. Nouns are very much like nouns in any language, they describe specific

types of objects that are important in system administration. It is easy to demonstrate how these two-part names reduce learning effort by looking at a few examples of verbs and nouns.

Nouns are less restricted, but they should always describe what a command acts upon. Windows PowerShell has commands such as Get-Process, Stop-Process, Get-Service, and StopService.

In the case of two nouns and two verbs, consistency does not simplify learning that much. However, if you look at a standard set of 10 verbs and 10 nouns, you then have only 20 words to understand, but those words can be used to form 100 distinct command names. Frequently, you can recognize what a command does by reading its name, and it is usually apparent what name should be used for a new command. For example, a computer shutdown command might be Stop-Computer. A command that lists all computers on a network might be Get-Computer. The command that gets the system date is Get-Date.

You can list all commands that include a particular verb with the -Verb parameter for GetCommand (We will discuss Get-Command in detail in the next section). For example, to see all cmdlets that use the verb Get, type:

```
PS> Get-Command -Verb Get
CommandType     Name                          Definition
-----------     ----                          ----------
Cmdlet          Get-Acl                       Get-Acl [[-Path] <String[]>]...
Cmdlet          Get-Alias                     Get-Alias [[-Name] <String[]...
Cmdlet          Get-AuthenticodeSignature     Get-AuthenticodeSignature [-... Cmdlet
Get-ChildItem                 Get-ChildItem [[-Path] <Stri...
...
```

The -Noun parameter is even more useful because it allows you to see a family of commands that affect the same type of object. For example, if you want to see which commands are available for managing services, type following command:

```
PS> Get-Command -Noun Service
CommandType     Name                          Definition
-----------     ----                          ----------
```

Thank You for previewing this eBook

You can read the full version of this eBook in different formats:

- HTML (Free /Available to everyone)
- PDF / TXT (Available to V.I.P. members. Free Standard members can access up to 5 PDF/TXT eBooks per month each month)
- Epub & Mobipocket (Exclusive to V.I.P. members)

To download this full book, simply select the format you desire below